C

Christian Zahler

1 C als Entwicklungswerkzeug

1.1 Geschichtliche Entwicklung von C:

C wurde 1972 in den Bell-Laboratories von AT&T von Dennis Ritchie und Brian Kernighan im Zusammenhang mit dem Betriebssystem UNIX entwickelt. C wurde aus der Sprache B (Ken Thompson 1970) entwickelt, diese wieder aus der "Basic Combined Programming Language" BCPL, die auf Großrechnern heute noch verwendet wird.

MS entwickelte QuickC und Visual C++, BORLAND die Varianten "TURBO-C" und "C++". Visual C++ und BORLAND C++ erlauben die Erstellung von Programmen für die MS-Windows-Welt. Internet-Applikationen können mit Java und dessen Derivaten J und Visual J++ erstellt werden, die aber auf der C-Syntax aufbauen.

Folgende drei Sprachstandards von C sind derzeit üblich:

- **K&R-Standard**: Ursprüngliches C nach Kernighan und Ritchie.
- **ANSI-C** 1989 (ANSI = American National Standards Institute): Dies ist das heute meistverwendete C. Neben der Standardisierung wurden verbesserte Prüfungsmöglichkeiten eingeführt.
- **C++** nach Ellis/Stroustrup: Objektorientierte Weiterentwicklung von C.

1.2 Die Turbo-C-Entwicklungsumgebung von (Borland)

In diesem Kurs werden C-Programme für DOS-Umgebungen erstellt. Die beliebteste DOS-Entwicklungsumgebung ist Turbo-C von der Firma Inprise(Borland).

Aufruf: `tc` oder `bc` [RETURN] vom aktuellen Verzeichnis aus (meist `c:\bc\bin`)

Es erscheint der sogenannte "Editor" (eigentlich: die **Integrierte Entwicklungsumgebung**, englisch **IDE** = "integrated developing environment"), in dem einfach drauf los geschrieben werden kann.

Die Erstellung eines C-Programms zerfällt in mehrere Schritte:

1. ***Editieren***: Eingabe des Programmtextes in einer höheren Programmiersprache mit dem eingebauten "Editor". (Ein Editor ist eine Art "Textverarbeitung für Programme".)

   Hinweis: Für UNIX-Programmierer gibt es oft keine integrierte Entwicklungsumgebung. UNIX-Entwickler beginnen im Editor vi, etwa mit dem Befehl

   ```
   vi -i prog1.c
   ```

   durch den der Editor im Einfügemodus gestartet wird.

   Beispiel: Editor der Turbo-C-Version 3.0

   Dann werden schrittweise Daten aus verschiedenen "Bibliotheken" hinzugefügt, wodurch der Programmcode immer länger wird.

2. ***Compilieren:*** Der Compiler wandelt den ASCII-Code in eine binäre, für die Maschine lesbare Form um. Ein solches Maschinenspracheprogramm hat die Endung `*.OBJ` (für Objekt-File).

3. ***Linken:*** Der Linker bindet div. "Bibliotheken" ein und trifft interne Entscheidungen für den Programmablauf. Er erzeugt ein startbares Programm des Typs `*.EXE`.

4. ***Aktivieren:*** Aufrufen des fertigen Programms.

Tastenkombinationen im Editor (Auszug)

[F1]	Hilfestellung aktivieren
[F2]	Im Editor befindlichen Quelltext speichern
[F3]	Öffnet das Dialogfenster
[F9]	Compiliert ein Programm via ***Compile/Make***
[F10]	aktiviert die Menüleiste
[Alt] [Buchstabe]	aktiviert Menüpunkte (die jeweils gültigen Buchstaben sind farbig in der Menüleiste hervorgehoben, z.B. [ALT] [F] öffnet das Menü ***File***)
[Alt] [F5]	Verlässt die C-Entwicklungsumgebung vorübergehend, um auf den DOS-Bildschirm umzuschalten
[Alt] [X]	Kehrt zur DOS-Ebene zurück und beendet die C-Entwicklungsumgebung
[Strg] [F9]	startet ein Programm (ggf. wird mit ***Compile/Make*** vorher compiliert)
[Einfg]	Schaltet Einfügemodus ein/aus
[Pos1]	geht zum Zeilenanfang
[Ende]	geht zum Zeilenende
[Strg] [Y]	ganze Zeile löschen

Blockoperationen

[Strg] [K] [B]	Blockanfang markieren
[Strg] [K] [K]	Blockende markieren
[Strg] [K] [V]	Block verschieben
[Strg] [K] [C]	Block kopieren
[Strg] [K] [Y]	Block löschen
[Strg] [K] [H]	Blockmarkierung verdecken/anzeigen
[Strg] [K] [P]	Block drucken
[Strg] [K] [I]	Block spaltenweise nach rechts
[Strg] [K] [U]	Block spaltenweise nach links

3 Lineare C-Programme

Unter einem **linearen Programm** versteht man ein Programm, welches nur aus den folgenden Teilen besteht:

- Eingabe
- Verarbeitung

- Ausgabe

Diese drei Teile werden linear – nacheinander – abgearbeitet.

Das traditionell erste Programm in jeder Programmiersprache ist ein "Hallo-Welt"-Programm. Es hat nur die Aufgabe, den Text "Hallo, Welt!" auf den Bildschirm auszugeben.

```
PROGRAMM 1: Hallo, Welt!
#include <stdio.h>
void main()
{                       /* Beginn des Anweisungsblocks von main */
  printf("Hallo, Welt!");
}                       /* Ende von main */
```

C-Programme bestehen im allgemeinen aus zwei Teilen:

1. Teil: Einbinden der benötigten Bibliotheken

Der C-Kern besteht aus ganz wenigen Befehlen. Nicht einmal Ausgabebefehle für den Bildschirm sind in diesem kleinen Kern enthalten. Daher muss man zusätzliche Dateien einbinden, in denen solche zusätzlichen Befehle gespeichert sind.

Dies geschieht mit der Anweisung

```
#include <Name der Bibliothek>  oder
#include "Name der Bibliothek"
```

Unterschied: Die spitzen Klammern werden verwendet, wenn die in den Klammern enthaltene Bibliotheksdatei im `\LIB`-Verzeichnis gesucht werden soll, die Anführungszeichen werden verwendet, wenn zusätzlich zum `\LIB`-Verzeichnis auch im aktuellen Verzeichnis gesucht werden soll.

Die häufigsten Ein und Ausgabebefehle sind enthalten in der Datei `STDIO.H` (Standard Input Output-Header). Eine Header-Datei enthält nur die Vereinbarungen für die Befehle (Funktionen), nicht aber die Funktionen selbst. Solche Vereinbarungen werden als **Prototypen** bezeichnet.

Befehle, die mit einem #-Zeichen (Nummern-Zeichen) beginnen, sind sogenannte **Präprozessor-Befehle**. Sie werden vor der eigentlichen Compilierung übersetzt.

2. Teil: Eigentliches Hauptprogramm

Jedes C-Programm besteht aus einer oder mehreren Funktionen, das sind in sich abgeschlossene Programmteile. Jedenfalls muss eine Funktion namens `main()` definiert werden, die als erstes abgearbeitet wird.

Das Wort `void` (engl. void = nichtig) bedeutet, dass diese Funktion keinen Rückgabewert hat, d.h. sie berechnet nichts, sondern "tut etwas".

Kommentare sind sehr wichtig für spätere Änderungen des Programms. Sie werden zwischen

```
/* Kommentar */
```

gesetzt. Manche C-Dialekte erlauben auch einzeilige Kommentare, die mit einem Doppel-Slash

```
// einzeiliger Kommentar
```

beginnen.

Wichtige Syntaxregeln

1. C unterscheidet Groß- und Kleinschreibung!

2. Jeder Befehl muss mit einem `;` (Strichpunkt, Semikolon) abgeschlossen werden!

Die Funktion `printf()` dient zur Bildschirmausgabe. Sie benötigt als Argument (in Klammern) eine **Zeichenkette (String)**. Eine Zeichenkette wird immer durch **Anführungszeichen** eingeschlossen und kann aus folgenden Elementen bestehen:

- Buchstaben
- Zahlen
- Escape-Sequenzen

Die Escape-Sequenzen stellen Steuerzeichen dar, mit denen etwa ein Piepston ausgelöst oder eine neue Zeile begonnen werden kann:

Escape Sequenzen

	Bedeutung	dezimal	hex
`\a`	Alarm (Piepston)	07	07
`\b`	Backspace (Rücklöschtaste)	08	08
`\t`	horizontaler Tabulator	09	09
`\r`	Carriage Return (Wagenrücklauf)	13	0D
`\n`	line (neue Zeile)	10	0A
`\`	ASCII-Zeichen mit dem hexadezimalen Code HH		
`\?`	?		
`\\`	\		
`\"`	"		
`\'`	'		

Die Bildschirmausgabe kann durch folgende Befehle noch professioneller gestaltet werden:

```
clrsrc();
```

Dieser Befehl ist in `conio.h` enthalten und löscht den Bildschirm.

```
getch();
```

Das Programm wartet, bis der Anwender auf eine beliebige Taste drückt.

Variablen

Wir wollen unser "Hallo-Welt"-Programm nun so abändern, dass eine Begrüßung auf dem Bildschirm ausgegeben wird. Jedoch soll die Begrüßung individuell verschieden sein, d.h. zuerst soll nach dem Namen gefragt werden, dann soll eine Eingabe möglich sein. Nach dieser Eingabe soll die Meldung "Hallo, Name!" auf dem Bildschirm ausgegeben werden.

Dies bedeutet aber, dass der PC sich den eingegebenen Namen "merken" muss.

Um eingegebene Werte für die Dauer des Programmablaufs zu speichern, bedient man sich des RAM-Speichers.

Unter einer Variablen versteht man einen **Speicherplatz im RAM**.

Die richtige Wahl des Variablentyps ist von entscheidender Bedeutung für die Geschwindigkeit und den Bedarf an Speicherplatz für ein Programm. Werden zum Beispiel nur ganze Zahlen (1, 3, 258, 13,...) verwendet, sollte man in jedem Fall nur Ganzzahlvariablen verwenden, da eine Fließkommavariable die Geschwindigkeit des Programms stark reduziert, vor allem dann, wenn Sie keinen Fließkommaprozessor besitzen.

Jede Variable muss vor ihrer Verwendung deklariert werden!

Deklaration von Variablen

Darunter versteht man die Angabe des Variablentyps und des Variablennamens in der Form

```
Variablentyp Variablenname;
```

Wie man hier sehen kann, bereitet die Variablendeklaration keine größeren Probleme.

Man gibt einfach den Variablentyp (etwa `int`) gefolgt von den Variablennamen (etwa `name`) ein.

Wenn man mehrere Variablen des gleichen Typs deklariert, so kann mann die Variablennamen durch Komma getrennt in einer Zeile schreiben. Die Deklaration wird durch ein Semikolon (Strichpunkt) abgeschlossen.

Der Variablenname kann sich aus beliebigen Buchstaben und Ziffern zusammensetzen, auch der Unterstrich "_" kann hierbei verwendet werden.

Wichtig für "Bezeichner" (identifier, Variablennamen)

- Ziffern dürfen nicht an erster Stelle stehen
- Deutsche Umlaute sind nicht gestattet
- C unterscheidet Klein- und Großschreibung d.h. `Steuer` und `steuer` sind nicht die gleichen Variablen.

Welche Variablentypen gibt es?

Wertebereiche von `signed`-Variablen (signed = "Zahlen mit Vorzeichen"):

Typ	Bedeutung	Bits	Wertebereich
`char`	"character" (Zeichen)	8	-128 bis + 127
`enum`	"enumerator" (Zähler)	16	0 bis + 655357
		16	-32768 bis + 32767
`int`	"integer" (Ganzzahl)	16	-32768 bis + 32767
`long`		32	-2147483648 bis + 2147483647
`float`		32	-3.4E37 bis + 3.4E38
`double`		64	1.7E-308 bis 1.7E308
`long double`		80	3.4E-4932 bis 1.1E+4932

Die angegebenen Wertebereiche können durch Angabe des Zusatzes `unsigned` verschoben werden:

Typ	Bits	Wertebereich
`unsigned int`	16	0 bis 65535
`unsigned long`	32	0 bis +4 Mrd.

Schließlich gibt es noch Variablentypen, die aus den genannten Grundtypen zusammengesetzt werden. Einen derartigen Variablentyp wollen wir auch jetzt kennen lernen:

```
PROGRAMM 2: Texteingaben
#include <stdio.h>
#include <conio.h>          /* wegen clrscr() */

void main()
{                           /* Beginn des Anweisungsblocks von main */
  char name[20];            /* Deklaration der Variable name */
  clrsrc();
  gets(name);               /* Eine Benutzereingabe wird erwartet */
  printf("\nHallo, %s!",name);  /* Ausgabe */
}                           /* Ende von main */
```

Mit `char name[20];` wurde eine Variable mit der Bezeichnung `name` definiert. Es handelt sich hier um eine **Zeichenkette (String)**, deren Länge in der eckigen Klammer angegeben werden muss. Eine Zeichenkette setzt sich aus vielen einzelnen char-Variablen zusammen. Wieso die Zeichenkette gerade so definiert wird, soll später erklärt werden. Der Befehl `gets()` ist ein Eingabebefehl speziell für String-Variablen.

Für die Eingabe von `int`-, `long`-, `float`-Variablen eignet sich der Befehl `scanf()` besser:

```
scanf("%d",&eingabe);
```

Der erste Parameter heißt Format-String. Er enthält eine Zeichenkette, die angibt, welcher Variablentyp gespeichert werden soll. Dabei bedeuten:

`int`	`%d`
`unsigned int`	`%u`
`unsigned int`	`%x` (hexadezimal)
`float`	`%f` `%g` `%e` `%E`
`char`	`%c`
`char*` (string, Text)	`%s` (nur bei printf!)
`long`	`%ld`
`unsigned long`	`%lu`
`double`	`%lf` `%lg` `%le`
`long double`	`%Lf` `%Lg` `%Le`
% Prozentzeichen	`%%`

Wichtig: Bei der Funktion `scanf()` muss vor dem Variablennamen das `&`-Zeichen stehen. Es bedeutet, dass an dieser Stelle nicht die Variable selbst genommen wird, sondern ihre **Speicheradresse im RAM**. **Ausnahme**: Bei **String-Variablen** steht **kein** `&`!

```
PROGRAMM 3: Steuerberechnung
#include <stdio.h>
#include <conio.h>
void main()             /* Beginn des Hauptprogramms */
{
  float netto,;
  clrscr();             /* Bildschirm löschen */
  printf("\n\t\t Steuerberechnung");
  printf("\n\n Bitte geben Sie den Nettobetrag ein");
  scanf("%f",&netto);   /* %f Speicheradresse für Eingabe netto */
  steuer = 0.2 * netto; /* Formel zum Berechnen von 20 % MWSt */
  printf("\n\n 20 %% MWSt von %.2f sind %.2f",netto,steuer);
                        /* %.2f bedeutet mit 2 Kommastellen */
  printf("Ende:Taste drücken!");
  getch();
}
```

Deklaration und Ansprache von Variablen

```
PROGRAMM 4a: Einfache Berechnung
#include <stdio.h>          /* Einbinden der Standard - Bibliothek */
int lohn,steuer,endbetrag;  /* Deklaration der Variablen */
void main (void)
{
  lohn = 25000;              /* 1. Zuweisung */
  steuer = 7800;             /* 2. Zuweisung */
  endbetrag = lohn - steuer; /* 3. Zuweisung */
  printf("Ausgezahlter Betrag: %d  S.-",endbetrag);
}
```

Variablen können aber auch **initalisiert** werden, das heißt, ihnen wird bereits bei der Deklaration ein Wert zugewiesen). Bei dem folgenden Beispielprogramm werden die Zuweisung 1 und 2 durch Initialisierung ersetzt.

Initialisierung von Variablen

```
PROGRAMM 4b: Einfache Berechnung mit Initialisierung
#include <stdio.h>    /* Einbinden der Standard - Bibliothek */
int lohn = 25000;     /* Deklaration  */
int steuer = 7800;    /* und Initialisierung */
int endbetrag;        /* Deklaration  */
void main(void)
{
  endbetrag = lohn - steuer; /* 3. Zuweisung */
  printf(" Ausgezahlter Betrag: %d  S.- ",endbetrag);
}
```

Wie man hier sieht, **verkürzt** diese Vorgangsweise die Länge des erzeugten Programms, da die Initialisierungswerte bei der Übersetzung direkt in das Datensegment des Programms geschrieben werden und die Zuweisungen 1 und 2 entfallen können.

GANZZAHLVARIABLEN

Ihnen können nur ganze Zahlen zugewiesen werden. Dazu gehören die Variablentypen `char`, `short`, `int` und `long`. Auch die Character-Variablen werden rechnerintern als Zahlen behandelt.

Folgende Zuweisungen sind also absolut identisch :

```
char zeichen;
zeichen = 'A'; /* Version 1 */
zeichen = 65;  /* Version 2 */
```

Um das Programm für andere lesbarer zu machen, ist es besser, Version 1 vorzuziehen, wenn Sie den Buchstaben A meinen.

Zusammenfassung Eingabebefehle

```
zeichen = getch();
```

Die Eingabe eines Zeichens wird abgewartet, das Zeichen wird in der Variablen `zeichen` abgelegt.

```
scanf("%d", &i);
```

Die Eingabe einer Integer-Zahl wird abgewartet (Abschluss der Eingabe mit der Enter-Taste), die Integer-Zahl wird in der Variablen `i` abgelegt.

```
gets(text);
```

Die Eingabe einer Zeichenkette (String) wird erwartet, der String wird in der Variablen `text` abgelegt.

Definition von Konstanten

1. Möglichkeit: Im Programm

```
const float PI = 3.14159;
const int MWSTSATZ = 10;
```

2. Möglichkeit: mit dem Präprozessor-Befehl `#define`:

```
#define PI 3.14159
#define MWSTSATZ 10
```

```
PROGRAMM 5: Kreisflächenberechnung
#include<stdio.h>
#include<conio.h>
#define PI 3.14159

void main()          /* Beginn des Hauptprogramms */
{
  float r,flaeche;
  clrscr();          /* Bildschirm löschen */
  printf("\n\t\t Kreisflächenberechnung");
  printf("\n\n bitte geben Sie den Radius ein!");
  scanf("%f",&r);    /* %f Speicheradresse für float,Eingabe r */
  flaeche=r*r*PI;    /* Formel zum Berechnen der Kreisfläche */
  printf("\n\n Kreisfläche:%f.2f m2",flaeche);
                     /* %.2f bedeutet mit 2 Kommastellen */
  printf("Ende:Taste drücken!");
  getch();
}
```

Formatierte Ausgabe

```
   12.34000   %.5f    Ausgabe mit 5 Kommastellen
  --12.34000  %10.5f  Ausgabelänge minimal10 Zeichen
00012.34000   %010.5f Auffüllen mit Nullen
  +12.34000   %+10.5f positive Zahlen mit "+"-Zeichen
   12.34000   %-10.5f Ausgabe immer linksbündig (minus)
              %-+010.5f
```

Ausdrücke und einfache Operationen

Der Zuweisungsoperator =

```
summe = a+b;   /* Ausdruck */
```

Der Wert des Ausdrucks auf der rechten Seite wird in den Speicherplatz der Variablen auf der linken Seite geschrieben, zum Beispiel:

```
summe = a+b;  /* richtig */
a+b=summe;    /* falsch */
```

Mathematische Operatoren

- Addition
- Subtraktion
- Multiplikation
- / Division (siehe unten!)
- % "Restoperator" (Modulo-Operator)

Beispiele: 7/2=3, 1 Rest

Das Ergebnis von 7 % 2 ist daher 1 (nur der Rest wird geschrieben).

- Integer Division: 7 / 2 = 3
- float Division: 7.0 / 2.0 = 3.5 (unbedingt Komma-Null angeben!)

Zusammengesetzte Zuweisungsoperatoren

Beispiel

```
i=i+4;
```
Wert von i wird um 4 erhöht

```
i+=4;
```
Gleichbedeutende Kurzschreibweise

Beispiel

```
i=i+1;
```
Wert von i wird um 1 erhöht

```
i+=1;
```
Gleichbedeutende Kurzschreibweise

```
i++;
```
noch kürzer

Der Operator ++ wird als **Inkrementierungsoperator** bezeichnet (inkrementieren heißt: "um 1 erhöhen".)

Beispiel

```
i=i-1;
```
Wert von i wird um 1 erniedrigt

```
i-=1;
```
Gleichbedeutende Kurzschreibweise

```
i--;
```
noch kürzer

Der Operator — wird als **Dekrementierungsoperator** bezeichnet (dekrementieren heißt: "um 1 erniedrigen".)

Implizite und explizite Typenumwandlung

Manchmal treffen Variablen mehrerer Datentypen zusammen. Hier tritt eine von C gesteuerte Typenumwandlung auf, und zwar in folgender Reihenfolge:

```
nachrangig                          vorrangig
int < unsigned int  <  long <  double
char                           float
2 * 3.0 = 6.0
int float float!
```

Man kann diese Typenumwandlung auch erzwingen (**explizite** Typenumwandlung). Dies geschieht durch Voranstellen des gewünschten Datentyps in Klammer, etwa:

```
int i;
(double)i;    /* int-Variable i wird in double-Variable umgewandelt */
```

Beim Arbeiten mit Konstanten kann diese Umwandlung auch durch **Anhängen eines Typenkennzeichens** erzwungen werden:

```
349L    Anhängen eines L          long int
245U    Anhängen eines U          unsigned int
3F      Anhängen eines F          float
046     Voranstellen einer 0      int-Konstante 38 in
                                  Oktaldarstellung
```


Christian Zahler christian.zahler@telecom.at

0xa3	Voranstellen von 0x	int-Konstante 163 in Hexadezimal darstellung

3 Verzweigungen

4.1 Einfache Verzweigung:

Verzweigungen bieten mehrere Möglichkeiten des Programmablaufes aufgrund von Ja/Nein-Entscheidungen. Wichtig ist die Bedingung, die nach dem Schlüsselwort `if` in einer Klammer steht.

Syntax

```
if (Bedingung)
{
  /* JA-Block */
}
else
{
  /* NEIN-Block */
}
```

Beispiel

```
void main()
{
  if (r>0)
  {                    /* Wenn r>0, wird dieser Block ausgeführt */
    flaeche =r*r*PI;
    printf("Fläche:%.2f";flaeche);
  }
  else
  {                    /* Wenn r<=0, wird dieser Block ausgeführt */
    printf("\n\n Ungültige Eingabe!");
  } /* Ende  */
  printf("\n Taste drücken!");
  getch();
}
```

Bedingungen haben auch selbst einen Wert:

true bzw. **ungleich 0**
false bzw. **0**

Bedingungen sind oft das Ergebnis von Vergleichen:

Vergleichsoperatoren

```
if (a>b)  {...} // größer
if (a<b)  {...} // kleiner
if (a==b) {...} // gleich; nicht verwechseln mit Zuweisungsoperator =!
if (a!=b) {...} // ungleich
if (a>=b) {...} // größer oder gleich
if (a<=b) {...} // kleiner oder gleich
```

Überlegen Sie: Was bedeutet

```
a==b=0    ?
```

Ausdrücke, die nur wahr (true, <>0) und falsch (false, ==0) als Ergebnis haben können, heißen BOOLEsche Ausdrücke.

Verknüpfungen BOOLEscher Ausdrücke (Logische Operationen)

Solche Ausdrücke können auch miteinander verknüpft werden:

Logisches "UND": Symbol &&

Ausdruck A	Ausdruck B	A && B
w	w	w
w	f	f
f	w	f
f	f	f

Logisches "ODER": Symbol ||

Außer dem logischen "UND" gibt es noch das logische "ODER", auch **Disjunktion** genannt.

Das "logische oder" entspricht dem sprachlichen "entweder ... oder ... oder beide".

"Ich möchte etwas trinken ODER etwas essen" bedeutet, dass ich etwas trinken ODER etwas essen ODER beides möchte.

Das Symbol für "ODER" lautet in C ||.

Die Wahrheitswertetabelle für "ODER":

Ausdruck A	Ausdruck B	A || B
w	w	w
w	f	w
f	w	w
f	f	f

Die Aussage "A oder B" ist also nur dann falsch, wenn beide Aussagen falsch sind.

Logisches "NICHT": Symbol !

Schließlich gibt es auch noch die Negation (das logische nicht), die "Verneinung" einer Aussage mit dem Symbol !.

Ausdruck A	!A
w	f
f	w

3.2 Mehrfache Verzweigung


```
PROGRAMM 4: Wie viele Tage hat ein Monat?
#include <stdio.h>
#include <conio.h>

void main()
{
  int monat;
  printf("\n  Nummer des Monats eingeben:");
  scanf("%D",&monat);
  switch (monat)
  {
  case 1:
  case 3:
  case 5:
  case 7:
  case 8:
  case 10:
  case 12:
    printf("Der Monat hat 31 Tage.");
    break;
  case 2:
    printf("Der Monat hat 28/29 Tage");
    break:
  case 4:
  case 6:
  case 9:
  case 11:
    printf("Der Monat hat 30 Tage.");
    break;
  default: printf("falsche Eingabe!");
  } /* Ende  */
} /* Ende  */
```

4 Schleifen = Wiederholungsstrukturen (Iterationen)

4.1 Kopfgesteuerte Schleife (while-Schleife)

Eine Schleife dient dazu, bestimmte Programmteile so lange abzuarbeiten, bis eine **Schleifenbedingung** den Wert **false** annimmt. Schleifenbedingungen müssen **BOOLEsche Ausdrücke** sein!

Kopfgesteuerte Schleifen haben die Schleifenbedingung am Beginn:

Beispiel

```
int i=0;
while (i<5)
{
  printf("%d",i);
  i++;
}
```

Es ist möglich, dass eine solche Schleife **gar nicht** abgearbeitet wird. (Wann?)

4.2 Schwanzgesteuerte Schleifen (do-while-Schleife)

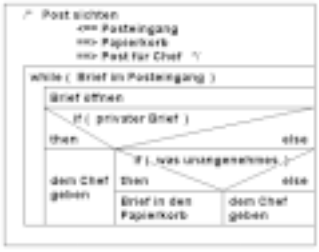
Schwanzgesteuerte Schleifen haben die Schleifenbedingung am Ende:

Beispiel

```
int i=0;

do
{
  printf("%d",i);
  i++
}
while (i<5);        //Schleifenbedingung am Ende
```

Diese Schleife muss mindestens einmal abgearbeitet werden!



4.3 Schleife mit bekannter Durchlaufzahl (Zählschleife, for-Schleife)

Beispiel

```
int i=0;
for (i=0;i<5;i++)
{
  printf("%d",i);  // entspricht einer while Schleife (kopfgesteuert)
}
```

Beispiel

```
do
{
  printf("Geben Sie Ihr Alter ein:");
  printf("%d";&alter);
  if (alter<0)
    printf("Ungültige Engabe: Eingabe wiederholen!");
}
while (alter<0);
```

Oft möchte man den Benutzer zwingen, richtige Eingaben vorzunehmen. Eine gute Möglichkeit besteht darin, die Aufforderung zur Eingabe so lange zu wiederholen, bis eine korrekte Eingabe erfolgt ist:

```
char antw;
do
{........
  printf("Nochmalige Eingabe (j/n)?");
  antw=getche(); /* getche() bedeutet: wartet auf Tastendruck
                    mit Echo, d.h. Taste ist am Bildschirm sichtbar */
}
while (antw!='n');
```

Zur Unterscheidung der Schleifenarten

	For-Schleife	Do-While-Schleife	While-Schleife
Abbruchbedingung	Durchlaufzahl	am Ende	am Anfang
Mindestdurchläufe	Anzahl vorgegeben	1	0
Maximale Durchlaufzahl	vorgegeben	"unbegrenzt"	"unbegrenzt"

5 Bitoperatoren

C ist eine hardwarenahe Programmiersprache. Manchmal ist es daher von Vorteil, nicht zeichenweise (byte-weise) zu arbeiten, sondern direkt Einfluss auf das Bitmuster zu nehmen (bitweises Arbeiten) – zum Beispiel, weil so schnellere Routinen möglich sind! Dafür stehen einige leistungsfähige Rechen- und Vergleichsoperatoren zur Verfügung:

(Schiebeoperatoren): >> und <<

Diese Operatoren schieben (shiften) das Bitmuster um eine oder mehrere Stellen nach links (<<) oder rechts (>>).

Beispiel

```
26 << 1 == 52
Bitmuster von 26:      0000 0000 0001 1010
<<1                    0000 0000 0011 0100  (entspricht 52)
```

Das Verschieben des Bitmusters um eine Stelle nach links entspricht einer schnellen Multiplikation mit 2. Rechts freiwerdende Stellen werden mit 0 aufgefüllt.

Achtung

Das höchstwertige Bit ist das Vorzeichen-Bit. Wird in diese Stelle eine 1 geschoben, so kann aus einem positiven Wert ein negativer Wert werden!

Der Operator >> verschiebt das Bitmuster um eine oder mehrere Stellen nach rechts. Links freiwerdende Stellen werden mit dem Vorzeichenbit aufgefüllt. Das Verschieben um eine Stelle nach rechts entspricht einer schnellen Division durch 2.

Beispiel

```
(-8) >> 2 == -2
Bitmuster von –8:      1111 1111 1111 1000
>>2                    1111 1111 1111 1110 (entspricht –2)
```

Darstellung negativer Zahlen

Man verwendet für die Darstellung negativer Zahlen die "Zweierkomplement-Darstellung":

Beispiel

```
–8
Bitmuster von 8:              0000 0000 0000 1000
NOT 8 (Einerkomplement)       1111 1111 1111 0111
+ 1                           0000 0000 0000 0001
–8 (Zweierkomplement)         1111 1111 1111 1000
```

Der ODER-Operator |

Die Verknüpfung erfolgt so wie beim ||-Operator, aber bitweise.

a	b	a | b
1	1	1
1	0	1
0	1	1
0	0	0

Beispiel

```
a = 0x55;      /*  0101 0101  */
b = 0xa7;      /*  1010 0111  */
```

```
c = a | b;     /*  1111 0111  == 0xf7 */
```

Der bitweise ODER-Operator kann verwendet werden, um in einem Byte ein bestimmtes Bit auf 1 zu setzen.

Beispiel

In der Variablen `a` (Typ `unsigned char`) ist das 4. Bit auf 1 zu setzen, die anderen Bits sollen unverändert bleiben.

Lösung

```
a = a | 0x08;   /* 0x08 binär: 0000 1000 */
```

Der bitweise UND-Operator &

Die Verknüpfung erfolgt so wie beim `&&`-Operator, aber bitweise.

a	b	a & b
1	1	1
1	0	0
0	1	0
0	0	0

Beispiel

```
a = 0x55;   / *  0101 0101  */
b = 0xa7;   /*  1010 0111  */
c = a | b;  /*  0000 0101  == 0x05 */
```

Der bitweise UND-Operator kann verwendet werden, um in einem Byte ein bestimmtes Bit auf 0 zu setzen.

Beispiel

In der Variablen `a` (Typ `unsigned char`) ist das 4. Bit auf 0 zu setzen, die anderen Bits sollen unverändert bleiben.

Lösung

```
a = a & 0x07;   /* 0x08 binär: 1111 0111 */
```

Der bitweise XOR-Operator ^

Hier erfolgt die Verknüpfung der beiden Bits über ein EXKLUSIVES (ausschließendes) ODER:

a	b	a ^ b
1	1	0
1	0	1
0	1	1
0	0	0

Beispiel

```
a = 0x55;   /*  0101 0101  */
b = 0xa7;   /*  1010 0111  */
c = a ^ b;  /*  1111 0010  == 0xf2 */
```

Der Einerkomplement-Operator ~

Invertiert jedes Bit.

a	~a
1	0
0	1

Beispiel

```
a = 0x55;  /*  0101 0101  */
a = ~a;    /*  1010 1010 == 0xaa */
```

Wie vorhin erwähnt, gilt für vorzeichenbehaftete ganzzahlige Datentypen:

```
~a + 1 = -a;
```

6 Bildschirmgestaltung und Grafik mit Borland-C

6.1 Bildschirmsteuerung

Ein normaler DOS-Textbildschirm besteht aus 80 Spalten und 25 Zeilen. Durch einige Anweisungen kann man die Professionalität der Bildschirmausgabe deutlich erhöhen:

Die Anweisung

```
goto(x,y);
```

setzt den Cursor auf die Spalte x und die Zeile y. Damit kann man sich oft die Angabe \n für eine neue Zeile sparen, indem man den Cursor direkt auf die gewünschte Position plaziert.

Diese Funktion kann auch zur Zentrierung von Texten auf dem Bildschirm verwendet werden. Beispiel:

```
char text[] = "Überschrift";
gotoxy((int)(40-length(text)/2),y);
printf(text);
```

Farbige Textdarstellung:

```
textcolor(RED);
cprintf("Text");
```

Hier wird das Wort "Text" in roter Farbe geschrieben. Wichtig: Die Farbänderung muss vor der Bildschirmausgabe erfolgen; außerdem muss statt `printf()` der Befehl `cprintf()` verwendet werden (`c` für `color`).

Das in Großbuchstaben geschriebene Wort `RED` stellt eine Farbkonstante dar; andere Konstanten lauten:

Konstante	Wert	Hintergrund?	Vordergrund?
BLACK	0	Ja	Ja
BLUE	1	Ja	Ja
GREEN	2	Ja	Ja
CYAN	3	Ja	Ja
RED	4	Ja	Ja
MAGENTA	5	Ja	Ja
BROWN	6	Ja	Ja
LIGHTGRAY	7	Ja	Ja
DARKGRAY	8	Nein	Ja
LIGHTBLUE	9	Nein	Ja
LIGHTGREEN	10	Nein	Ja
LIGHTCYAN	11	Nein	Ja
LIGHTRED	12	Nein	Ja
LIGHTMAGENTA	13	Nein	Ja
YELLOW	14	Nein	Ja
WHITE	15	Nein	Ja
BLINK	128	Nein	***

```
textcolor(GREEN+BLINK);  /* grünblinkender Text */
textbackground(LIGHTGREY);
/* Bildschirmhintergrund Lichtgrau-Text grünblinkend */
```

Änderung des kompletten Hintergrundes

```
textbackground(LIGHTGREY);
clrscr();
```

6.2 Grafik in BORLAND C

Gerade heute ist es wichtig, Programme auf grafischer Basis zu erstellen. Dafür gibt es in BORLAND-C (**wohlgemerkt: NUR dort!**) eine Reihe von leistungsfähigen Befehlen, die in kurzer Zeit sehr ansprechende Ausgaben auf dem Bildschirm ermöglichen.

Im Programm muss zunächst der Bildschirm auf Grafikmodus "umgeschaltet" werden. Je nach Art des Bildschirms ist die Auflösung und die Möglichkeit, Farben darzustellen, unterschiedlich. Daher ist es nötig, zu wissen, welche "Bildschirmkarte" im Rechner installiert ist. Diese Karte unterstützt die Ausgabe von Zeichen und auch Grafik auf dem Bildschirm. Ein Monochrom-Bildschirm wird meist mit einer so genannten Hercules-Karte (Auflösung: 720 x 348 Pixel) betrieben, bei Farbbildschirmen ist heute die VGA-Karte (Auflösung: 640 x 480 Pixel) Standard.

Das "Umschalten" in den Grafikmodus wird in C folgendermaßen erreicht:

```
PROGRAMM 5: Demo-Programm zur Grafik
#include <stdio.h>
#include <conio.h>
#include <graphics.h>  /* Hier sind die wichtigsten Grafikbefehle */
#include <stdlib.h>
void main()
{
```

```
    a,b,graphdriver,graphmode;
    int startx,starty,endx,endy,number;
    char f;
    detectgraph(&graphdriver, &graphmode);
    initgraph(&graphdriver,&graphmode,"..\\bgi");
    putpixel(200,100,LIGHTGREEN);  /* Farbkonstanten in conio */
    while (!kbhit());       /* Schleife ohne Schleifenkörper! */
         /* Die Funktion () ermittelt,
            ob eine Taste gedrückt wurde. */
    closegraph();
}
```

Der Umschaltebefehl in den Grafikmodus heißt `initgraph()`. Er benötigt 3 zusätzliche Angaben:

- Die Integer-Variable `graphdriver`, die angibt, welchen "Grafiktreiber" ich verwenden muss. Ein Grafiktreiber ist ein Programm, das abhängig von der verwendeten Grafikkarte die Darstellung von Grafik in C-Programmen ermöglicht. Die Variable `graphdriver` ist eine Integer-Variable, die folgende Werte annehmen kann:

Konstante	Wert	Bemerkungen
DETECT	0	automatische Erkennung
CGA	1	
MCGA	2	
EGA	3	
EGA64	4	
EGAMONO	5	
IBM8614	6	
HERCMONO	7	
ATT400	8	
VGA	9	
PC3270	10	

Weiß ich z.B., dass ich einen VGA-Bildschirm habe, so kann ich schreiben:

```
graphdriver=9; /* oder graphdriver = VGA */
```

Um Programme aber auf verschiedenen Computern mit verschiedenen Karten lauffähig zu machen, bedient man sich meist der Anweisung `detectgraph`, die selbst herausfindet, welche Grafikkarte vorhanden ist, also:

```
detectgraph(&graphdriver, &graphmode);
```

- Die Integer-Variable `graphmode`, die - bei gegebenem Grafiktreiber - noch gewisse Auswahlmöglichkeiten bei Auflösung und Farbe bietet. Bei größerer Auflösung sind meist weniger Farben möglich.

- Den **Pfad**, wo C die **Grafiktreiber** findet. Man erkennt diese Treiber an der Erweiterung "`*.BGI`" (Borland Graphics Interface).

```
HERC.BGI
EGAVGA.BGI
CGA.BGI     usw.
```

Man sollte diese Treiber immer im selben Verzeichnis wie C haben.

Am Ende einer Grafikprogrammierung ist es nötig, mit `closegraph()` wieder in den Textmodus zurückzuschalten.

Vergisst man diesen Befehl, so kann es zu überraschenden Aktionen beim nächsten Programmstart kommen!

Im Grafikmodus gibt es ein eigenes Koordinatensystem. Der Ursprung befindet sich links oben, er hat die Koordinaten (0,0).

Die Funktionen `getmaxx()` und `getmaxy()` liefern die jeweilige größte x- bzw. y-Koordinate. Für einen VGA-Bildschirm liefert

```
getmaxx: 639
getmaxy: 479
```

Der rechte untere Punkt hat also hier die Koordinaten (639,479).

Wichtig: Diese beiden Funktionen funktionieren nur dann, wenn vorher mit `initgraph` ein Grafikmodus gesetzt wurde!

Wichtige Grafik-Grundbefehle

`putpixel(x,y,color)`
zeichnet einen Punkt an die Stelle `x,y`, Farbe `color` entweder als Zahl oder englisch als Wort angeben, z. B.

```
putpixel(230,450,YELLOW)
```

`line(x1,y1,x2,y2)`
zeichnet eine Strecke von `x1,y1` nach `x2,y2`

`rectangle(x1,y1,x2,y2)`
zeichnet ein Rechteck; linker oberer Punkt `x1,y1`, rechter unterer Punkt `x2,y2`

`circle(x,y,radius)`
zeichnet Kreis mit Mittelpunkt `x,y` und Radius `r`

`ellipse(x,y,startwinkel, endwinkel,xradius,yradius)`
zeichnet Ellipse(nsegment) mit Mittelpunkt `x,y`, Hauptachse `xradius`, Nebenachse `yradius`, Begrenzung durch 2 Winkelangaben

`outtext(x,y,"text")`
schreibt einen Text, beginnend bei `x,y`, auf den Grafikschirm

Weitere Befehle: `arc`, `bar`, `bar3d`, `drawpoly`, `sector`, `pieslice`

Finden Sie selbst heraus, was diese Befehle bewirken!

Die Zeichenfarbe kann mit `setcolor(color)` gesetzt werden, die Hintergrundfarbe mit `setbkcolor(color)`.

Für die Einstellung der Schriftart und Größe der Texte gibt es den Befehl

```
sttextstyle(font, direction, charsize).
```

Werte für die Variablen

font

DEFAULT_FONT	0	8x8 Bitmusterzeichensatz
TRIPLEX_FONT	1	Vektorzeichensatz
SMALL_FONT	2	Vektorzeichensatz
SANS_SERIF_FONT	3	Vektorzeichensatz
GOTHIC_FONT	4	Vektorzeichensatz

Wichtig: Die Vektorzeichensätze sind in eigenen Dateien abgespeichert, die die Erweiterung `.CHR` (für character) haben. Mitgeliefert werden zB `litt.chr` (für little), `sans.chr` (für Sansserif), `trip.chr` (für TriplexFont), `goth.chr` (für GothicFont) usw.

direction

HORIZ_DIR	0	von links nach rechts
VERT_DIR	1	von unten nach oben

charsize

USER_CHAR_SIZE	0	benutzerdefinierte Zeichengröße
NORM_SIZE	1	normale Textgröße

Die Dicke der Linien kann mit folgendem Befehl geändert werden:

```
setlinestyle(linestyle, pattern, thickness)
```

SOLID_LN	0	durchgezogen
DOTTED_LN	1	gepunktet
CENTER_LN	2	Punkt-Strich-Punkt
DASHED_LN	3	strichliert
USER_BIT_LN	4	Benutzerdefiniert

pattern

EMPTY_FILL	0	füllt mit der Hintergrundfarbe
SOLID_FILL	1	füllt mit der Vordergrundfarbe
LINE_FILL	2	=========
LTSLASH_FILL	3	///////////
SLASH_FILL	4	/////////// mit dicken Linien
BKSLASH_FILL	5	\\\\\\\\\\\ mit dicken Linien
LTBKSLASH_FILL	6	\\\\\\\\\\\
HATCH_FILL	7	leicht schraffiert
XHATCH_FILL	8	stark schraffiert (überkreuzend)
INTERLEAVE_FILL	9	abwechselnde Linien
WIDE_DOT_FILL	10	weit auseinander liegende Punkte
CLOSE_DOT_FILL	11	eng beieinander liegende Punkte
USER_FILL	12	benutzerdefinierbares Füllmuster

thickness

NORM_WIDTH	1	normale Linienbreite
THICK_WIDTH	3	dicke Linien

Schließlich können geschlossene Flächen mit `floodfill(x,y,randfarbe)` angemalt werden, wobei die Füllfarbe und das Füllmuster zuerst mit `setfillstyle(pattern,color)` gesetzt werden muss. Die möglichen Werte für `color` wurden schon bei der Beschreibung der Funktion `textcolor` aufgelistet.

Oft ist es nötig, zwischen Grafik- und Textmodus "hin- und herzuschalten". Dazu gibt es folgende Befehle:

restorecrtmode()	schaltet auf Textmodus zurück
setgraphmode(graphMode)	schaltet auf Grafikmodus zurück, löscht Bildschirm
cleardevice()	bleibt im Grafikmodus, löscht Bildschirm

Grafikzeichenfenster: `setviewport`, `clearviewport`

7 Funktionen

Funktionen sind abgeschlossene Unterprogramme in C. Jedes C-Programm besteht aus einer Reihe von Funktionen. Die Funktion `main()` hat die besondere Eigenschaft, dass sie als erste abgearbeitet wird.

Man unterscheidet:

Funktionen ohne Rückgabewert

```
void funktion(Parameterliste)
{
  Anweisungen;
}
```

Funktionen mit Rückgabewert

```
void funktion(Parameterliste)
{
  Anweisungen;
  return Rückgabewert;
}
```

Diese Funktion würde einen `int`-Wert als Rückgabewert liefern.

Parameter: Sie werden benützt, um zwischen Funktionen Daten auszutauschen. Parameter können prinzipiell jeden Datentyp haben.

Beispiel

```
int quadrat(int a)
{
  return a*a;
}
```

Diese Funktion wird im aufrufenden Block folgendermaßen angesprochen:

```
void main()
{
  int x;
  long ergebnis;
  char eingabe[5];
  printf("Geben Sie eine ganze Zahl ein:");
  x = atoi(gets(eingabe));
  ergebnis = quadrat(x);
  printf("\n %d hoch 2 = %ld",x, ergebnis);
  getch();
}
```

PROTOTYPEN

Die Reihenfolge der Funktionen innerhalb eines C-Programms ist prinzipiell egal. Damit der Compiler aber absehen kann, welche Funktionen mit welchen Parametern noch deklariert werden, ist es sinnvoll, am Beginn des Programms eine Liste mit Prototypen anzugeben, z.B.

```
int quadrat(int);     // Prototyp der Funktion quadrat = Deklaration

void  main()
{ ... }

int quadrat(int a)    // Funktion selbst = Definition
{
  return a*a;
}
```

GLOBALE und LOKALE VARIABLEN

Globale Variablen erkennt werden außerhalb jeder Funktion deklariert. Auf globale Variablen kann aus sämtlichen Funktionen eines Programms aus sie zugegriffen werden.

Nun gibt es in C die Möglichkeit, Variablen lokal zu deklarieren. Sie können dann nur in der Funktion oder in dem Block benutzt werden, in dem sie deklariert wurden – außerhalb sind sie unbekannt.

Beispiel für den Gebrauch lokaler Variablen

```
#include <stdio.h>
void main(void)
{
   int lokale_variable;
     /* eine lokale Variable wird innerhalb der Funktion deklariert */
  lokale_variable = 10;
  printf("Inhalt von lokaler Variablen: %d",lokale_variable );
}
```

Nach dem Verlassen der Funktion verlieren lokale Variablen ihren Wert. Wurde die lokale Variable initialisiert, so wird ihr dieser Wert bei jedem erneuten Aufruf wieder zugewiesen.

Deklariert man in einer Funktion eine lokale Variable, die die gleiche Bezeichnung wie eine globale Variable trägt, wird immer auf die lokale Variable zugegriffen. Die globale Variable ist dann in dieser Funktion nicht mehr ansprechbar, wie man in folgenden Beispiel sehen kann:

```
#include <stdio.h>

int zahl = 30;

void main(void)
{
  int zahl;
  zahl = 10;
  printf(" Zahl = %d",zahl );
}
```

Die im Block deklarierte Variable `zahl` hat mit der globalen Variablen mit der gleichen Bezeichnung nichts zu tun. Als Ausgabe erhält man also 10 .

VARIABLENZUSÄTZE

const

Variablen, die den Zusatz `const` erhalten, sind keine Variablen mehr, sondern Konstanten. Diese müssen gleich bei der Deklaration initialisiert werden. (Der Wert von Konstanten darf ja nicht mehr verändert werden!)

```
const float PI = 3.1415926;
```

volatile

Durch den Zusatz `volatile` teilt man dem Compiler mit, dass sich der Inhalt einer Variablen auch ohne Zuweisung ändern kann. Wenn z.B. in der Variablen die momentane Zeit enthält und automatisch aktualisiert wird, oder wenn mit einem Zeiger auf sie zugegriffen wird.

```
volatile int zeit_in_sekunden;
```

register

Wenn man eine Variable besonders häufig benutzt, kann sie zwecks Geschwindigkeitssteigerung direkt in einem Prozessorregister abgelegt werden.

Borland-C erlaubt bis zu zwei Registervariablen (diese müssen vom Typ `short` oder `int` sein). Deklariert man mehr als zwei Registervariablen, so werden die überzähligen Variablen wie normale Variablen behandelt.

Registervariablen dürfen nur lokale Variablen sein!

```
register int zaehler;
```

extern

Borland-C erlaubt die Aufteilung eines Programms in mehrere Module, die getrennt übersetzt werden können.Wird nun in einem Modul eine Variable deklariert, auf die man aus einem anderen Modul heraus zugreifen will, taucht ein Problem auf:

Da sie in diesem Modul nicht deklariert wurde, kann der Compiler mit ihr nichts anfangen. `extern` teilt ihm nun den Typ dieser Variable mit, reserviert aber keinen Speicherplatz für sie, denn dann wäre ja zweimal Platz für ein und dieselbe Variable reserviert.

```
extern int woanders;
```

static

Lokale Variablen sind von Natur aus dynamisch: wird die Funktion verlassen, verliert die Variable umgehend ihren Wert. Um diese manchmal unerwünschte Eigenschaft zu umgehen, existiert der Variablenzusatz `static`. `static`-Variablen behalten ihren Wert auch nach Verlassen der Funktion. Trotzdem bleiben sie aber lokale Variablen, auf die man nach Verlassen der Funktion nicht mehr zugreifen kann. Initialisiert man eine `static`-Variable, so erfolgt die Wertzuweisung nur beim ersten Aufruf der Funktion, bei jedem weiterem Aufruf bleibt der Wert, den die Variable beim Verlassen der Funktion hatte, erhalten.

```
#include <stdio.h>
void main(void)
{
  static int zahl = 10;
  /* Initialisieren der static-Variablen */
  printf("Inhalt von Zahl : %d",zahl );
  getchar();
}
```

REKURSIONEN

Als Rekursionen werden Funktionen bezeichnet, die in ihrem Code einen Aufruf von sich selbst enthalten.

Wichtig:

- Kommt es zu einem solchen Aufruf, so wird nicht dieselbe Funktion aufgerufen, sondern eine neue Instanz dieser Funktion im Speicher erzeugt!
- Rekursionen benötigen unbedingt eine **Abbruchbedingung**!

```
PROGRAMM 6: Rekursion - 2^x
#include <stdio.h>
#include <conio.h>

long zweihoch(int zahl);

void main()    /* Rekursives Programm 2^x */
{
  int zahl=0;
  clrscr();
  printf("Bitte geben Sie eine Zahl ein: ");
  scanf("%d", &zahl);
  printf("Zahl: %d", zweihoch(zahl));
  getch();
}

long zweihoch(int zahl)
{
  if (zahl>=1)
    return 2*zweihoch(zahl-1);
  else
    return 1;
}
```

```
PROGRAMM 7: Rekursion - Sierpinski-Kurve
#include <stdio.h>
#include <conio.h>
#include <graphics.h>
#include <stdlib.h>
#include <time.h>

void main()
{
  int a,b,graphdriver,graphmode;
  int startx,starty,endx,endy,number;
  char f;
  detectgraph (&graphdriver, &graphmode);
  initgraph(&graphdriver,&graphmode,"..\\bgi");
  randomize();     /* in stdlib.h */
  startx = random(getmaxx());
  starty = random(getmaxy());
  do {
    number = random(3)+1;
    if (number==1) {
      a = (int)(0.5*getmaxx());
      b = (int)(0.1*getmaxy());
    }
    if (number==2) {
      a = (int)(0.2*getmaxx());
      b = (int)(0.9*getmaxy());
    }
    if (number==3) {
      a = (int)(0.8*getmaxx());
      b = (int)(0.9*getmaxy());
    }
    endx = (int)(0.5*(startx+a));
    endy = (int)(0.5*(starty+b));
    putpixel(endx,endy,LIGHTGREEN);  /* Farbkonstanten in conio */
    startx = endx;
    starty = endy;
  }
  while (!kbhit());
  closegraph();
}
```

8 Zusammengesetzte Datentypen

8.1 Felder (Arrays) und Strings

EINDIMENSIONALE FELDER

Felder sind eine Ansammlung von Variablen gleichen Typs, die (logisch, nicht wertmäßig) ähnlichen Inhalt und denselben Namen besitzen. Die einzelnen Elemente eines Feldes werden dann über einen in eckigen Klammern `[ ]` eingeschlossenen Index angesprochen.

Beispielprogramm

```
#include <stdio.h>
double umsatz[12]; /* Felddeklaration */
void main (void)
{
  umsatz[5] = 120000.30; /* Zuweisung 1 */
  printf(" Umsatz Juni : %f  S.-", umsatz[5] );
}
```

In diesem Beispiel wurde für die Umsätze ein Feld von zwölf (für jeden Monat eine) `double`-Variable deklariert. Durch die `/* Zuweisung 1 */` wird der Variablen `umsatz[5]` der Umsatz des Monats Juni !! zugewiesen. Das liegt daran, dass in der Programmiersprache TURBO-C die Felder immer mit Index 0 beginnen und somit `umsatz[5]` die sechste Variable in diesem Feld ist.

Es stehen also die Variablen umsatz[0] bis `umsatz[11]` zur Verfügung.

Achtung bei Umgang mit Feldern! Der Compiler prüft nicht, ob der angegebene Index auch zulässig ist. Wenn Sie mit der Variablen `umsatz[12]` auf den Umsatz des 13. Monats zugreifen wollen, so wird das den Compiler nicht weiter stören, das Programm wird aber beim Ausführen mit einiger Sicherheit darauf reagieren. Auch Felder können initialisiert werden. Dabei muss man die initialisierten Werte durch geschwungene Klammern einschließen und die einzelnen Werte durch Kommas trennen. z.B.

```
int zahlen[5] = {1,3,5,7,9};
```

Die Anzahl der Initialisierungswerte muss mit der Zahl der Feldelemente genau übereinstimmen.

Strings: Eine wichtige Sonderstellung nehmen Strings (Zeichenketten) ein. Sie werden mit

```
char text[10];          /* nur Deklaration */
char b[10] = "Hans";    /* Initialisierung */
```

Strings werden automatisch mit einer ASCII-0 abgeschlossen.

MEHRDIMENSIONALE FELDER

TURBO-C ist nicht nur auf eindimensionale Felder beschränkt. Auch mehrdimensionale Felder sind sinnvoll einsetzbar: Bei der Erfassung der monatlichen Umsätze in zwei Filialen wird man ein Feld mit zwei Indizes definieren:

```
double umsatz[2][12];
```

Der erste Index steht für die Filiale, der zweite für den Monat. Mit `umsatz[1][5]` hätte man so Zugriff auf den Juni-Umsatz der zweiten Filiale.

Man kann auch weitere Dimensionen einführen, so etwa eine Filiale noch in einzelne (etwa: 4) Abteilungen unterteilen, deren Umsatz getrennt ausgewiesen werden soll. Das sieht dann folgendermaßen aus:

```
double umsatz[2][4][12];
```

Wenn man nun den Umsatz der dritten Abteilung der zweiten Filiale im Monat Januar erfahren will, muss man `umsatz[1][2][0]` aufrufen.

Die Initialisierung mehrdimensionaler Felder erfordert schon etwas mehr Aufwand und Überlegung. z.B.

```
int zahlen[2] [5] = { {23,5,66,10,20},{ 20,9,66,10,44} };
```

Der rechte Index wächst dabei am schnellsten,das heißt

```
zahlen [0][0] == 23
zahlen [0][1] == 5
zahlen [1][4] == 44
```

Man kann auch mehrdimensionale Zeichenketten initialisieren. Die Anwendung zeigt das folgende Beispiel:

```
#include <stdio.h>
char jahreszeiten[4][9] = {"Frühling", "Sommer", "Herbst", "Winter" };
void main(void)
{
  puts(jahreszeiten[0]);
  puts(jahreszeiten[1]);
  puts(jahreszeiten[2]);
  puts(jahreszeiten[3]);
}
```

Der erste Index gibt die Jahreszeit an (vier an der Zahl),der zweite entspricht der Länge des längsten Namens ("Frühling" == 8) plus eins für die Endkennung.

8.2 Strukturen

DEFINITION NEUER VARIABLENTYPEN

typedef

Steht einem der Sinn nach individueller Programmierung, dann kann man `typedef` verwenden.Damit kann man neue Variablentypen neue Namen geben oder aber auch neue Typen selbst definieren. Soll z.B. der Umsatz in einer `float`-Variablen gespeichert werden, kann statt des normalen

```
float umsatz_1, oktober;
```

folgendes geschrieben werden:

```
typedef float umsatz;
umsatz umsatz_1,oktober;
```

Man kann auf den ersten Blick erkennen, dass es sich bei der deklarierten Variablen um einen Umsatz handelt. Diese Methode hat aber noch einen Vorteil: Sollten die Umsätze so steigen, dass man mit einer `float`-Variablen nicht mehr auskommt, kann man den Variablentyp einfach durch Änderung des Typs den neuen Erfordernissen anpassen:

```
typedef double umsatz;
```

Dadurch werden alle Variablen dieses Typs automatisch zu `double`-Variablen.

Die eigentliche Bedeutung von `typedef` ist, völlig neue Typen im Zusammenhang mit `struct` oder `union` konstruieren zu können. So würden zum Beispiel zwei komplexe Zahlen ohne `typedef` definiert:

```
struct complex
{
  double real;
  double imaginaer;
};
struct complex Komplexe_Zahl1;
struct complex Komplexe_Zahl2;
```

und so mit `typedef`:

```
struct struct
{
  double real;
  double imaginaer;
} complex;
complex Komplexe_Zahl1;
complex Komplexe_Zahl2;
```

Wie man hier sieht, kann man sich auf Dauer eine Menge Schreibarbeit einsparen. Auch ist man hier mit `typedef` flexibler, als wenn man einfach nur `struct` verwendet hätte. Auch wird das Programm leichter lesbar. Daher gehört `typedef` zum Stil moderner Programmierung und sollte so oft verwendet werden, wie es sinnvoll ist.

9.3 Unions

Mit einer `union` können Daten verschiedenen Formats überlagern werden.

Beispiel

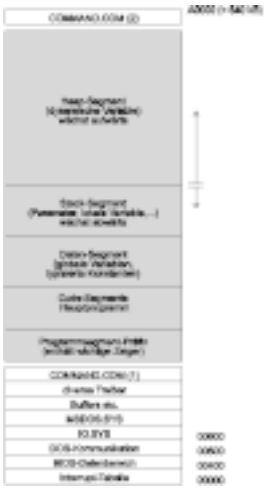
```
union x {
  float f;
  double d;
  unsigned char array[20];
} bsp;
```

Die Variablen `f`, `d` und `array` der Union belegen denselben Speicherplatz. Der Speicherplatz, den die *union bsp* insgesamt beansprucht, richtet sich nach der größten Variablen, hier nach `array`.

9 Pointer

Dazu muss man etwas über die Organisation des Arbeitsspeichers in C wissen. C unterteilt den Arbeitsspeicher (640 kByte, da DOS auf dem direkten Weg nicht mehr ansprechen kann!) in einzelne Segmente:

- **Code-Segmente**: dort sind die Funktionen abgespeichert
- **Daten-Segment**: dort finden sich globale Variablen und Konstanten
- **Stack-Segment**: wächst abwärts. Dort sind Parameter und lokale Variablen zu finden; ganz wichtig bei jedem Funktions- und Prozeduraufruf; bei der Rückkehr ins Hauptprogramm werden diese Informationen wieder entfernt. Da aus der Grafik ersichtlich ist, dass der Stack irgendwann einmal mit dem Datensegment zusammenstößt, ergibt sich, dass der Stack eine begrenzte Größe (64 kB) aufweist!



- **Heap-Segment:** wächst aufwärts. Hier besteht keine Gefahr einer "Kollision"; d.h. für die dort abgespeicherten dynamischen Variablen (werden wir später besprechen) gibt es – bis 640 kB – keine Größenbegrenzung!

Beispielsweise braucht man den Heap-Speicher auch für den Grafiktreiber (z.B. `EGAVGA.BGI`). Bei Aufruf der Prozedur `initgraph()` wird sofort der entsprechende Platz für den Treiber im Heap reserviert. Dieser Platz wird bei Aufruf von `closegraph()` wieder freigegeben.

Bisher haben wir immer Variablen verwendet, die einen Namen hatten und die einen gewissen Speicherplatz benötigten. C kann in der Standardeinstellung für ein Programm nur einen Speicherbereich von 64 kByte (= 1 Segment) reservieren, für alle Variablen, die das Programm benützt, stehen insgesamt jedoch auch nur 64 kByte zur Verfügung. Oft ist dies für ein rasches Arbeiten zu wenig.

Man bräuchte Variablen, die - bei Bedarf - den ganzen zur Verfügung stehenden RAM-Speicher belegen könnten, also theoretisch bis 640 kByte groß sein könnten. Günstig wäre auch, dass man die Größe nicht festlegen muss, sondern dass diese sich erst bei der Ausführung des Programms ergibt.

Solche Variablen gibt es: Sie heißen **dynamische Variablen**, da ihre Größe eben nicht von vornherein festgelegt ist. Sie haben KEINEN Namen!

Frage

Wie findet das Programm nun die Informationen, die in dieser Variablen abgespeichert sind, wenn die Variable keinen Namen hat?

Antwort

Im Programm treten **Zeigervariablen** (engl. **POINTER**) auf, die auf den (namenlosen) Speicherbereich zeigen.

Beispiel

```
int *p;
```

Man kann Zeigervariablen auch Adressen zuweisen:

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

Beachte: Typen von Zeigern benötigen das `*` rechts (also `int *`), Namen von Zeigervariablen links (also `*p`)!

Solche dynamischen Variablen brauchen einen eigenen Speicherbereich, in dem sie abgespeichert werden. Dieser von DOS verwaltete Bereich heißt **HEAP**.

PROGRAMM 8: Demo-Programm zur Verwendung von Zeigervariablen

```
/* Demoprogramm für die Verwendung von Zeigern */
#include <stdio.h>
#include <conio.h>
void main()
{
  char name[30] ;
  char *gruss; /* Zeiger auf einen undefinierten Bereich */
  clrscr();
  gruss = "Guten Abend,"; /* gruss == &gruss[0]
     Der Zeiger gruss zeigt jetzt auf eine Stringkonstante
     "Guten Abend" - diese hat aber keinen (!!!) Variablennamen …
     dynamische Variable */
  printf ("\nIhr Name?");
  scanf("%s",name);
  printf("\n%c",*gruss);    /* Gibt "G" aus */
  printf("\n%c",*(gruss+1));/* Gibt "u" aus */
  printf("\n%c",*gruss+1);  /* Gibt "H" (= ASCII('G')+1) aus */
  printf("\n%s",gruss);     /* Gibt "Guten Abend" aus */
  printf("\n%s%s\n",gruss,name);
                            /* printf(const char *format,argumente) */
  getch();
}
```

Nun haben wir die Kenntnisse, um eine eigene C-Funktion zu programmieren. Ziel dieser Funktion ist eine fertige Eingaberoutine, mit der Texte eingegeben werden können. Optisch dienen die bekannten Windows-Eingabefelder als Vorbild. Selbstverständlich soll es möglich sein, die Eingabe zu editieren, falsche Eingaben zu korrigieren usw.

Dazu ist es nötig, jeden einzelnen Tastendruck abzufangen und zu überprüfen!

PROGRAMM 9: String-Eingabe-Routine

```
/* M_strin.C
   Autor: Ing. Josef Müllner, WIFI St. Pölten
   adaptiert von Chr. Zahler
*/
#include "stdio.h"
#include "conio.h"
#include "string.h"
/*--------------------------------------------------------------------*/
#define ENTER       13
#define ESC         27
#define CTRL_Y      25
#define DEL         83
#define BACKSPACE    8
#define CRSRUP      72
#define CRSRDOWN    80
#define PAGEUP      73
#define PAGEDOWN    81
#define F1          59
/*--------------------------------------------------------------------*/
void StringInput (int x, int y, int lg, char *p_string, char *p_et,
             int txtcolor, int bgcolor)
/* Koordinaten x,y, Länge des Eingabefeldes, Ptr auf Programmvariable,
   Ptr auf Programmvariable Endetaste
   Farbe des Textes, Farbe des Hintergrundes
   Farben Originalzustand wird wieder hergestellt

03.02.92  Stringinput in "C"
          Einfache Stringinput_routine mit folgenden Funktionen
          Editiertasten:
          CTRL_Y u. DELete ... löscht Feld, Cursor zum Beginn
          Backspace ... 1 Schritt nach links (am Feldende 2 Schritte)
          ESC Abbruch ... holt default Wert ins Feld
          Edit-Keys : F1, CursorUp, CursorDown, PageUp, PageDown, CR,
*/
{
  struct text_info txtinfo;       /* Record für Textattr. .... */
  int i;
  int cursor, slen;               /* Cursor-Position, Stringlänge*/
  int hp;                         /* Hilfspointer */
  char hstg[80];                  /* Hilfsstring */
  char strich[80];                /* String */
```

```
unsigned char NormZeich;        /* normales ASCII Zeichen */
unsigned char ErwZeich;         /* erweitertes Zeichen */
gettextinfo(&txtinfo);          /* akt.  Textinfodaten sichern */
textcolor(txtcolor);            /* schalte auf gewünschte Farb.*/
textbackground(bgcolor);
/* fülle strich wegen "\0"*/
/* \0 ende an die richtige Stelle
   Buffer[] enthält restlichen Füllzeichen */
for (i=0; i<=79; strich[i]='_', i++)
  strich[lg-strlen(p_string)]='\0';
if (strlen(p_string)== lg)   /* Wie lange ist übergeb. String   */
  cursor = strlen (p_string);
else  /* position des Cursors nach Text */
  cursor = strlen (p_string)+1;
gotoxy(x,y);
cprintf("%s",p_string);    /* Defaulttext zum Schirm */
cprintf("%s",strich);      /* Strasse_____ (_) zum Schirm */
x=x-1;                     /* wenn p=1 -> x+p =original x sein*/
strcpy(hstg, p_string);    /* leg Sicherheitskopie an */
hp = cursor;               /* merke auch orginal Cusorpos. */
do
{
  gotoxy(x+cursor, y);     /* setze Cursor */
  NormZeich = getch();     /* hole Zeichen ohne Anzeige */
  if (! NormZeich)         /* Normzeich ist 00 bei erw. Zeich !*/
    ErwZeich = getch();    /* nach 00 hole zweites Byte */
  else
    ErwZeich = 0;          /* es war ein 1byte zeichen ! */

  switch (ErwZeich)
  {
  case BACKSPACE: ErwZeich = BACKSPACE; break;
  case CTRL_Y: ErwZeich = CTRL_Y; break;
  case ESC: ErwZeich = ESC; break;
  case ENTER : ErwZeich = ENTER;  break;
  } /*  end  */
  switch (ErwZeich)        /* je nach Steuerzeichen */
  {
  case CTRL_Y:             /* delete kompl. Feld */
  case DEL:                /* gleiche Funktion*/
    strich[lg-strlen(hstg)]='_';    /* Del. \0 für strnset !*/
    cursor=1;              /* Eingabe löschen */
    *p_string = '\0';      /* Endemarke an den Beginn der
                              Programmvariablen */
    strnset(strich,'_', 80);  /* string number set; Ziel,
                                 Zeichen,Anzahl */
    strich[lg]='\0';       /* \0 ende an die richtige Stelle
                              Buffer[] enthält restl. Füllzei.*/
    gotoxy(x+cursor, y);
    cprintf("%s", strich); /* Feld wieder vorzeichen */
    break;
  case BACKSPACE:
    if (cursor > 1 )       /* BS nur erlaubt > 1 */
    { cursor--;            /* ptr nach links */
      *(p_string+)='\0';   /* neues Ende, array ab [0]*/
      gotoxy(x+1, y);      /* Cursor in Startpos. */
      cprintf("%s",p_string);   /* string neu anzeigen */
                           /* Rest am Schirm mit "_" füllen */
      for (i=1; i <= lg-strlen(p_string); cprintf("_"), i++);
    }
    break;
  case ESC:
    strcpy(p_string, hstg);    /* stelle Orginal her   */
    cursor=hp;             /* Orginal Cursor Pos. */
    for (i=0; i<=79; strich[i]='_', i++)      /* fülle strich */
      strich[(p_string)]='\0';
      /* \0 ende an die richtige Stelle; Strich[] enthält
      restliche Füllzeichen */
    gotoxy(x+1,y);         /* oben x-1                */
    cprintf("%s",p_string);/* Defaulttext zum Schirm */
    cprintf("%s",strich);  /* Strasse_____ (_) zum Schirm */
    break;
  default:
    if  (NormZeich >= 32 )
    {
      if (cursor < lg )     /* Noch Platz ? */
      { cursor=cursor+1;
        strncat(p_string, &NormZeich, 1);/* Concatenate anh */
        cprintf("%c",NormZeich); /* Auch zum Schirm */
      }  /* Wenn noch Platz */
      else        /* Feld voll */
      {
        cprintf("%c",0x07);      /* pieps am Ende */
        *(p_string+)='\0'; /*neues Ende, array ab [0]*/
        strncat(p_string, &NormZeich, 1);/* Concatenate anh */
        cprintf("%c",NormZeich);         /* Auch zum Schirm */
      }
    }   /* NormZeich >= 32 */
  } /* end switch (ErwZeich) */
  } while (ErwZeich !=PAGEUP && ErwZeich != PAGEDOWN &&
     ErwZeich != CRSRUP && ErwZeich != CRSRDOWN &&
     ErwZeich != F1 && ErwZeich !=  ENTER);
  slen = strlen (p_string);
  if (slen < lg)                  /* restl. '_' am Schirm ? */
  {
    gotoxy(x+slen+1, y);
    strich[lg-strlen(hstg)]='_';   /* Del. \0 für strnset !*/
    strnset(strich,' ', 80);       /* string number set;
                                      Ziel, Zeichen, Anzahl */
    strich[lg-strlen(p_string)]='\0';
    /* \0 ende an die richtige Stelle, strich[] enthält
       restl. Füllzei. */
    cprintf  ("%s",strich);  /* restl. "___" löschen */
  }
  *p_et = ErwZeich ;  /* Endetaste ans HP übergeben */
  textattr(txtinfo.attribute); /* setze urspr. Attribute wieder */
}/* ende Funktion */
/*---------------------------------------------------------------------*/
```

10Dateien

10.1 Zugriff mittels Filepointer (Stream-I/O)

Mit jeder Datei ist ein Filepointer vom Typ `*FILE` verbunden. Dieser zeigt auf eine Struktur, in der alle wesentlichen Merkmale der Datei verzeichnet sind. Daten werden zunächst in einem Zwischenspeicher (**Puffer**) im RAM gehalten, dann erst auf Diskette oder Festplatte geschrieben.

In C ist die Verarbeitung von **Binär- und Textdateien** möglich.

Zuerst muss ein Pointer auf den Datentyp deklariert werden:

```
FILE *datei;
```

Nun wird die Datei mit Hilfe der Funktion `fopen()` geöffnet. (Genauer: Es wird ein "Datenstrom", eine Art "Verbindung" zur Datei auf der Festplatte/Diskette... hergestellt.)

Syntax von fopen()

```
FILE *fopen (const char *filename, const char *mode);
```

wobei:

`filename`	DOS-Pfad der Datei als Stringkonstante
`mode`	Art, wie die Datei geöffnet werden soll

Für `mode` sind folgende Werte möglich – Zusammensetzung je eines Wertes aus der 1. und 2. Gruppe:

1. Gruppe

`"r"`	(read) Datei zum Lesen öffnen; die Datei muss bereits existieren
`"w"`	(write) Datei zum Schreiben öffnen; falls die Datei bereits existiert, wird ihr Inhalt gelöscht
`"a"`	(append) Datei zum Schreiben am Dateiende (zum Anhängen von Daten) öffnen
`"r+"`	Datei zum Lesen und Schreiben öffnen; die Datei muss bereits existieren
`"w+"`	Datei zum Lesen und Schreiben öffnen; falls die Datei bereits existiert, wird ihr Inhalt gelöscht. Falls die Datei noch nicht existiert, wird sie angelegt.
`"a+"`	Datei zum Lesen und Anhängen von Daten öffnen; falls die Datei noch nicht existiert, wird sie angelegt.

2. Gruppe

`"b"`	Bearbeitung im Binärmodus
`"t"`	Bearbeitung im Textmodus

Ist die Ausführung der Funktion `fopen()` fehlerhaft, so wird der Nullpointer zurückgeliefert.

Beispiel

```
FILE *datei;
datei = fopen("C:\\PROG\\DATEN.TXT","r+t");
```

Wichtig: Es ist dabei auch möglich, eine Verbindung zu einem **Gerät** herzustellen:

```
datei = fopen("COM1:","w");
```

Nach der Bearbeitung der Datei muss sie mit der Funktion `fclose()` geschlossen werden.

Syntax

```
int fclose (FILE *filepointer);
```

Beispiel

```
fclose (datei);
```

Übersicht über Funktionen, die zum Lesen von und Schreiben auf Dateien dienen:

a) Lesen und Schreiben einzelner Zeichen

```
int fputc(int c, FILE *fp);
```

schreibt das Zeichen `c` in die Datei `fp`; bei erfolgreicher Ausführung wird `c` zurückgegeben, sonst `EOF` (End of File; Konstante mit dem Wert –1).

```
int fgetc(FILE *fp);
```

liest ein Zeichen aus der Datei `fp`; bei Dateiende oder fehlerhafter Ausführung `EOF` als Rückgabewert

b) Lesen und Schreiben von

```
int fputs(char *s, FILE *fp);
```

schreibt den String `s` in die Datei `fp`; bei erfolgreicher Ausführung wird das zuletzt angegebene Zeichen zurückgegeben, sonst `EOF` (End of File; Konstante mit dem Wert –1). fputs hängt kein \0 an!

```
char *fgets(char *s, int n, FILE *fp);
```

liest den String `s` aus der Datei `fp`; bei erfolgreicher Ausführung wird einer Pointer auf den String geliefert, bei Dateiende oder fehlerhafter Ausführung `NULL` als Rückgabewert. `fgets()` liest Zeichen für Zeichen bis zum nächsten newline-Code \n, maximal jedoch n-1 Zeichen und fügt an die gelesene Zeichenkette noch \0 an.

c) Formatierte Dateieingabe und Dateiausgabe

```
int fprintf(FILE *fp, const char format, ...);
```

Formatierte Ausgabe in eine Datei. Arbeitet genauso wie `printf()`, nur erfolgt die Ausgabe an die Datei `fp`. Gibt bei fehlerfreier Ausführung die Anzahl der ausgegebenen Zeichen zurück, sonst `EOF`.

```
int fscanf (FILE *fp, const char format, ...);
```

Formatiertes Lesen aus der Datei `fp`. Gibt die Anzahl der Felder zurück, die fehlerfrei gelesen werden konnten.

d) Lesen und Schreiben von Blöcken (NUR BEI BINÄRDATEIEN SINNVOLL)

```
size_t fwrite(const void *ptr, size_t size, size_t n, FILE *fp);
```

Schreibt `n` Datenelemente der Größe `size_t`, deren Startadressen durch `ptr` gegeben ist, in die durch `fp` angegebene Datei. Zurückgeliefert wird die Anzahl der Elemente, die fehlerfrei gelesen werden konnten.

```
size_t fread(const void *ptr, size_t size, size_t n, FILE *fp);
```

Liest `n` Datenelemente der Größe `size` aus der durch `fp` angegebenen Dateien in den Speicherbereich, auf den `ptr` zeigt. Zurückgeliefert wird die Anzahl der Elemente, die fehlerfrei gelesen werden konnten. Ist das Dateiende bereits vor dem Aufruf von `fread()` erreicht, so wird der Wert 0 zurückgeliefert.

Relativer Zugriff

Der Zugriff auf Dateien ist auf mehrere Arten möglich:

- **sequentiell**: Datensätze werden hintereinander gelesen. Länge der Datensätze flexibel. Nachteil: Es muss bei jedem Suchvorgang vom Dateianfang weggelesen werden – sehr langsam! (Wurde etwa bei den *.INI-Dateien von Windows 3.1x verwendet.)
- **relativ**: Datensätze sind alle gleich lang, damit kann man "direkt" auf jeden Datensatz zugreifen.
- **index-sequentiell:** Über eine Index-Datei ist ein "direkter" Zugriff auf einen bestimmten Datensatz möglich.

Beim relativen Zugriff gibt es einen "Dateizeiger", der immer die aktuelle Datensatznummer gespeichert hat. Diesen Dateizeiger kann man mit folgenden Funktionen beeinflussen:

```
void rewind(FILE *fp);
```

setzt die Position des Dateizeigers auf den Dateianfang zurück

```
int fseek(FILE *fp, long offset, int whence);
```

Setzt die momentane Position innerhalb einer Datei für die folgenden Lese- und Schreiboperationen. Der Parameter `offset` gibt die Entfernung in Bytes an, der Parameter `whence` sagt, ob diese Entfernung vom Dateianfang, vom Dateiende oder relativ zur aktuellen Position gerechnet werden soll. Dafür gibt es in stdio.h drei vordefinierte Konstanten:
`SEEK_SET == 0`: relativ zum Dateianfang
`SEEK_CUR == 1`: relativ zur momentanen Position
`SEEK_END == 2`: relativ zum Dateiende

```
long ftell(FILE *fp);
```

Liefert die momentane Position des Dateizeigers. Bei einem Fehler wird –1 zurückgegeben.

10.2 Systemnaher Zugriff auf den Hauptspeicher (ungepufferte Dateiverarbeitung)

DOS bietet eine zweite Art des Dateizugriffs. Jeder geöffneten Datei wird von DOS eine "Dateizugriffsnummer" (= File Handle) zugeordnet, über die diese Datei dann angesprochen werden kann. Die maximale Anzahl der möglichen File Handles wird in der `CONFIG.SYS`-Datei durch den Eintrag `FILES=nn` festgelegt. Diese Art des Zugriffs ist sehr schnell.

Dafür gibt es folgende Funktionen, die aber in der Online-Hilfe detailliert beschrieben werden:

```
int open(const char *pathname, int access, unsigned mode);
```

Bibliothek `IO.H`; öffnet eine Datei für Lese- und Schreibzugriff; bei erfolgreicher Ausführung wird der File Handle zurückgeliefert. Der Parameter `access` gibt die Öffnungsart an, die mit den O_xxx-Konstanten in der Bibliothek `FCNTL.H` festgelegt wird. `mode` ist nur bei neu erzeugten Dateien notwendig und beschreibt, ob Lese- und/oder Schreibzugriff erlaubt ist.

```
int close(int handle);
```

Schließt die Datei.

```
int read(int handle, void *buf, unsigned len);
```

Liest Daten von einer Datei.

```
int write(int handle, void *buf, int nbyte);
```

Schreibt Daten in eine Datei.

```
long lseek(int handle, long offset, int fromwhere);
```

analog (setzt Dateizeiger)

```
long tell(int handle);
```

analog ftell (liefert Position des Dateizeigers)

10.3 Wichtige DOS-Schnittstellen:

Es ist gerade beim Arbeiten mit Dateien wichtig, auch einmal eine Datei löschen oder umbenennen zu können. Daher gibt es in C für alle wesentlichen DOS-Befehle auch entsprechende C-Funktionen.

Eine Auswahl

```
int chdir (const char *path);
```

DOS: cd
Fehlerfrei: 0
Fehler: -1

```
int mkdir (const char *path);
```

DOS: md
Fehlerfrei: 0
Fehler: -1

```
int rmdir (const char *path)
```

DOS: rd
Fehlerfrei: 0
Fehler: -1

```
int getcurdir (int drive, char *dir);
```

drive: 0 = aktuell; 1 = A: 2 = B: usw.
ermittelt aktuelles Verzeichnis
Fehlerfrei: 0
Fehler: -1

```
char *getcwd(char *buf, int buflen);
```

get current working directory; liefert momentan gesetztes Arbeitsverzeichnis zurück
Fehlerfrei: Zeiger auf buf-String
Fehler: `NULL`

```
int setdisk(int drive);
```

drive: 0 = A: 1 = B: usw.
setzt ein Laufwerk als Standard
Fehlerfrei: Gesamtzahl der Laufwerke

```
int getdisk(void);
```

ermittelt das momentan gesetzte Laufwerk
Fehlerfrei: Laufwerksnummer (0=A, 1=B, ...)

```
int findfirst(char *path, struct ffblk *ffblk, int attrib);
```

`ffblk`: Dateityp für Block, der Dateiname, Uhrzeit, Datum und Größe der Datei enthält
Findet ersten Eintrag im Verzeichnis, der mit dem Suchmuster `path` übereinstimmt.
Fehlerfrei: 0
Fehler: -1

```
int findnext(struct ffblk *ffblk);
```

Findet den nächsten Eintrag, der mit dem in `findfirst` definierten Suchmuster übereinstimmt.
Fehlerfrei: 0
Fehler: -1

```
int rename(const char *oldname, const char *newname);
```

DOS: ren
Fehlerfrei: 0
Fehler: -1

```
int unlink(const char *filename);
```

DOS: del
Fehlerfrei: 0
Fehler: -1

```
long filelength (int handle);
```

Fehlerfrei: Dateigröße in Bytes
Fehler: -1

```
int system (const char *command);
```

führt einen DOS-Befehl aus, indem `COMMAND.COM` geladen wird
Fehlerfrei: Exitcode von `COMMAND.COM` bzw. des Programms

11.4 Standarddateien

In C gibt es drei Standarddateien, die immer geöffnet sind:

`stdin` Standardeingabedatei (normalerweise `CON`, also die Tastatur)

`stdout` Standardausgabedatei (normalerweise `CON`, also der Bildschirm)

`stderr` Standardfehlerdatei

So wirkt etwa die Funktion `printf()` auf `stdout`, normalerweise also auf den Bildschirm. Man kann allerdings von DOS aus die Ausgabe umlenken. Annahme: Wir haben ein fertig übersetztes Programm `prog10.exe`. Dann könnte man mit

```
prog1 > prn
```

die Ausgaben auf den Drucker umleiten.

```
PROGRAMM 10: Demo-Datenbank (Adressen)
#include <stdio.h>
#include <conio.h>
#include <string.h>
#include <ctype.h>
#include "m_.c"

/************************  DATEITYPDEFINITIONEN *******************/
typedef enum {erstellen, erweitern} Aktion;
struct Person {
 char name [20];
 char vorname [15];
 char titel [5];
 char strasse [15];
 char plz [7];
 char ort [18];
};
const int sl=sizeof(struct Person);
/************************** PROTOTYPEN ***************************/
void hauptmenue (void);
void maske (char *ueberschrift);
void schreiben (char dateiname[], Aktion wastun);
void aendern (char dateiname[]);
void loeschen (char dateiname[]);
void ausgabe (char dateiname[]);
/************************  IMPLEMENTIERUNG ********************/
void main(void)
{
  hauptmenue();
} /* main */
void maske (char *ueberschrift)
{
  textcolor(WHITE);
  textbackground(BLACK);
  clrscr();
  gotoxy(3,2); puts(ueberschrift);
  gotoxy(3,3); puts("=============");
  gotoxy(3,5); puts("Name:");
  gotoxy(3,7); puts("Vorname");
  gotoxy(3,9); puts("Titel:");
  gotoxy(3,11); puts("Straße:");
  gotoxy(3,13); puts("Postleitzahl");
  gotoxy(3,15); puts("Ort:");
} /*  maske*/

void schreiben (char dateiname[], Aktion wastun)
{
  FILE *datei;
  struct Person Kunde;
  char antw;
  if (wastun == erstellen)
    datei = fopen (dateiname, "wb");
  else /* wastun == erweitern */
  {
    datei = fopen (dateiname, "ab");
    fseek(datei,0L,SEEK_END);
  }
  do
  {
    maske("E I N G A B E");
    strcpy(Kunde.name,"");
    strcpy(Kunde.vorname,"");
    strcpy(Kunde.titel,"");
    strcpy(Kunde.strasse,"");
    strcpy(Kunde.plz,"");
    strcpy(Kunde.ort,"");
    StringInput(18,5,20,Kunde.name,"\r",WHITE,BLUE);
    StringInput(18,7,15,Kunde.vorname,"\r",WHITE,BLUE);
    StringInput(18,9,5,Kunde.titel,"\r",WHITE,BLUE);
    StringInput(18,11,15,Kunde.strasse,"\r",WHITE,BLUE);
    StringInput(18,13,7,Kunde.plz,"\r",WHITE,BLUE);
    StringInput(18,15,18,Kunde.ort,"\r",WHITE,BLUE);
    fwrite(&Kunde,sl,1,datei);
    gotoxy(15,21);puts("Weitere Eingaben? [J/N]");
    gotoxy(40,21);antw = toupper (getch());
  } // do
  while (antw!='N');
```

```
  fclose(datei);
} /* schreiben */

void hauptmenue (void)
{
  char auswahl;
  char dateiname[30];

  strcpy(dateiname,"ADRESSEN.DBK");
  textcolor(WHITE);
  textbackground(BLACK);
  do
  {
    clrscr();
    gotoxy(10,2); puts("A d r e ß d a t e n b a n k");
    gotoxy(10,3); puts("==========================");
    gotoxy(5,5); puts("Wählen Sie aus:");
    gotoxy(5,7);  puts("1  Erstellen einer neuen Datei");
    gotoxy(5,9);  puts("2  Erweitern einer bestehenden Datei");
    gotoxy(5,11); puts("3  Ändern einer bestehenden Datei");
    gotoxy(5,13); puts("4  Löschen von Datensätzen "
                       "einer bestehenden  Datei");
    gotoxy(5,15); puts("5  Ausgabe");
    gotoxy(5,17); puts("9  Programmende");
    gotoxy(5,20); puts("Auswahl: ");
    gotoxy(15,20);
    auswahl = getche();
    switch (auswahl)
    {
    case '1': schreiben(dateiname, erstellen);
              break;
    case '2': schreiben(dateiname, erweitern);
              break;
    case '3': aendern(dateiname);
              break;
    case '4': loeschen(dateiname);
              break;
    case '5': ausgabe(dateiname);
              break;
    case '9': break;
    } //
  } // do
  while (auswahl!='9');
} /* hauptmenue */

void aendern (char dateiname[])
{
  FILE *datei;
  struct Person Kunde;
  int nummer;
  maske ("Ä N D E R N");
  gotoxy(20,2);puts("Datensatznummer eingeben:");
  gotoxy(50,2);scanf("%2d",&nummer);
  datei = fopen (dateiname, "r+b");
  fseek(datei,(long)(nummer-1)*sl,0);
  fread(&Kunde,sl,1,datei);
  gotoxy(18,5); puts(Kunde.name);
  gotoxy(18,7); puts(Kunde.vorname);
```

Christian Berger

```
  gotoxy(18,9); puts(Kunde.titel);
  gotoxy(18,11); puts(Kunde.strasse);
  gotoxy(18,13); puts(Kunde.plz);
  gotoxy(18,15); puts(Kunde.ort);
  StringInput(18,5,20,Kunde.name,"\r",WHITE,BLUE);
  StringInput(18,7,15,Kunde.vorname,"\r",WHITE,BLUE);
  StringInput(18,9,5,Kunde.titel,"\r",WHITE,BLUE);
  StringInput(18,11,15,Kunde.strasse,"\r",WHITE,BLUE);
  StringInput(18,13,7,Kunde.plz,"\r",WHITE,BLUE);
  StringInput(18,15,18,Kunde.ort,"\r",WHITE,BLUE);
  StringInput(datei,(long)(-sl),SEEK_CUR);
  StringInput(&Kunde,sl,1,datei);
  StringInput(datei);
} /* aendern */

void loeschen (char dateiname[])
{
  FILE *datei;
  FILE *hilfsdatei;
  struct Person Kunde;
  int nummer,i=0;
  charantw;
  char temp[]="TEMP.TMP";
  maske ("L Ö S C H E N");
  (20,2);puts("Datensatznummer eingeben:");
  (50,2);scanf("%2d",&nummer);
  datei = fopen (dateiname, "rb");
  fseek(datei,(long)(nummer-1)*sl,SEEK_SET);
  fread(&Kunde,sl,1,datei);
  gotoxy(18,5); puts(Kunde.name);
  gotoxy(18,7); puts(Kunde.vorname);
  gotoxy(18,9); puts(Kunde.titel);
  gotoxy(18,11); puts(Kunde.strasse);
  gotoxy(18,13); puts(Kunde.plz);
  gotoxy(18,15); puts(Kunde.ort);
  fclose(datei);
  gotoxy(30,20); puts ("Wirklich löschen [j/n] ?");
  antw = toupper(getch());
  if (antw=='J')
  {
    datei = fopen (dateiname, "rb");
    hilfadatei = fopen (temp, "wb");
    while (fread(&Kunde,sl,1,datei))
    {
      if (i!=nummer-1) fwrite(&Kunde,sl,1,hilfsdatei);
        i++;
     } // while
     fclose (hilfsdatei);
     fclose (datei);
     unlink(dateiname);
     rename(temp,dateiname);
  } // if antw =='J'
} /*loeschen */

void ausgabe (char dateiname[])
{
  FILE *datei;
  struct Person Kunde;
  datei = fopen (dateiname, "rb");
  while (fread(&Kunde,sl,1,datei))
  {
     maske ("A U S G A B E");
     gotoxy(18,5); puts(Kunde.name);
     gotoxy(18,7); puts(Kunde.vorname);
     gotoxy(18,9); puts(Kunde.titel);
     gotoxy(18,11); puts(Kunde.strasse);
     gotoxy(18,13); puts(Kunde.plz);
     gotoxy(18,15); puts(Kunde.ort);
     gotoxy(20,20); puts("Weiter - Taste drücken!");
     getch();
  }
  fclose(datei);
} /* ausgabe */
```

Literatur

Kernighan/Richie: Programmieren in C. Hanser Verlag, 1983