Programmieren in C

E. Schachinger

8. Oktober 2001

Inhaltsverzeichnis

Kapitel 1

Grundlegendes

1.1 Variable

Variable sind die Grundelemente eines Programmes, da sie überhapt erst eine Manipulation von Daten erlauben. Einem Datum ist stets eine Variable zugeordnet. Die formale Definition einer Variablen besteht aus der Vergabe eines Variablen-Namens, welcher über eine Variablen-Deklaration mit einem Variablen-Typ verknüpft wird. Variable, welche nicht über eine Deklaration eingeführt wurden, werden vom Compiler nicht erkannt und führen zu einer entsprechenden Fehlermeldung. Rechnen kann man mit Variablen natürlich erst dann, wenn ihnen ein Wert zugewiesen wurde.

1.1.1 Variablen - Name

Variable werden durch einen Namen identifiziert. Solche Variablen - Namen können aus einer beliebigen Kombination von Buchstaben und Ziffern, sowie dem underscore '_' Zeichen bestehen. Das erste Zeichen muß ein Buchstabe sein, wobei das underscore Zeichen als Buchstabe gilt. Der Name kann max. 256 Zeichen lang sein; C ist weiters sensitiv auf Groß- und Kleinschreibung. Es gibt auch noch eine ganze Anzahl nicht zulässiger Variablen - Namen, nämlich alle jene, welche Schlüsselworten der C - Programmiersprache entsprechen. Wir werden diese Schlüsselworte im Rahmen dieses Kurses noch ausführlich kennen lernen.

1.1.2 Variablen - Typ, -Größe, -Deklaration und Konstante

Variablen-Typ:

`void`	der 'leere' Typ.
`char`	speichert ein Zeichen (ASCII).
`int`	speichert eine ganze Zahl.
`float`	speichert eine Fließkommazahl in einfacher Genauigkeit.
`double`	speichert eine Fließkommazahl in doppelter Genauigkeit.

Zusätzlich gibt es:

`short int`	oder `short`, 'kurze' ganze Zahl.
`long int`	oder `long`, 'lange' ganze Zahl.
`unsigned int`	positive ganze Zahl.
`unsigned char`	Zeichen ohne Vorzeichen (Byte).
`register int`	ganze Zahl, welche in einem Register gespeichert wird.

Die Modifikation `const` zum Datentyp zeigt an, daß die Variable unveränderlich ist, also:

```
const char
const int
const float
const double
const unsigned int
```

Variablen-Deklaration:

Die Variablen-Deklaration definiert Variable innerhalb des Programmes oder eines Programmteiles und verknüpft den Variablen-Namen mit dem Variablen-Typ und kann auch dazu verwendet werden der Variablen einen Anfangswert zuzuweisen (Initialisierung) . An dieser Stelle ist es notwendig darauf hinzuweisen, daß manche Compiler Variable prinzipiell mit dem Wert Null inizialisieren, manche tun dies wiederum nur im Debugg-Mode, und manche gar nicht.

Beispiele:

```
int x,nNorm,res;
long int field[1000];
double dX,dY,dZ;
```

```
const double Pi = 4.0*atan(1.0); /* Pi in Maschinengenauigkeit
*/
```

Man sieht also, daß die Deklaration von Variablen durchaus bereits komplexe Formen annehmen kann; die Bedeutung der einzelnen Statements wird noch genauer zu erläutern sein. Im der letzten Deklaration wird der Variablen mit dem Namen `Pi` vom Typ `const double` der Zahlenwert π zugeordnet, welcher über $4 * \arctan(1)$ mit Maschinengenauigkeit bestimmt wird. (Angefügt ist noch ein Kommentar, welcher zwischen `/* */` eingeschlossen ist.) Wir erkennen: die Variablendeklaration kann bereits dazu benützt werden einer Variablen einen bestimmten Wert zuzuweisen.

Es ist zu diesem Zeitpunkt wesentlich darauf hinzuweisen, daß der aktuelle Speicherbedarf der einzelnen Datentypen von der Anlage abhängt, auf welcher ein C-Programm entwickelt oder exekutiert wird. Dieser Platzbedarf kann durch den Systembefehl `sizeof(...)` in Bytes (8 Bit pro Byte) bestimmt werden. Ganz wichtig ist dieser Befehl zur Bestimmung der größten Zahl, welche in den Datentypen `unsigned char`, `int`, `unsigned int`, `short int` und `long int` abgespeichert werden kann. (Datenüberlauf ist eine sehr ungute Fehlerquelle!)

Konstante:

`124`	ganze Zahl.
`124.0`	Fließkommazahl.
`0.12e+3`	Fließkommazahl in wissenschaftlicher Notation.
`0.12E-4`	Fließkommazahl in wissenschaftlicher Notation.
`124L`	ganze Zahl vom Typ `long int`.
`0x1F`	ganze Zahl in Hexadezimalschreibweise, Typ `int`.
`037`	ganze Zahl in Oktalschreibweise, Typ `int`.
`'A'`	das Zeichen 'A', Typ `char`.
`"Zeichenkette"`	eine Zeichenkette, welche mit ASCII NUL abgeschlossen wird.

```
'\0' '\t' '\n' '\\' '\014' '\x14' '\'' '\%' '\"' ""
```

sind spezielle Konstante vom Typ `char`, welche Zeichen aus der ASCII-Reihe symbolisieren: NUL, TAB, NEWLINE, BACKSLASH, ASCII oktal 14, ASCII hexadezimal 14, der Apostroph, das %-Zeichen und der Doppelapostroph, schließlich ist noch die leere Zeichenkette angegeben.

Programm 1:

Schreiben Sie ein Programm, welches Ihnen die Länge unterschiedlichster Datentypen und Konstante in Byte angibt. Es ist dazu der Befehl `sizeof(...)` heranzuziehen.

1.2 Operatoren

Das Besondere an der Programmiersprache C ist die große Vielfalt an Operatoren, welche zur Verknüpfung von Variablen, sogenannten *Ausdrücken*, herangezogen werden können.

1.2.1 Arithmetische Operatoren

Es sind dies die binären arithmetischen Operatoren, welche zur arithmetischen Verknüpfung von Variablen herangezogen werden:`+` (Addition), `-` (Subtraktion), `*` (Multiplikation), `/` (Division). Es gibt weiters den Modulus-Operator `%`, welcher nur Variable vom Typ `short int`, `int` und `long int` verknüpfen kann. Schließlich gibt es noch das unäre `-`.
Operator Präzedenz:

```
* / %
+ -
```

Die Operatoren gruppieren ferner von links nach rechts. Eine geeignet eingeführte Klammerung hebt diese Präzedenz auf:

```
double a,b,c;
...        a * b       +c;
     zuerst, zwischenspeichern
... a*      (b + c)      ;
       zuerst, zwischenspeichern
...    -a   *b;.
     unäres Minus
```

Werden Variable unterschiedlichen Typs miteinander verknüpft, so wird das Ergebnis der Verknüpfung auf den höchstwertigen Typ umgewandelt:

```
double a;
int b;
... a*b; /* Ergebnis ist vom Typ double. */
```

1.2.2 Der Wertzuweisungsoperator

Er wird durch das Zeichen `=` symbolisiert:

```
e1 = e2;
```

Hier ist `e1` eine Variable, welcher jener Wert zugewiesen wird, welcher dem Ausdruck `e2` entspricht. Damit darf `e1` nicht vom Typ `const ...` oder gar eine Konstante sein! Man bezeichnet `e1` auch als einen *lvalue*, also einen

lefthand value, etwas, was links vom Wertzuweisungsoperator stehen darf. `e2` wird somit nicht notwendigerweise ein Variablenname sein, es kann jeder gültige Ausdruck, also auch selbst wiederum eine Wertzuweisung sein. Solche Ausdrücke bezeichnet man dann konsequenter Weise als *rvalues*. Beispiele:

```
a = b = c = 0;
```
Die Variablen `a`, `b` und `c` werden auf den Wert der Konstanten Null gesetzt.

```
a = c = b+d;
```
Den Variablen `a` und `c` wird der Wert der arithmetischen Verknüpfung `b+d` zugewiesen. Dies entspricht der Statementfolge:
```
a = b+d;
c = b+d;
```

Man kann diesen Befehl auch 'besser' lesbar machen:
```
a = (c = b+d);
```
womit auch transparent wird, daß `(c = b+d)` wiederum ein 'Ausdruck' im Sinne der Programmiersprache C ist.

Wichtig!

> Jede Verknüpfung zweier Variabler hat einen Wert, damit auch die Wertzuweisung. Ihr Wert ist der Wert des Ausdruckes, welcher sich durch Ausführen der Operationen auf der rechten Seite des `=` Operators ergibt.

Programm 2:

Schreiben Sie ein kleines Programm zur arithmetischen Verknüpfung von Variablen. Weiters: wie findet man bei Fließkommazahlen die Stellen hinter dem Komma? Umwandlung ASCII-Zeichen in Variable vom Typ `int`.

1.2.3 Relationale und logische Operatoren

Die relationalen Operatoren sind:

`>`	größer
`>=`	größer gleich
`<`	kleiner
`<=`	kleiner gleich
`==`	gleich
`!=`	nicht gleich
`&&`	logische UND-Verknüpfung
`||`	logische ODER-Verknüpfung
`!`	unäres nicht.

Der Wert einer solchen Verknüpfung ist stets 0 (`false`), wenn sie nicht erfüllt ist, oder nicht gleich Null (`true`).

Präzedenz:

`>`, `>=`, `<`, `<=`
`==`, `!=`
`&&`, `||`

Treten `&&` und `||` im selben Ausdruck auf, so werden sie stets von links nach rechts ausgewertet, wobei in solchen Mehrfachausdrücken `&&` Präzendenz über `||` hat.

1.2.4 Inkrement- und Dekrement-Operatoren

Dies sind unäre und stellungsabhängige Operatoren:

`++`	inkrementiert um 1
`--`	dekrementiert um 1

Diese Operatoren können nur auf Variable wirken!

Der Ausdruck:

`++n`	hat den Wert `n+1` und die Variable `n` wird um 1 inkrementiert.
`n++`	hat den Wert `n` und die Variable `n` wird um 1 inkrementiert.
`--n`	hat den Wert `n-1` und die Variable `n` wird um 1 dekrementiert.
`n--`	hat den Wert `n` und die Variable `n` wird um 1 dekrementiert.

Programm 3:

Untersuchen Sie den Wert der folgenden Verknüpfungen bzw. die Wirkung von verschiedenen Operatoren:

```
int a,b=2,c=5,d;
double db=2.0,dc=5.0;
!(a = c > b)
b > c && dc > db
++c
c++
--b
b--
!(c%2)
!c%2
```

Begründen Sie die Ergebnisse.

1.2.5 Bitweise logische Operatoren

Es sind die folgenden Operatoren für Bitmanipulationen definiert:

`&`	bitweise UND
`|`	bitweise ODER
`<<`	bitweises Verschieben nach links
`>>`	bitweises Verschieben nach rechts
`^`	bitweises EXCLUSIVE OR
`~`	bitweises Einskomplement

Man verwendet den `&` Operator um Bits in einer Variablen auszublenden und den `|` Operator um Bits in einer Variablen zu setzen.
Die `<<` und `>>` Operatoren führen Rechts- und Linksverschiebungen des Bitmusters durch, welches in einer Variablen gespeichert ist. Es bedeutet etwa:

```
a << 3
```

daß der Inhalt der Variablen `a` um drei Stellen nach links verschoben wird. Die dabei rechts freiwerdenden Bitpositionen werden mit Null aufgefüllt. Also:

```
unsigned char a;
```

Befehl	Inhalt
`a = 1;`	`0000 0001`
`a << 3;`	`0000 1000`

Für Bitoperationen verwendet man am besten den Variablentyp `unsigned int`, da dann sichergestellt ist, daß bei der Operation `>>` freiwerdende Bitpositionen mit Null gefüllt werden. Beim Typ `int` gibt es Maschinenunterschiede.

Programm 4:

Wir verwenden eine Variable vom Typ `unsigend int` um eine Kette von $S = 1/2$ Spins darzustellen. Die Einstellung 'spin up' soll durch eine 1 und die Einstellung 'spin down' durch eine Null symbolisiert werden. Man kann daher jede Bitposition einer Variablen einem Spin der Kette zuordnen. Wieviele Spins können wir daher in einer Variablen vom Typ `unsigned int` speichern? Wir numerieren nun die Spinpositionen durch und verwenden das äußerst rechte Bit für die Position 0. Es soll nun durch eine Verschiebeoperation der Spin auf der Position 5 auf 'up' gesetzt werden. Überprüfen sie das Ergebnis

durch Ausdruck der Variablen im Hexadezimalcode (`%x` anstelle von `%d` als Platzhalter im `printf` Befehl!). Löschen Sie die Spineinstellung und setzen Sie nun den Spin auf der Position 5 auf 'up' unter Verwendung einer Bit-Operation. Überprüfen Sie das Ergebnis! Testen Sie ob ein bestimmer Spin auf 'up' gesetzt ist oder nicht unter Verwendung einer Testvariablen (Maske)!

1.2.6 Der Bedingungsoperator

Dieser Operator ist ternär, verknüpft also drei Ausdrücke:

`e1 ? e2 : e3`

Der Ausdruck `e1` wird dabei zuerst bearbeitet und ist eine logische Verknüpfung. Ergibt diese Verknüpfung den Wert `true`, dann erhält obiger Ausdruck den Wert `e2`, hat die logische Verknüpfung hingegen den Wert `false`, so erhält diese ternäre Verknüpfung den Wert `e3`.
Zwei typische Anwendungen sind die Bestimmung des Minimums, bzw. des Maximums:

```
double dz,da=2.0,db=5.0;
dz = da <= db ? da : db;
         /* dz erhält den Wert 2.0 zugewiesen, Minimum */
dz = da >= db ? da : db;
         /* dz erhält den Wert 5.0 zugewiesen, Maximum */
```

Es ist noch wesentlich anzumerken, daß `e2` und `e3` nicht vom selben Datentyp sein müssen.

1.2.7 Erweiterte Wertzuweisungsoperatoren

Diese Operatoren führen zu Ausdrücken der Form:

`e1 op= e2`

mit `op` als Platzhalter für die Symbole `+`,`-`, `*`, `/`, `%`, `>>`, `<<`, `&`, `^`, und `|`,also:

```
a += 2;    entspricht   a = a+2;
t <<= 3;   entspricht   t = t << 3;
usw.
```

1.2.8 Der Cast-Operator

Dieser unäre Operator ist wie folgt definiert:

`(vtyp)` *Ausdruck*

wobei `vtyp` ein Variablen-Typ ist. Dieser Operator wandelt nun den Wert des *Ausdruckes* in den gewünschten Variablen-Typ um.

Kapitel 2

Der Programmfluß

2.1 Anweisungen und Blöcke

Ein *Ausdruck*, welcher mit ; abgeschloßen wird, ist eine *Anweisung*. Es können mehrere Anweisungen durch Gruppierung in { ... } zu einem Block zusammengefaßt werden, welcher dann eine logische Einheit darstellt. Ein solcher Block kann auch Variablen-Deklarationen enthalten. So definierte Variable sind nur **innerhalb** dieses Blockes gültig. Man spricht von **lokalen Variablen**. Solche Deklarationen werden stets **vor** der ersten Anweisung in den Block eingefügt.

> **Ein Block kann stets die Stelle einer Anweisung einnehmen!**

2.2 Die `if - else` Verzweigung

Sie hat die folgende Struktur:

```
if (e1)
      Anweisung 1
( else
      Anweisung 2 )   optional
```

Es wird zunächst der Wert des Ausdrucks `e1` bestimmt; er wird als logischer Ausdruck interpretiert, welcher nur `true`, also ungleich Null, oder `false`, also gleich Null sein kann. Hat `e1` den Wert `true`, so wird die *Anweisung* 1 ausgeführt. Hat hingegen `e1` den Wert `false`, so wird *Anweisung* 1 **nicht** ausgeführt, eventuell aber *Anweisung* 2, soferne eine `else`-Sequenz angegeben wurde.

Schachtelungen sind möglich:

```
if (e1)
    if (e2)
        Anweisung 1
    ( else                          ) optional
          Anweisung 2
( else                              ) optional
      Anweisung 3
```

2.3 Die `else if` Verzweigung (Mehrfachverzweigung)

Sie hat die folgende Form:

```
if (e1)
    Anweisung 1
else if (e2)
    Anweisung 2
else if (e3)
    Anweisung 3
( else                              ) optional
      Anweisung 4
```

2.4 Die `switch`-Anweisung

Dies ist eine spezielle Form der Mehrfachverzweigung. In dieser Anweisung wird überprüft, ob ein Ausdruck einen Wert aus einer Liste **konstanter**

Werte entspricht. Es erfolgt dann eine entsprechende Verzweigung.

```
switch (e1) {
case a :
      Anweisungen 1
      (break;)                          optional
( case b :                              )
( case c :                              )
(      Anweisungen 2                    ) optional
( case d :                              )
(      Anweisungen 3                    )
(      (break;)          optional       )
default :
      Anweisungen 4
      break;
}
```

Es wird hier überprüft ob der Ausdruck `e1` den Wert der Konstanten `a` hat (eine Konstante, also ein Buchstabe oder eine ganze Zahl, kein Textstring!). Trifft dies zu, so werden die *Anweisungen* 1 ausgeführt. Folgt auf *Anweisungen* 1 die Anweisung `break;`, so wird die `switch`-Anweisung verlassen. Fehlt `break;`, so wird mit *Anweisungen* 2 fortgesetzt. Hat `e1` hingegen den Wert der Konstanten `b` oder `c`, so werden die *Anweisungen* 2 und *Anweisungen* 3 ausgeführt und dann die `switch`-Anweisung verlassen, wenn auf *Anweisungen* 3 `break;` folgt. Hat hingegen `e1` den Wert `d`, so werden nur die *Anweisungen* 3 ausgeführt. Kann von `e1` keiner der vier Fälle erfüllt werden, so werden die *Anweisungen* 4 ausgeführt. Die *Anweisungen* müssen dabei nicht durch { } geblockt werden, was Sie aber nicht daran hindert es trotzdem zu tun. Sollen auch Variablen-Definitionen enthalten sein, so ist eine Blockung **unbedingt** vorzusehen. Auf jeden Fall steht die `break;`-Anweisung **außerhalb** des Blockes.

Es ist gute Praxis stets den `default`-Teil zu verwenden, da dies hilft Fälle zu erkennen, in denen die anderen Bedingungen nicht erfüllbar sind. (Zumeist Fehler im Programm oder in der Analyse des Problems!) Dieser Teil wird dann nicht eingebaut werden, wenn bewußt nur ganz bestimmte Fälle durch die `switch`-Anweisung behandelt werden sollen. In diesem Fall kann man auch einen 'leeren' `default`-Teil vorsehen, welcher nur `break;` enthält.

2.5 Schleifen

Schleifen dienen zum wiederholten Durchführen einer Anweisung bis eine spezielle Abbruchbedingung erfüllt ist.

2.5.1 Die `while`-Anweisung

Sie hat die Form:

```
while (e1)
    Anweisung
```

Solange der logische Ausdruck `e1` den Wert `true` hat, wird die *Anweisung* durchgeführt. Dies führt zur 'berühmten' Endlosschleife:

```
while (1)
    Anweisung
```

in welcher die *Anweisung* ununterbrochen wiederholt wird. (Einzige Rettung: Programmabbruch durch Eingabe von `^C`!)

Hat hingegen `e1` den Wert `false`, so wird die *Anweisung* nicht ausgeführt.

2.5.2 Die `for`-Anweisung

Sie hat die Form:

```
for (e1; e2; e3)
    Anweisung
```

Diese Anweisung entspricht auch:

```
e1;
while (e2) {
    Anweisung
    e3;
}
```

Es wird also zuerst die Anweisung `e1;` ausgeführt, dann wird der logische Ausdruck `e2` evaluiert und solange dieser den Wert `true` hat, wird die *Anweisung* und danach die Anweisung `e3;` ausgeführt. Etwa:

```
int i;
for (i=0; i<5; i++)
    printf("i = %d\n",i);
```

gibt die Zahlen 0 bis 4 am Bildschirm aus.

2.5.3 Die `do - while`-Anweisung

In den beiden zuvor besprochenen Anweisungen wurde die Bedingung zur Ausführung der Schleife **vor** dem Eintritt in die Schleife überprüft. Bei der nun zu besprechenden Anweisung wird die Bedingung zur **weiteren** Ausführung der Schleife erst am **Ende** der Schleife überprüft. Die Schleife wird also **mindestens** einmal durchlaufen. Syntax:

```
do
    Anweisung
while (e1);
```

Es wird also zuerst *Anweisung* ausgeführt und danach der Wert des logischen Ausdruckes `e1` bestimmt. Hat dieser den Wert `true`, so wird *Anweisung* wiederum ausgeführt mit nachfolgender Überprüfung des Wertes von `e1`. Nimmt `e1` den Wert `false` an, wird die Schleife verlassen.

2.5.4 Die `break`-Anweisung

Diese Anweisung erlaubt das Verlassen einer Schleife, wenn sich beim Abarbeiten der *Anweisung* herausstellt, daß ein Verbleib in der Schleife sinnlos ist. Dies erlaubt eine Neuformulierung der `do - while` Anweisung unter Verwendung der `while` Anweisung:

```
while (1) {
    Anweisung
    if (e1) break;
}
```

oder unter Verwendung der `for`-Anweisung:

```
for (;;) {
    Anweisung
    if (e1) break;
}
```

In beiden Fällen wurde eine Endlosschleife programmiert. Im Zuge der Berechnungen ergibt es sich aber, daß eine Abbruchbedingung, der Ausdruck `e1`, den Wert `true` annehmen kann, worauf die Schleife mit Hilfe der `break`-Anweisung verlassen werden kann.

2.5.5 Die `continue`-Anweisung

Diese Anwesiung führt dazu, daß die nächste Iteration begonnen wird, daß also alle nach `continue;` im Block enthaltenen Anweisungen übersprungen werden.

```
int i,N=20;
for (i=0; i<N; i++) {
     .
     .
     if (!(i%2)) continue;
     .
     .     /* exekutiere nur für gerade i */
     .
}
```

2.5.6 Die goto-Anweisung, Marken

Die goto-Anweisung führt zur Verzweigung des Programmes an eine Stelle, welche durch eine Marke (Label) gekennzeichnet ist. Es gibt relativ wenig Situationen, welche eine goto-Anweisung notwendig machen, Puristen verlangten sogar, diese Anweisung überhaupt zu eliminieren. Eine typische Situation, in welcher eine goto-Anweisung die Lesbarkeit des Programmes stark verbessert ist sicher dann gegeben, wenn Fehlermitteilungen abgearbeitet werden sollen. Hier ein Beispiel:

```
int main()
{
     int e_flag=0;
          .
          .
     if (e1) {
          e_flag = 1;     /* Fehler 1*/
          goto error;
     }
          .
          .
     if (e2) {
          e_flag = 2;     /* Fehler 2*/
          goto error;
     }
          .
          .
     return 1;

error:           /* Marke */
     switch (e_flag) {
     case 1:
          Fehlermitteilung 1
          break;
     case 2:
          Fehlermitteilung 2
          break;
          .
          .
     default:
          break;
     return 0;
}
```

Wie wir aus dem Beispiel ersehen ist eine Marke eine Variable, welche durch einen Variablen-Namen gekennzeichnet ist, und welche durch den Postfix-Operator : in eine Marke umfunktioniert wird. Eine solche Marke muß nicht deklariert werden, aber, wie bereits gesagt ist eine Marke eine Variable und ist als solche nur **lokal** definiert.

2.5.7 Der ,(Komma)-Operator

Er verknüpft zwei Ausdrücke, und hat die Form:

```
e1, e2
```

Ein Paar von Ausdrücken `e1` und `e2`, welche durch ein `,` voneinander getrennt sind, werden von links nach rechts ausgewertet. Die Kombination `e1, e2` ist wiederum ein Ausdruck, welcher den Wert `e2` annimmt. Man kann natürlich auch mehr als zwei Ausdrücke durch den `,`-Operator getrennt auswerten, also `e1, e2, e3, ....`

Beispiel:

```
int c,t;
c = (t = 3, t+5);
```

dies entspricht den Anweisungen:

```
int t,c;
t = 3;
c = t+5;
```

Der `,`-Operator hat die niedrigste Priorität, er ist daher fast immer mit Klammerung zu verwenden um das gewünschte Ergebnis zu erzielen. Die größte Bedeutung hat der `,`-Operator im Zusammenhang mit der `for`-Anweisung.

Programm 5:

Schreiben Sie ein kleines Programm, welches obiges Beispiel für den `,`-Operator überprüft. Welches Ergebnis ist zu erwarten, wenn keine Klammerung vorgesehen wird? Wenn Sie nach obigem Beispiel die Anweisung `a = t = 3, f = a+5;` ausführen, welches Ergebnis ist zu erwarten? Warum? Weiters soll dieses Programm eine Schleife enthalten, welche die Zahlen 1 bis 10 ausdruckt, aber zu jeder Zahl auch angibt, ob diese Zahl gerade oder ungerade ist. Es ist dabei dasselbe Problem mit der `while`-, der `for` und der `do - while`-Anweisung zu lösen. Schließlich ist noch eine Schleife zu programmieren, in welcher nur die geraden (ungeraden) Zahlen zwischen 1 und 10 ausgedruckt werden. (Auch hier gibt es verschiedene Möglichkeiten, versuchen Sie bitte mindestens zwei ...)

Programm 6:

Es sollen wieder Spin-Zustände gesetzt und ausgelesen werden. Hiebei sind aber nun Schleifen einzusetzen. Die Zahl der Zustände in der Spinkette wird durch `const unsigned int NUM_STATES = 6;` gegeben. Es sind dann etwa alle Spins an geraden Positionen in den 'up'-Zustand zu versetzen, oder alle Spins an ungeraden Positionen. Die Spinzustände in der Kette sind in der folgenden Form am Bildschirm auszugeben:

`Spin-Zustand = |010101>`

wenn etwa alle geraden Spins gesetzt sind. Beachten Sie bitte, daß in diesem Zusammenhang die Null als gerade Zahl gilt!

2.5.8 Abschließende Bemerkung

Schleifen und auch `if`-Anweisungen sind in beliebiger Tiefe schachtelbar. Um Anweisungen eindeutig zu machen, muß auf korrekte Blockung geachtet werden!

```
if (e1)                    if (e1) {
     if (e2)                    if (e2)
          Anweisung 1                Anweisung 1
     else                       } else
          Anweisung 2                Anweisung 2
```

Während im linken Beispiel die *Anweisung* 2 nur ausgeführt wird, wenn `e1 true` und `e2 false` ergibt, wird im rechten Beispiel *Anweisung* 2 nur ausgeführt, wenn `e1` den Wert `false` hat!

> **Regel:** Es ist besser (und leichter lesbar) durch 'zuviel' Blockung das Programm eindeutig zu gestalten, als sich auf die impliziten Präzedenzen zu verlassen.

Kapitel 3

Programmstruktur und Funktionen

3.1 Allgemeines

Die Programmstruktur eines C-Programmes zerfällt im allgemeinen in den Hauptteil `main` und eine mehr oder minder große Zahl von Funktionen, wobei der Hauptteil `main` selbst von der Struktur her eine Funktion darstellt, welche allerdings mit dem Betriebssystem kommuniziert.

Es ist ganz typisch, daß ein C-Programm in eine große Zahl von Einzelschritten zerlegt wird, welche innerhalb von Funktionen abgearbeitet werden, welche selbst wiederum in möglichst allgemeiner Art geschrieben werden, um ihre Wiederverwendung sicherzustellen. Zusätzlich verwendet C noch eine enorme Zahl von 'Library-Funktionen', also Funktionen, welche vom verwendeten Compiler bzw. Betriebssystem abhängen. Ein Teil dieser Funktionen sind im ANSI-Standard normiert.

3.2 Die Funktion

Die Funktion ist das dominierende Strukturelement eines C-Programmes. Die grundsätzliche **Funktions-Definition** ist wie folgt gegeben:

```
ftyp function_name(vtyp p1, vtyp p2, ...)
{
     ftyp value;

     Anweisungen
     return value;     /* Rückgabe des Funktionswertes */
}
```

Der in den geschlungenen Klammern eingeschlossene Teil wird dabei der **Funktions-Körper** genannt.

Eine Funktion beschreibt die funktionale Abhängigkeit des Funktionswertes von den Funktionsparametern. Die Funktion ist dabei vom Variablentyp `ftyp`, das bedeutet, daß die Funktion einen Wert vom Variablentyp `ftyp` an den aufrufenden Programmteil zurückgibt. Unterbleibt die Angabe von `ftyp`, so wird angenommen, daß der Funktionswert vom Typ `int` ist. Der Funktionswert wird durch die `return`-Anweisung an das aufrufende Programm übergeben. Die Funktionsparameter sind dabei `p1`, `p2`, usw. und sie sind vom Variablentyp `vtyp`. Der **Funktions-Name** `function_name` ist dabei ein gültiger Variablen-Name.

Der Funktionstyp `void` bedeutet insbesonders, daß die Funktion keinen Wert zurückgibt. Dies bedeutet, daß entweder die `return`-Anweisung überhaupt unterbleibt, oder daß die 'leere' `return;`-Anweisung zu verwenden ist.

Eine Funktions-Defintion darf nicht innerhalb eines Funktions-Körpers liegen!

Eine Funktions-Definition wird von einer (vorangestellten) **Funktions-Deklaration** begleitet. Sie hat folgende Form:

```
ftyp function_name(vtyp,vtyp,...);
```

Die Funktions-Deklaration ermöglicht die Überprüfung des formal korrekten Funktionsaufrufes durch den Compiler. Bei fehlerhaftem Aufruf, werden vom Compiler entsprechende Fehlermitteilungen ausgegeben.

Jede Funktion hat eine bestimmte Anzahl (auch Null) Parameter `p1`, `p2`, ..., welche wiederum von einem bestimmten Datentyp sind. Es ist wichtig, daß Parameter als Ausdrücke zu verstehen sind, deren 'Wert' an die Funktion übergeben werden. Dazu wird für jeden Parameter (implizit) eine Variable desselben Namens angelegt, welche mit dem übergebenen Wert initialisiert wird. Es geschieht also folgendes:

```
double fu(int a, double b)
{
 /*     int a = a;      */
 /*     double b = b;       */
     double value;
     a += 2;
     return b * a;
}
```

wird vom Compiler angelegt

Konsequenz: Wird eine Variable als Parameter übergeben, so ist ihr Wert innerhalb der aufgerufenen Funktion (von 'außen' her gesehen) nicht veränderbar! Andererseits kann ein *beliebiger*, gültiger Ausdruck als Parameter eingesetzt werden, dessen Wert (entsprechend des Variablentyps des Parameters umgewandelt) an die Funktion übergeben wird.

Die Parameter spielen die Rolle einer lokalen Variablen.

Funktionen können *rekursiv* verwendet werden, also sich selbst wiederum aufrufen. Ganz wesentlich bei dieser Art der Anwendung ist der Einbau einer Abbruchbedingung, da sonst eine schwer zu durchbrechende Schleife entsteht.

Die einfachste ('leere') Funktion hat folgende Deklaration und Definition:

```
void dummy(void);     /* Deklaration */
     .
     .
void dummy() {}      /* Definition */
```

Jede Funktion, welche nicht vom Typ `void` ist, **muß** zumindest eine `return`-Anweisung enthalten, welche **unbedingt** erreicht werden kann. Eine solche Anweisung ist von der Form:

`return` *Ausdruck*;

Der Wert, welcher vom *Ausdruck* angenommen wird, wird - entsprechend typumgewandelt - zum Funktionswert gemacht.

Sehen wir unser obiges Beispiel an, so ergibt:

```
double c;
c = fu(2,3.0);
```

den Wert 12.0 für die Variable `c`. `fu(2,3.0)` ist dabei ein Ausdruck vom Wert 12.0.

Jetzt verstehen wir auch die auf Seite 3 eingeführte Deklaration: `const double Pi = 4.0*atan(1.0);` Es besteht offensichtlich eine 'Library-Funktion'

```
double atan(double);
```

welche den Arcustangens berechnet. Nachdem der $\arctan(1.0) = \pi/4$ ist, ergibt dann die Multiplikation mit 4.0 den Wert π für die unveränderliche Variable `Pi`.

Programm 1:

Ergänzen Sie das erste Programm mit obiger Definition für `Pi` und überprüfen Sie das Ergebnis. Unterhalb des Befehls: `#include <stdio.h>` fügen Sie bitte den Befehl `#include <math.h>` ein.

Programm 7:

Zerlegen Sie die Aufgaben des Programmes 6 in Funktionen. Es sind drei Unterprogramme zu schreiben:

(a) Setzen der Spins auf geraden Gitterplätzen.

(b) Setzen der Spins auf ungeraden Gitterplätzen.

(c) Ausgabe des Spinzustandes.

(d) Optional: Funktion zum Setzen ('up') des Spins an einer bestimmten Stelle der Spinkette.

(e) Optional: Funktion zum Löschen ('down') des Spins an einer bestimmten Stelle der Spinkette.

Auf eine möglichst allgemeine Definition der Funktionen ist zu achten. Das heißt, daß die Größe der Spinkette nicht von vornherein festgelegt ist. Es muß daher auch überprüft werden, ob die Zahl der Gitterplätze überhaupt sinnvoll bearbeitet werden kann. Es ist dabei davon auszugehen, daß die Spinkette wiederum in einer Variablen vom Typ `unsigned int` abzuspeichern ist. Fehlermeldungen bei fehlerhaftem Aufruf! Test der Funktionen vom Programm `main` aus mit Fehlerbehandlung im Stile von Seite 15.

Damit sind die grundlegenden Definitionen von Funktionen eingeführt. Mit den sich erweiternden Kenntnissen werden in den folgenden Kapiteln bestimmte Details erneut, genauer, zu diskutieren sein. An den hier vorgestellten grundsätzlichen Vereinbarungen wird sich aber nichts mehr ändern.

3.3 Wichtige 'Library'-Funktionen

Hier sollen die wichtigsten Funktionen der Mathematik-Funktionsbibliothek angeführt werden. Sie sind im allgemeinen wie folgt deklariert:

```
double fu(double x);
```

Die Funktions-Definitionen kann man über den Befehl `#include <math.h>` in das Programm einbinden (siehe Seite 26). `fu` steht dabei für:

- Trigonometrische Funktionen:

  `sin`, `cos`, `tan`
  `asin`, `acos`, `atan`

  weiters gibt es noch

  ```
  double atan2(double y, double x);
  ```

  welche den $\tan(y/x)$ im korrekten Quadranten berechnet. Winkel-Argumente sind in allen Fällen in `rad` anzugeben. Funktionswerte, welche Winkeln entsprechen, werden in `rad` zurückgegeben.

- Hyperbolische Funktionen:

  `sinh`, `cosh`, `tanh`
  `asinh`, `acosh`, `atanh`

- Logarithmen und Exponentialfunktion:

  `log`, `log10`, `exp`, `pow(double x, double y)`

  Die Funktion `pow(x,y)` berechnet x^y.

- Besselfunktionen:

  `j0`, `j1`, `jn(int n, double x)`
  `y0`, `y1`, `yn(int n, double x)`

  `j0(x)` berechnet die Besselfunktion erster Art von der Ordnung 0 für das Argument `x`. `j1(x)` macht dasselbe für die Besselfunktion erster Art und der Ordnung 1. `jn(n,x)` berechnet die Besslfunktion erster Art und der Ordnung `n` für das Argument `x`. Die `y`-Funktionen machen dasselbe für Besselfunktionen der zweiten Art.

- Die Fehlerfunktion `erf(x)` berechnet:

  $$\frac{2}{\sqrt{\pi}} \int_0^x dt\, e^{-t^2}$$

  und der Aufruf `erfc(x)` entspricht `1.0-erf(x)`.

- Verschiedenes:

```
int abs(int);          /* Absolutbetrag integer */
double fabs(double);   /* Absolutbetrag Fließkommazahl */
double ceil(double);   /* Aufrunden nächste ganze Zahl */
double floor(double);  /* Abrunden nächste ganze Zahl */
double sqrt(double);   /* Quadratwurzel */
void srand(int);       /* Setzen der Seedzahl für */
                       /* Zufallszahlen */
int rand(void);        /* Pseudozufallszahl */
```

und so gibt es, abhängig von der Implementierung noch einige hundert (oder noch mehr) Funktionen, welche man in sein Programm einbauen kann. Man sollte sich allerdings auf ANSI-Standard Funktionen beschränken, wenn das Programm auf unterschiedlichen Bertriebssystemen laufen soll. Soll das Programm nur unter UNIX laufen, so kann man noch vieles mehr verwenden.

Die Funktionsbeschreibungen kann man stets über den UNIX-Befehl:

```
%   man sin
```

aufrufen. Hier wird zum Beispiel die 'Manualpage' für die `sin`-Funktion aufgerufen. Unter einem LINUX-System kann man zum Beispiel auch mit

```
%  xman &
```

ein Hilfsprogramm starten, welches es erlaubt in den 'Manualpages' zu blättern. Man konzentriert sich dabei auf die Section (3) Subroutines. Hier sind alle vom System unterstützten Funktionen in ihrer Funktion und Anwendung beschrieben.

3.4 Der Gültigkeitsbereich von Variablen

Grundsätzlich beschränkt sich der Gültigkeitsbereich einer Variablen auf jenen Block, innerhalb welchem sie deklariert wurde. Wird der Block verlassen, so geht die Variable unbedingt zusammen mit ihrem Inhalt verloren.

Diese Regel mußte aber bereits mit der Einführung von Funktionen, welche ja nur eine andere Art von Variablen sind, durchbrochen werden, da ja die Funktions-Definition außerhalb eines Blockes erfolgt, aber sinnvoller Weise innerhalb von Blöcken 'bekannt' sein muß.

3.4.1 Externe Variable

Mit der Funktion wurde die erste **externe** Variable eingeführt (ohne sie explizite so zu nennen). Sie wird außerhalb von Blöcken definiert, ist aber innerhalb von Blöcken bekannt. Wir können verallgemeinern:

> Wird eine Variable außerhalb eines Blockes deklariert, so ist sie programmweit ab der Deklaration bekannt. Wird auf eine solche externe Variable innerhalb eines Blockes zugegriffen, so sollte man sie in diesem Block als `extern` deklarieren, um die Lesbarkeit des Programmes zu erhöhen.

Beispiel:

```
double foo(double);
int main()
{
     extern double x_data;  /* führt zu Fehler beim Link */
     .
     .
     return 1;
}
```

↑ `x_data` unbekannt

```
double x_data;
```

↓ `x_data` bekannt

```
double foo(double x)
{
     extern double x_data;
     .
     .
     x_data = ...;
     return ...;
}
```

3.4.2 Statische Variable

Statische Variable führt man über die Deklaration, etwa `static double a;` ein, indem man das Schlüsselwort `static` **vor** den Variablen-Typ setzt.

Deklariert man eine Variable innerhalb eines Blockes als `static`, so behält sie ihren letzten Wert auch nach Verlassen des Blockes bei. Sie ist zwar außerhalb des Blockes unbekannt, sie verliert aber ihren Wert nicht und dieser

steht nach Wiedereintritt in den Block zur Verfügung.

Beispiel:

```
void foo(int);
.
.
void foo(int a)
{
     static int init=1;

     if (init) {          /* Initialisierung, init = 1 */
          .
          .               /* Initialisierung */
          .
          init = 0;
     }
     .
     .
}
```

Wird hingegen eine Variable außerhalb eines Blockes als `static` deklariert, so ist sie nur innerhalb des speziellen Programmteiles bekannt, innerhalb welchem sie definiert wurde. Auch Funktionen können als `static` deklariert werden, wenn sie nur von 'lokaler' Bedeutung sind.

3.4.3 Rekursivität

Aufgrund der Definition von Funktionen außerhalb von Blöcken, sind sie programmweit bekannt und sind daher definitionsgemäß *rekursiv*.

Beim rekursiven Aufruf von Funktionen wird jedesmal ein neuer Satz von lokalen Variablen generiert, daher ist hier eine gewisse Vorsicht angebracht. Im allgemeinen ist rekursiver Code nicht schneller als konventionell geschriebener, er ist aber sicherlich sehr kompakt - was aber nicht notwendiger Weise die Lesbarkeit verbessert.

Ganz besonderes Augenmerk ist auf eine korrekte Formulierung der Abbruchbedingung zu legen, da man bei rekursiver Programmierung sehr leicht eine Endlosschleife produzieren kann.

Programm 8:

Rekursive Programmierung von $n!$.

3.5 Der C-Präprozessor

Die Anwendung des Präprozessors ist der erste Schritt im Ablauf der Programmkonvertierung durch den C-Compiler. Wir wurden bereits bei den bisherigen Programmbeispielen (unbewußt) mit diesem Element des C-Kompilers konfrontiert.

3.5.1 Einbinden von Files

Hiezu steht folgender Präprozessor-Befehl zur Verfügung:

```
#include <filename>
#include "filename"
```

Durch diese Befehle wird ein File, welcher C Anweisungen und/oder Defintionen enthält eingelesen und vom Compiler als Teil des zu bearbeitenden Programmes interpretiert. Der File wird dabei durch `filename` bezeichnet, wobei `filename` ein gültiger Filename sein muß, welcher vom Betriebssystem auch entsprechend interpretiert werden kann. In der ersten Form des Befehls, wird der File dort gesucht, wo es die Implementierung des C-Compilers vorschreibt. Der bereits bekannte Befehl

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

veranlaßt den Präprozessor den File `stdio.h` einzulesen, wobei angenommen wird, daß dieser File im Implementierungsbereich des Compilers aufzufinden ist. Dieser File ist ein **Header-File** (dewegen die Extension `.h`) und enthält alle notwendigen Funktions-Definitionen des Standard Input/Output Interfaces. Ein anderer Header-File ist `math.h`, welcher die Funktions-Definitionen der Mathematik-Library enthalten. Die Manual-Seiten geben Ihnen stets an, welcher Header-File zu inkludieren ist, wenn Sie eine bestimmte Library-Funktion nutzen möchten.

In der zweiten Form des Befehls wird der File des Namens `filename` zuerst im aktiven Verzeichnis gesucht. Dies ist jenes Verzeichnis, in welchem auch das zu bearbeitende Programm abgespeichert ist. Enthält `filename` auch eine Pfadangabe, so wird in dem durch den Pfad definierten Verzeichnis nach dem angegebenen File gesucht.

Durch `#include` angeschlossene Files, können selbst wieder `#include` Befehle enthalten.

3.5.2 Makro-Substitution

Ein Makro wird durch folgenden Befehl definiert:

```
#define name substitut
```

wie etwa in:

```
#define BUFSIZE 500
```

Dieser Makro führt dazu, daß vom Präprozessor überall dort, wo im Programm `BUFSIZE` auftritt, das Substitut `500` eingesetzt wird. Also:

```
#define BUFSIZE 500
     .
     .      for (i=0; i<BUFSIZE; i++)
```

verwandelt sich nach Durchlauf des Prprozessors in:

```
#define BUFSIZE 500
     .
     .      for (i=0; i<500; i++)
```

Man kann auch folgende Definition eines Macros zum Auffinden des Maximums verwenden:

```
#define MAX(A,B) ((A) > (B) ? (A) : (B))
```

was zum Beispiel aus

```
z = MAX(x+1,y+1);
```

die Anweisung

```
z = ((x+1) > (y+1) ? (x+1) : (y+1));
```

macht. MAX ist aber keine Funktion, daher sind solche Vereinbarungen, so praktisch sie auf den ersten Blick auch aussehen, nur mit großer Vorsicht zu verwenden.

Ein einmal definierter Makro kann mit der Anweisung

```
#undef name
```

aufgehoben werden.

3.5.3 Bedingungen

Es gibt hiezu die Präprozessorbefehle:

```
#if
#elif /* entspricht else if */
#else
#endif
#ifdef name /* ist name definiert? */
#ifndef name /* ist name nicht deefiniert? */
```

Ist man sich, zum Beispiel, nicht sicher, ob bereits der Makro `BUFSIZE` definiert wurde, so kann man wie folgt vorgehen:

```
#ifndef BUFSIZE
#define BUFSIZE 500
#endif
```

oder

```
#if !defined(BUFSIZE)
#define BUFSIZE 500
#endif
```

Für ein systemunabhängiges Programm könnte man zum Beispiel systemabhängige Programmelemente wie folgt einbauen:

```
#ifdef SYSV /* code nur für UNIX System V */
     .
     .
#endif
#ifdef MSDOS /* code für MS-DOS */
     .
     .
#endif
```

wobei angenommen wird, daß innerhalb des C-Compilers `SYSV` definiert ist, wenn man unter dem UNIX System V arbeitet, oder `MSDOS`, wenn unter DOS gearbeitet wird. Die aktuell von den C-Compilern gesetzten Definitionen sind aus den jeweiligen Dokumentationen zu erfahren.

Ganz typisch für Header-Files ist folgende Befehls-Sequenz:

```
#ifndef __HEADER_H
#define __HEADER_H
     .
     .              /* Deklarationen */
     .
#endif
```

Dies verhindert, daß der Headerfile `header.h` bei mehrfachem `#include "header.h"` Präprozessorbefehl auch mehrfach eingelesen wird.

Programm 9:

Zerlegen Sie das Programm 7 in einen File, welcher nur das Hauptprogramm enthält und einen File, welcher nur die Funktionen zur Manipulation von Spins enthält. Plazieren Sie die die Funktions-Deklarationen in einen Header-File und testen Sie das Programm.

Es gibt noch eine ganze Anzahl weiterer Präprozessorbefehle, welche auch zum Teil vom verwendeten C-Compiler abhängig sind. Es wurde hier ganz bewußt nur jenes Minimum angeführt, welches im normalen Programmieralltag unverzichtbar ist.

Kapitel 4

Felder und Pointer

4.1 Felder (Arrays)

Ein Feld ist die geordnete Aneinanderreihung von Daten unter einem Variablen-Namen. Man deklariert ein Feld mit Hilfe der folgenden Variablen-Deklaration: `vtyp var_name[l1][l2]...[ln];` wobei `vtyp` der Variablen-Typ ist und `l1`, `l2`, ..., `ln` die Zahl der Elemente in der jeweiligen Dimension angeben. So bedeutet etwa `int x[20][30];` eine zweidimensionale Matrix vom Variablen-Typ `int` der Dimension 20×30.

Will man ein bestimmtes Element aus dieser Matrix herausgreifen, so bedient man sich folgender Wertzuweisung:

```
int i,j,x[20][30],y;

i = 5;
j = 6;
y = x[i][j];
x[i][j] = 20;
```

Damit wird das siebente Element der sechsten Reihe dews Feldes `x` herausgegriffen.

Elemente eines Feldes werden also durch den Postfix-Operator

`[n]`

herausgegriffen, wobei soviele Operatoren auf den Variablennamen anzuwenden sind, als es der Dimension des Feldes entspricht. `n` ist dabei der Index des Elementes.

> Indizes beginnen stets mit dem Wert **Null**. Es bsteht weiters die Regel, daß bei höherdimensionalen Feldern der rechte Index schneller 'läuft'.

Aus dieser Regel ergibt sich für die Initialisierung eines zweidimensionalen Feldes in der Variablen-Deklaration:

```
int x[2][4] = {{ 1, 2, 3, 4},
               { 5, 6, 7, 8}};
```

Es handelt sich hier um eine zweizeilige Matrix mit vier Kolonnen. Sie wird wie folgt initialisiert:

$$(x)_{ij} = \begin{pmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \end{pmatrix}.$$

4.2 Pointer (Zeiger)

Ein Pointer ist eine Variable, welche die Memory-Adresse einer Variablen als Wert enthält. Pointer und Felder sind einander sehr nahe verwandt, als der Variablen-Name eines Feldes stets ein Pointer (entsprechenden Typs) aufgefaßt wird, welcher als Wert die Memory-Adresse des Elementes `[0][0]...[0]` des Feldes enthält.

Die Deklaration von Pointervariablen erfolgt durch Vergabe eines Variablen-Namens, welchem das Zeichen `*` vorangestellt wird:

```
int *pi,*pj,a;
double *pda, df;
```

Hier sind `pi` und `pj` Pointervariable vom Typ `int`, während `a` eine Variable vom Typ `int` ist. Weiters ist `pda` eine Pointervariable vom Typ `double`, ihr Wert zeigt also auf eine Variable vom Typ `double`.

Programm 1:

Erweitern Sie das Programm um die Befehle:

```
printf("int *\t\t%d Bytes\n",sizeof(int *));
printf("int *\t\t%d Bytes\n",sizeof(double *));
```

Diskutieren Sie das Ergebnis!

4.3 Der &(Address)-Operator

Dies ist ein unärer Operator, welcher die Memory-Adresse des Operanden ermittelt:

&lvalue

Dieser Ausdruck hat den Wert der Memory-Adresse von *lvalue*, also in der Regel einer Variablen, welche aber keine `register`-Variable sein darf.

```
int *pi,a;
pi = &a;
```

Der Wert, welcher in der Pointer-Variablen `pi` abgespeichert wird, ist die Memory-Adresse der Variablen `a`.

4.4 Der *(Dereference)-Operator

Dies ist wiederum ein unärer Operator, welche nur auf eine Pointervariable angewendet werden kann:

**operand*

Dieser Operator erlaubt es auf den Inhalt jenes Objektes zuzugreifen, auf welches die Pointervariable *operand* zeigt.

So bedeutet:

```
int *pi,a,y,z[20];
pi = &a;            /* pi zeigt nun auf a */
a = 3;              /* a erhält den Wert 3 */
*pi = 3;            /* wiederum wird a der Wert 3 zugewiesen */
y = *ip;            /* y wird der Wert von a zugewiesen */
y = a;              /* y wird der Wert von a zugewiesen */
ip = z;             /* ip zeigt nun auf z[0] */
ip = &z[0];         /* ip zeigt auf z[0] */
ip++;               /* ip zeigt nun auf z[1] */
ip += 2;            /* ip zeigt nun auf z[3] */
--ip;               /* ip zeigt nun auf z[2] */
y = *(z+2);         /* y wird der Wert von z[2] zugewiesen */
*(z+12) = 0;        /* ident zu z[12] = 0; */
```

Die Operatoren `*` und `&` haben eine gegenüber den arithmetischen Operatoren höhere Präzedenz.

4.5 Zeichenketten (Strings)

An Zeichenketten kann man am besten das Spiel zwischen Feldern und Pointern studieren und lernen; deshalb sollen an dieser Stelle Zeichenketten ausführlich besprochen werden.

Wir haben bereits zu Beginn auf Seite 3 die Zeichenkette als Konstante kennengelernt. Die Zeichenkette ist nichts anderes als ein Feld vom Variablen-Typ `char`. Die Variablen-Deklaration

```
char text[] = "Zeichenkette";
```

definiert ein eindimensionales Feld von Zeichen, welche aus so vielen Elementen besteht, daß die Textkonstante `"Zeichenkette"` mit dem abschließenden `'\0'` Zeichen darin gespeichert werden können. Ident wäre folgende Variablen-Deklaration:

```
char *ptext = "Zeichenkette";
```

`ptext` ist hier eine Pointervariable vom Typ `char`, welche auf jenen Memorybereich zeigt, in welchem das erste Zeichen des Strings `"Zeichenkette"` vom Compiler abgespeichert wurde.

Es wurde bereits darauf hingewiesen, daß der Variablen-Name `text`, welcher das Zeichenfeld bezeichnet, ebenfalls als Pointer aufzufassen ist, welcher auf jenen Memorybereich weist, welcher dem ersten Feld des Zeichenfeldes entspricht. Es ist also folgendes möglich:

```
char text = "Zeichenkette";
char *ptext;
.
.
ptext = text;                /* ptext zeigt auf den */
                             /* String text */
```

Wir wollen nun die Arbeit mit solchen eindimensionalen Feldern an einem Beispiel erörtern: wir wollen eine Funktion schreiben, welche eine Zeichenkette von einem Memorybereich auf einen anderen kopiert, die Funktion `strcpy`:

```
void strcpy(char[],char[]);

void strcpy(char t[], char s[])
{     /* t ... target */
       /* s ... source */
     register int i;
     for (i=0; (t[i] = s[i]) != '\0'; i++);
}

int main()
{
     char text[] = "Test-String",copy[50];

     strcpy(copy,text);
     printf("Kopie = %s\n",copy);
}
```

Hier wurde nur die Feldeigenschaft ausgenützt und die Tatsache, daß ein String stets mit `'\0'` abgeschlossen wird. Die spezielle Form der Formulierung der Abbruchbedingung stellt sicher, daß auch `'\0'` vom Feld `s` in das Feld `t` übertragen wird.

Es fällt des weiteren auf, daß offensichtlich Felder auch Parameter von Funktionen sein können, und daß bei eindimensionalen Feldern offensichtlich ein Angabe der Dimension des Feldes unterbleiben kann. Schließlich scheint das Programm anzunehmen, daß die vorgenommenen Veränderungen auch nach 'außen' sichtbar werden, da ja sonst das Programm wohl nicht sinnvoll wäre. Worauf stützt sich diese Annahme? Sie beruht darauf, daß die beiden Parameter `t` und `s` ja eigentlich Pointervariable sind, welche jeweils auf das erste Element der entsprechenden Felder zeigen. Parameter werden nun, wie bereits besprochen, Unterprogrammen *wertmäßig* übergeben. In diesem Fall legt nun der Compiler zwei lokale Pointervariable vom Typ `char` an, welche den Wert erhalten, der den beiden Parametern beim Aufruf übergeben werden, also Memory-Adresse des ersten Elementes des Feldes `text` nach `s` und die Memoryadresse des Feldes `copy` nach `t`. Diese Werte sind nach außen innerhalb der Funktion `strcpy` auch nicht veränderbar, wohl aber die Inhalte der Memorybereiche, auf welche diese Pointer verweisen.

Wir wollen nun explizite der Tatsache Rechnung tragen, daß `s` und `t` eigentlich Pointervariable sind und schreiben:

```
void strcpy(char*,char*);

void strcpy(char *t, char* s)
{                                              /* t ... target */
                                               /* s ... source */
    register int i;
    for (i=0; (t[i] = s[i]) != '\0'; i++);
}
```

Wir sehen, daß eine Pointervariable auch wie ein eindimensionales Feld behandelt werden kann.

Im nächsten Schritt gehen wir auf reine Pointerschreibweise über:

```
void strcpy(char*,char*);

void strcpy(char *t, char* s)
{                                   /* t ... target */
                                    /* s ... source */
     while ((*t = *s) != '\0') {
          t++;
          s++;
     }
}
```

Wir verwenden hier eine Pointerinkrementierung anstelle eines Index um die Elemente des Feldes `s` nach `t` übertragen zu können. Für eine korrekte Inkrementierung eines Pointers ist sein Typ von entscheidender Bedeutung. Es gilt die Regel:

> Ein Pointer wird stets so inkrementiert/dekrementiert, daß er auf ein gültiges Feldelement verweist.

So wird etwa

```
double x[] = { 1.0, 2.0, 3.0, 4.0},*px;
px = x;
px += 2;
printf("Element: %lf\n",*px);
```

als Ergebnis `Element: 3.0` am Bildschirm ausgeben.

Aus all diesen Überlegungen finden wir schließlich die Endform unserer Funktion `strcpy`:

```
void strcpy(char*,char*);

void strcpy(char *t, char* s)
{                                 /* t ... target */
                                  /* s ... source */
     while (*t++ = *s++);
}
```

in welcher wir noch Operator-Präzendenzen in geeigneter Weise ausnützen.

Zur Behandlung von Zeichenketten gibt es eine umfangreiche Library, auf welche man zugreifen kann. Sie erfordert die Einbindung des Files `string.h`. Auf die Dokumentation kann man mit Hilfe des UNIX-Befehls

```
% man string
```

zugreifen.

Programm 10:

Testen Sie das Programm `strcpy` in seinen unterschiedlichen Varianten. Verwenden Sie eine `#define`-Anweisung um beim Compilieren des Programmes die unterschiedlichen Versionen ansprechen zu können. Verwenden Sie die Funktion `memset` um zuvor das Zielfeld auf Leerzeichen zu setzen. (Verwenden Sie den UNIX-Befehl `man memset` um sich über den Gebrauch von `memset` zu informieren.) Wie können sie dies machen ohne explizite die Feldgröße beim Aufruf von `memset` angeben zu müssen? Kopieren Sie das Feld `text` nach `copy` beginnend bei Feld 3 des Feldes `copy`. Drucken Sie nur das Element 7 des Feldes `copy` aus. Welche Möglichkeiten haben Sie?

Programm 11:

Schreiben Sie Funktionsaufrufe zum Parsen bzw. Ergänzen von Filenamen. (*Parsen* heißt in Sprachelemente zerlegen.) Ein UNIX-Filename ist wie folgt definiert:

```
path/filename.extension
```

Der Pfad selbst kann von der Form `~`, `.`, `..` oder `xxx/yyy/zzz` sein. Er kann aber auch fehlen. `filename` ist der Name des Files (beliebige Zeichen außer `/` und `.`, `extension` ist eine Zusatzbezeichnung, welche den im File enthaltenen Datentyp beschreiben soll. Etwa `.c` für Files, welches ein C-Programm enthalten, oder `.h` für Header-Files, oder `.dat` für allgemeine Datenfiles. Auch `extension` kann fehlen. Schreiben Sie nun Funktionen, welche es erlauben einen String, welcher einen gültigen Filenamen enthalten soll, dahingehend zu untersuchen ob er eine Pfadangabe enthält und wenn ja, wie `path` aussieht, wie `filename` lautet, ob eine `extension` enthalten ist, und wenn ja, wie sie lautet. Sollte der Pfad fehlen, so ist der String mit dem aktuellen Pfad zu ergänzen (Funktion `getcwd`), fehlt die `extension`, so ist der String mit einer vorgegebenen Extension zu erweitern. Verwenden Sie die Funktionsaufrufe zur Manipulation von Strings. Alle diese Funktionen sind in einem getrennten File zu entwickeln.

4.6 Felder von Pointern

Pointer sind ganz normale Variable, also können sie auch in Feldern organisiert werden. Es ist also grundsätzlich auch die folgende Variablen-Deklaration möglich:

```
int *pa[10];
```

Dadurch wird ein Feld deklariert, welches aus 10 Elementen vom Typ Pointer auf `int` besteht.

Typisch ist etwa folgende Vereinbarung für ein Feld aus Strings:

```
static char *mon[] = {"Jänner"," Februar", ..., "Dezember"};
```

welches eigentlich ein Feld von Pointern vom Typ `char` ist. Der Pointer `mon[0]` zeigt dabei auf das `J` von `Jänner`, `mon[1]` auf das `F` von `Februar`, etc.

Wir erkennen, daß eigentlich ein Feld von Pointern ein zweidimensionales Feld abbildet. So wird `mon[0][3]` den Wert `n` haben! Es zeigt also jeder Pointer des Feldes `mon` auf eine **Zeile** dieses zweidimensionalen Feldes. Die einzelnen Zeilen können dabei durchaus von unterschiedlicher Länge sein!

Man kann nun natürlich wiederum einen Pointer definieren, welcher selbst mit seinem Wert auf einen anderen Pointer zeigt. Einen solchen Pointer wird man konsequenter Weise wie folgt deklarieren:

```
char **pmon;
```

Damit wird eine Pointervariable vom Typ Pointer auf den Typ `char` eingeführt. Man kann eine solche Deklaration wie folgt anwenden:

```
char *mon[] = {"Jänner"," Februar", ..., "Dezember", NULL};
char **pmon;
.
pmon = mon;
```

Die Pointervariable `pmon` zeigt nun auf das Element `mon[0]`, da ja `mon` selbst ein Pointer auf das erste Element des Feldes `mon[]` ist. Hier wurde mit `NULL` auch der sogenannte Null-Pointer eingeführt, also ein Pointer vom Wert Null. Man kann obige Deklaration dazu verwenden um in einfacher Weise die im Feld `mon[]` gespeicherten Strings ausdrucken zu können:

```
int i=0;
while (*pmon != NULL)
     printf("%d. Monat = %s\n",i++,*pmon++);
```

Der Null-Pointer `NULL` spielt hier also die Rolle von `'\0'` am Ende eines Strings.

> Die Verwendung von `'\0'` zum Beenden von Strings, bzw. von `NULL` zum Beenden von String-Feldern ermöglicht eine Verarbeitung dieser Felder, ohne daß man zum Zeitpunkt des Programmentwurfes die eigentliche Dimension der Strings oder der String-Felder wissen muß. Dieses Prinzip ist natürlich auch auf andere Anwendungen übertragbar.

4.7 Command-Line Argumente

Bisher wurde das C-Programm einfach nur über seinen Namen aufgerufen. Es wäre aber sinnvoll das Programm durch Angabe von Parametern, eventuell Angabe eines Files, welcher die zu verarbeitenden Daten enthält, zum Zeitpunkt des Aufrufes zu modifizieren. Sehr gerne setzt man auch einen Parameter, welcher angibt, ob Zwischenergebnisse ausgegeben werden sollen oder nicht. Um nun solche Argumente abarbeiten zu können verwendet man die **vollständige** Definition der Funktion `main`:

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

Die Variable `argc` gibt dabei die Zahl der vom System übergebenen Argumente an und sie hat stets einen Wert ≥ 1. `argv` zeigt auf ein Feld von Pointern vom Typ `char`, welche ihrerseits auf Strings zeigen, welche die Parameter des aufrufenden Systembefehles wiedergeben. Wird das Programm `prog1` etwa durch

```
% prog1 -v -ftest.dat
```

aufgerufen, so zeigt `*argv` auf den String `prog1`, `*(argv+1)` auf den String `-v` und `*(argv+2)` auf `-ftest.dat`. `argc` hat den Wert 3. Es ist des weiteren UNIX-Konvention, daß Parameter im Programmaufruf stets mit `-` beginnen.

Programm 12:

Schreiben Sie eine Funktion `stparg`, welche es erlaubt Parameter, welche beim Programmaufruf angegeben werden, zu entschlüsseln. Insbesonders sollte diese Funktion feststellen können, ob ein bestimmter Parameter gesetzt wurde, etwa `-v`. Wurde ein bestimmter Parameter gesetzt, so sollte die Funktion die Zusatzinformation an das aufrufende Programm übermitteln. Also etwa beim Parameter `-ftest.dat` den Filenamen `test.dat`.

4.8 Dynamische Memory-Zuweisung

Es gibt zwei System-Funktionen, welche die dynamische Zuordnung von Memory-Bereichen während der Exekution eines Programmes gestatten:

- `void *malloc(size_t n);`

Diese Funktion gibt einen Pointer vom Typ `void` auf einen Memory-Bereich von `n` Bytes zurück. Ist eine Zuordnung nicht möglich, so wird ein `NULL` Pointer zurückgegeben. Der Varaiblen-Typ `size_t` ist ein Synonym für den Variablen-Typ `int`; er soll andeuten, daß der Parameter `n` als Größenangabe zu verstehen ist. Will man also ein `double` Feld von 200 Elementen dynamisch zuordnen, kann man wie folgt vorgehen:

```
double *pArray;
if ((pArray = (double*)malloc(200*sizeof(double)))
                          == NULL) {
     printf("No memory available\n");
     return 0;
}
pArray[20] = 322.5;
.
free(pArray);   /* Freigabe des Memory */
```

- `void *calloc(size_t n, size_t size);`

  Diese Funktion leistet an sich dasselbe wie die Funktion `malloc`. Der einzige Unterschied besteht darin, daß nun der zugeordnete Memory-Bereich mit Null initialisiert wird. Damit lautet obiger Aufruf:

```
double *pArray;
if ((pArray = (double*)calloc(200,sizeof(double)))
                          == NULL) {
     printf("No memory available\n");
     return 0;
}
pArray[20] = 322.5;
.
free(pArray);   /* Freigabe des Memory */
```

Dynamisch zugeordnete Memory-Bereiche **müssen** vor Verlassen des Programmes (einer Funktion) unter Verwendung der Funktion

```
void free(void *);
```

and das System zurückgegeben werden, wie in den obigen Beispielen bereits angedeutet wurde.

Vor Verwendung dieser Funktionen ist es angebracht die Manualseiten aufmerksam durchzulesen. Die Funktionen benötigen nämlich das Einbinden eines bestimmten Header-Files (üblicher Weise `stdlib.h`), welcher manchmal Produktabhängig ist.

4.9 Pointer auf Funktionen

Es kommt in Anwendungen häufig vor, daß man Funktionen als Parameter an eine Funktion zu übergeben hat. Denken Sie zum Beispiel an ein Integrationsprogramm, welches numerisch das Integral über einen Integralkern zu berechnen hat, welchen man seinerseits in einer Funktion definieren wird.

Es ist ja üblicher Weise ein bestimmtes numerisches Verfahren zur Quadratur für eine große Anzahl von Integralkernen anwendbar und es ist nicht unbedingt einzusehen, daß man jedesmal ein neues Programm schreiben muß nur weil sich der Integralkern geändert hat.

Grundsätzlich wird der Pointer auf eine Funktion, welche vom Typ `void` ist, und welche keine Parameter hat durch

```
void (*foo)(void);
```

deklariert. Ein Pointer auf eine Funktion, welche einen Wert vom Typ `int` zurückgibt, und welche zwei Parameter hat (einen vom Typ `int` und einen vom Typ `double`), wird dann wie folgt zu deklarieren sein:

```
int (*foo)(int,double);
```

Eine Integrationsroutine, welches folgendes Integral zu bestimmen hat

$$I = \int_0^{2\pi} d\varphi \, \cos^2 \varphi$$

wird etwa wie folgt zu definieren sein:

```
double integral(double,double,double,double(*)());
double fu(double);

int main()
{
     double relerr=1.0e-4,   /* relative Genauigkeit */
          res;   /* Resultat */
     const double tpi = 8.0*atan(1.0);

     res = integral(0.0,tpi,relerr,fu);
     printf("Ergebnis = %lf\n",res);
}

double integral(double a,   /* untere Grenze */
                double b,    /* obere Grenze */
                double err,   /* relative Genauigkeit */
                double (*f)(double))
{
     double x,fu,res=0.0;
     .
     .
     fu = (*f)(x);   /* Integralkern an Stelle x */
     .
     .
     return res;
}

double fu(double x)   /* Integralkern */
{
     double r=cos(x);
     return r*r;
}
```

Man kann nun natürlich auch Felder von Pointern auf Funktionen deklarieren. Dies wird vor allem dann von Vorteil angewendet werden können, wenn man eine Entwicklung nach einem Basisfunktionensystem zu programmieren hat.

Bevor das hier besprochene in Programme umgesetzt werden soll, noch ein kurzes Wort zur doch recht 'kryptischen' Form der Deklaration eines Pointers auf eine Funktion. Da die zunächst in den Sinn kommende Deklaration

```
void *foo(void);
```

bereits eine Deklaration für eine Funktion ist, welche einen Pointer vom Typ `void` zrückgibt, und welche keine Parameter hat so ist sie für unsere Zwecke unbrauchbar.

Zum anderen können wir - bei einigem guten Willen - in

```
function_name(...)
```

`(...)` als einen 'Funktionsoperator' interpretieren, welcher aus dem Variablennamen `function_name` den Namen einer Funktion macht. Weiters wissen wir, daß der unäre Operator `*` in der Deklaration einer Variablen diese als Pointer ausweist, und daß dieser Operator in der Deklaration ganz offensichtlich von niedrigerer Präzendenz sein muß als der `(...)`-Operator, da sonst die oben gegebene Interpretation von `void *foo(void)` nicht gültig sein kann. Um das Präzedenzdilemma aufheben zu können, muß man also durch eine geeignete Klammerung die Interpretation der Variablen `foo` als Pointer auf eine Funktion, welche keinen Wert zurückgibt, erzwingen, also

```
void (*foo)(void);
```

wie bereits angeführt.

Soll hingegen `foo` ein Pointer auf eine Funktion sein, welche einen Wert vom Typ `void *` zurückgibt, so muß man folgende Deklaration einführen:

```
void *(*foo)(void);
```

Programm 13:

Verwenden Sie das Library-Programm `qsort` zum Sortieren von eindimensionalen Datenfeldern verschiedenen Datentyps (`int` und `double`). Schreiben Sie die dazu erforderlichen Vergleichsfunktionen entsprechend der im Manual angegebenen Spezifikationen.

Programm 14:

Führen Sie die Entwicklung nach einem vollständigen Funktionensystem (in unserem Fall Besselfunktionen zweiter Art) nach der Formel

$$f(x) = \sum_{j=0}^{N} a_j \, Y_j(x)$$

durch. Beschränken Sie sich auf $N = 3$ und

$$(a)_j = (0.1, 1.5, 0.2, 0.01)$$

an der Stelle $x = 1.0$. Verwenden Sie einerseits ein Feld von Pointern für die Basisfunktionen und überprüfen Sie das Ergebnis andererseits durch direktes Ausmultiplizieren. Achtung: Die Besselfunktionen der Ordnung $j > 1$ haben in ihrem Funktionsaufruf zwei Parameter! Dies muß durch geeignete Einführung von neuen Funktionen umgangen werden.

Kapitel 5

Dateneingabe und -ausgabe

Dateneingabe und -ausgabe sind *nicht* Teil der Programmiersprache C, aber sie sind natürlich notwendige Voraussetzungen damit der Benutzer mit dem Programm und umgekehrt wechselwirken kann. Es hat sich in den Jahren ein Standardsatz von Funktionen eingebürgert, welcher heute als 'Teil' von C interpretiert werden kann. Diese Standardfunktionen wurden inzwischen auch in den ANSI-Standard übernommen, sodaß man davon ausgehen kann, daß diese Funktionen überall mit identer Funktionalität zur Verfügung stehen.

5.1 Grundsätzliches

Die grundsätzliche Idee der Standard-Dateneingabe bzw. -ausgabe besteht darin, daß sämtliche Operationen auf Texteingabe bzw. -ausgabe aufgebaut sind. Man hat also einen Strom (*stream*) von Zeichen (Bytes), aus welchen die Ein/Ausgabe aufgebaut sein wird. Strukturiert wird dieser Zeichenstrom durch Sonderzeichen, wie `\c` (carriage return) oder `\n` (carriage return plus line feed). Die Strukturierung kann schließlich auch darin bestehen, daß man problemabhängig Records (Felder mit einer bestimmten Anzahl von Bytes) liest/schreibt.

Alle Funktionen der Standard-Ein/Ausgabe sind im Header-File

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

definiert.

Man bezeichnet nun ganz allgemein diese Zeichenströme als **Files** und das Basiselement der Ein/Ausgabeoperationen ist ein Pointer, welcher auf einen File zeigt - der **Filepointer**:

```
FILE *fp;  /* Definition der Variablen fp */
           /* als Filepointer */
```

Es gibt drei vordefinierte Filepointer:

```
FILE *stdin;   /* Standard Eingabe */
               /* Tastatur des Terminals */
FILE *stdout;  /* Standard Ausgabe */
               /* Bildschirm des Terminals */
FILE *stderr;  /* Fehler - File */
               /* Bildschirm des Terminals */
```

Diese Pointer sind bereits in `stdio.h` vordefiniert und müssen nicht mehr deklariert werden!

Es wurde ein eigener Satz von Unterprogrammen definiert, welche mit `stdin` und `stdout` kommunizieren, sie sollen hier aber nicht gesondert besprochen werden, da sie nur Sonderfälle etwas allgemeiner definierter Funktionen sind.

5.2 Der Filezugang

Ein Datenfile muß dem Programm erst einmal zur Verfügung gestellt werden, damit auf diesem File Ein/Ausgabeoperationen ausgeführt werden können. Dazu muß eine Verbindung zwischen dem Filepointer und dem physischen File (etwa auf einer Fsetplatte) hergestellt werden. Man bedient sich dazu der Funktion `fopen`:

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

`name` Filename (wenn nötig mit Pfad, etc)

`mode` Bearbeitungsmodus

- `"r"` Öffnen zum Lesen. Der File muß also existieren.
- `"w"` Ein File wird zum Schreiben geöffnet.
  Besteht er bereits, so geht der alte Inhalt verloren.
  Besteht er noch nicht, so wird er erzeugt.
- `"a"` (append) An einen bestehenden File werden neue Daten am Ende des Files angeschlossen. Besteht der File noch nicht, so wird er erzeugt.
- `"r+"` Öffnen des Files zum Update (Lesen **und** Schreiben.) Der File existiert bereits.
- `"w+"` Ein File zum Schreiben **und** Lesen wird erzeugt.
- `"a+"` (append) Es wird an das Ende des Files angeschlossen, gleichzeitig ist Lesen und Schreiben erlaubt.

Die Files `stdin`, `stdout` und `stderr` sind stets einem Programm zugeordnet, wobei `stdin` nur zum Lesen verwendet werden kann, während `stdout` und `stderr` nur zum Schreiben verwendet werden können.

Kann die Verbindung zu einem File nicht hergestellt werden, so gibt `fopen` einen `NULL`-Pointer als Filepointer zurück. Man kann dies zur Ausgabe von Fehlerhinweisen nützen:

```
FILE *fp;
if ((fp = fopen("xxxx.dat","r")) == NULL) {
     Fehlermitteilungen
     Fehleranzeige für aufrufendes Programm
     return ...;
}
```

Wurde die Verarbeitung eines Files abgeschloßen, so wird der Filezugang mit Hilfe von

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

beendet. Durch diesen Befehl werden alle noch in Buffern befindliche Daten auf den File übertragen (wenn auf den File geschrieben wurde), die Buffer werden gelöscht und die Verbindung zwischen Programm und physischem Datenfile wird getrennt.

Tritt beim Aufruf von `fclose` ein Fehler auf, so wird der Wert `EOF` zurückgegeben. Eine erfolgreiche Operation wird durch Rückgabe des Wertes 0 signalisiert.

Bei gebufferter Datenübertragung (wie etwa beim Terminal) ist es oft erforderlich die Buffer vor einer Operation zu leeren. Hiezu dient:

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

Insbesonders dient

```
fflush(stdin);
```

zum Löschen des Terminalbuffers von nicht zu bearbeitenden Zeichen. Wird `fflush` für einen File ausgeführt, auf welchen geschrieben wird, so führt das dazu, daß der Inhalt des Schreibbuffers physisch auf den File übertragen wird. `fflush` gibt `EOF` zurück, wenn ein Schreibfehler aufgetreten ist, sonst wird der Wert 0 zurückgegeben. Im allgemeinen ist die Wirkung von `fflush` auf gebufferte Eingabe nicht definiert!

Weiters verwendet man noch die Funktion `freopen`, wenn man einen bestehenden Filepointer einem anderen File zuordnen möchte:

```
FILE *freopen(const char *name, const char *mode, FILE *fp);
```

Die ersten zwei Parameter haben die idente Bedeutung wie bei der Funktion `fopen`; der dritte Parameter `fp` bezeichnet dann einen bereits verwendeten Filepointer.

```
FILE *fx;
if ((fx = fopen("xxxx.dat","r")) == NULL) {
     Fehlermitteilungen
     Fehleranzeige für aufrufendes Programm
     return ...;
}
.
.
fclose(fx);
.
if ((fx = freopen("yyyy.txt","r",fx)) == NULL) {
     Fehlermitteilungen
     Fehleranzeige für aufrufendes Programm
     return ...;
}
.
fclose(fx);
```

Es gibt weiters eine Funktion, welche es erlaubt den Fehlerstatus eines Files abzufragen:

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

Der Wert 0 zeigt an, daß kein Fehlerstatus gegeben ist. Ein Wert ungleich 0 zeigt einen Fehlerstatus an. Schließlich kann man noch mit

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

abfragen, ob das Ende eines Datenfiles erreicht wurde. Ein von 0 verschiedener Wert gibt an, daß das Ende des Files erreicht wurde, auf welchen der Filepointer `fp` weist.

Es gibt noch eine Reihe weiterer Funktionen, welche es erlauben Files zu manipulieren, wie etwa `remove` zum Entfernen eines Files, `rename` zum Umbenennen eines Files, `tmpfile` zum Zuordnen von temporären Files, oder Funktionen zum Setzen der Bufferung (`setvbuf`, `setbuf`). Die Manualseiten des UNIX-Systemes geben Auskunft über die Verwendung dieser Funktionen.

5.3 Dateneingabe

Die Basisoperation wird von der Funktion

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

zur Verfügung gestellt, welche von einem File, auf welchen der Filepointer `fp` zeigt, ein Byte einliest. `fgetc` gibt also ein Byte als `unsigned char` umgewandelt in `int` an das aufrufende Programm zurück. Der Wert `EOF` wird zurückgegeben, wenn das Ende des Files erreicht wurde.

Die folgende Anweisungssequenz liest einen File zeichenweise bis zum Ende ein:

```
FILE *fp;
char c;
if ((fp = fopen("xxxx.dat","r")) == NULL) {
     Fehlermitteilungen
     Fehleranzeige für aufrufendes Programm
     return ...;
}
while ((c = (char)fgetc(fp)) != EOF) {
     weitere Verarbeitung
}
fclose(fp);
```

Will man Zeichen vom Terminal einlesen, so benützt man

```
char c;
c = fgetc(stdin);
```

Den nächsten Schritt in der Hierarchie von Funktionen zum Einlesen von Daten stellt die Funktion

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

dar. Sie erlaubt es vom File, auf welchen der Filepointer `fp` weist, `n-1` Zeichen (Bytes) in das Feld `t` zu übertragen. Das Einlesen wird beendet, wenn eines der Zeichen `\c` oder `\n` entdeckt wurde (auch wenn noch nicht `n-1` Zeichen eingelesen wurden). Das Abschlußzeichen (`'\c'` oder `'\n'`) ist dabei in `t` enthalten, weiters wird `t` mit `'\0'` abgeschloßen. Wurden `n-1` Zeichen eingelesen, so wird eine weitere Übertragung abgebrochen und `t` mit `'\0'` abgeschlossen. Eine erfolgreiche Übertragung der Daten wird dadurch signalisiert, daß `fgets` den Wert des Pointers `t` zurückgibt. Der `NULL`-Pointer wird zurückgegeben, wenn das File-Ende aufgefunden wird, oder wenn sonst ein Fehler aufgetreten ist.

Ein File kann wie folgt zeilenweise abgearbeitet werden:

```
#define BUFF_SIZE 150
FILE *fp;
char buff[BUFF_SIZE];

if ((fp = fopen("xxxx.dat","r")) == NULL) {
     Fehlermitteilungen
     Fehleranzeige für aufrufendes Programm
     return ...;
}
while(fgets(buff,BUFF_SIZE,fp) != NULL) {
     Verarbeiten der Daten
}
fclose(fp);
```

Auf der nächsten Stufe der Hierarchie findet man die Funktion

```
size_t fread(void *t, size_t size, size_t nobj, FILE *fp);
```

Mit Hilfe dieser Funktion werden in das Feld `t` `nobj` Objekte der Größe `size` in Bytes von einem File übertragen, auf welchen der Filepointer `fp` zeigt. Die Funktion hat den Wert `nobj`, wenn sie erfolgreich ausgeführt werden konnte, sonst wird die Zahl der tatsächlich eingelesenen Objekte angegeben. `feof` und `ferror` müssen zur Fehleranalyse herangezogen werden. Beispiel:

```
#define SIZE 200
.
double array[SIZE];
FILE *fp;
int i;
.
.
i = fread(array,sizeof(double),SIZE,fp);
```

Schließlich gibt es noch das formatierte Einlesen mit Hilfe der Funktion

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

Der String `format` definiert das Format, nach welchem die Daten von jenem File eingelesen und konvertiert werden sollen, welcher dem Filepointer `fp` zugeordnet ist. Die Daten werden dabei in Variable gespeichert, auf welche **Pointer** weisen, die anstelle von `...` in der Parameterliste folgen. Die Funktion beendet das Einlesen von Daten, wenn das Format abgearbeitet ist und gibt dann die Zahl der erfolgreichen Umwandlungen an das aufrufende Programm zurück. Die Funktion gibt `EOF` zurück, wenn das Ende des Datenfiles aufgefunden wurde.

Die Funktion ignoriert Leerzeichen und Tabulatoren und ist eigentlich mit sehr großer Vorsicht anzuwenden, insbesonders dann, wenn von `stdin` eingelesen werden soll. Hier verwendet man besser

```
int sscanf(char *s, const char *format, ...);
```

welche die im String `s` enthaltene Zeichenkette umwandelt. Man geht also etwa wie folgt vor:

```
#define SIZE 132
char buff[SIZE];
double data;

fflush(stdin);
fgets(buff,SIZE,stdin);
sscanf(buff,"%f",&data);
```

Es gelten die folgenden Umwandlungscodes im Formatstring:

- `%d` Umwandlung in `int`
- `%x` Umwandlung in `int` hexadezimal interpretiert
- `%o` Umwandlung in `int` oktal interpretiert
- `%i` Umwandlung in `int`, `0x..` wird hexadezimal interpretiert, führende 0 wird oktal interpretiert
- `%u` Umwandlung in `unsigend int`
- `%c` Umwandlung in `char`
- `%s` Umwandlung in String. Einlesen wird mit `'\n'` beendet. Der String wird mit `'\0'` abgeschlossen.
- `%e` Umwandlung in `float, double` Exponentialdarstellung wird erwartet
- `%f` Umwandlung in `float, double` Fließkommadarstellung wird erwartet
- `%g` Umwandlung in `float, double` beliebiges Datenformat

Sollte, etwa beim Einlesen einer Tabelle eine Kolonne überlesen werden, dann gibt man im Format etwa `%*d` an. Dies führt dann dazu, daß eine Integer-Zahl überlesen wird. Es muß natürlich auch kein Pointer auf eine Variable in der Parameterliste für diese Kolonne angegeben werden.

5.4 Datenausgabe

Für die Ausgabe von Daten gilt diesselbe Hierarchie von Funktionen:

1. Die Funktion

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

schreibt ein Zeichen (Byte) auf den File, welcher dem Filepointer `fp` zugeordnet ist. Es wird das geschriebene Zeichen als Funktionswert zurückgegeben, wenn die Operation erfolgreich verlaufen ist. Der Funktionswert ist `EOF`, wenn ein Fehler aufgetreten ist.

2. Die Funktion

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

schreibt den String `s` auf den File, welcher dem Filepointer `fp` zugeordnet ist. Der String **sollte** das Zeichen `'\n'` am Ende des Dateninhaltes enthalten, um ein geordnetes (zeilenweises) Wiedereinlesen der auf den File geschriebenen Daten sicherzustellen. Die Funktion hat den Wert `>= 0`, wenn alles gut gegangen ist. Bei einem Übertragungsfehler hat die Funktion den Wert `EOF`.

3. Die Funktion

```
size_t fwrite(const void *p, size_t size, size_t nobj,
              FILE *fp);
```

schreibt aus dem Feld `p` `nobj` Objekte der Größe `size` auf den File, welcher dem Filepointer `fp` zugeordnet ist. Die Funktion hat einen Wert, welcher der Zahl der geschriebenen Objekte entspricht, also `<= nobj`. Ist der Wert `< nobj`, so ist ein Fehler bei der Übertragung der Daten aufgetreten.

4. Die Funktion

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

erlaubt eine formatisierte Ausgabe von Daten auf den File, welcher dem Filepointer `fp` zugeordnet ist. Insbesonders gilt:

`fprintf(stdout,...);` entspricht `printf(...);`

Für das Format `format` gelten grundsätzlich jene Regeln, welche bereits bei der Besprechung der Funktion `fscanf` aufgestellt wurden. Zusätzlich können aber zwischen dem `%`-Zeichen und dem Umwandlungssymbol

- Hinweise angegeben werden, welche die Umwandlung von Zahlen modifiziert:

`-`	die Konvertierung wird linksbündig ausgeführt
`+`	eine Zahl wird stets mit Vorzeichen ausgegeben
␣	ist die Zahl positiv, so wird die erste Position mit einem Leerzeichen gefüllt, sonst folgt ein - Zeichen.
`0`	Umwandlung mit führenden Nullen
`o`	erste Stelle ist stets eine Null
`x`	`0x` wird dem Ergebnis vorangestellt

- eine Zahl angegeben werden, welche eine minimale Feldlänge definiert. So erzwingt etwa `%4d` eine Umwandlung des Inhaltes einer Variablen vom Typ `int` in ein Feld von *minimal* 4 Zeichen.
- ein Dezimalpunkt angegeben werden, welcher die Feldlänge von der Genaugkeit trennt.
- eine Zahl angegeben werden, welche einem Dezimalpunkt folgt, welche die Maximalzahl von Zeichen angibt, welche aus einer Zeichenkette zu drucken sind. Etwa wandelt `%20.10s` eine Zeichenkette in in Feld von 20 Zeichen um, in welches maximal 10 Zeichen der auszugebenden Zeichenkette einzutragen sind. Der Rest wird mit Leerzeichen aufgefüllt. Die Angabe `%7.3lf` wiederum wandelt den Inhalt einer Variablen vom Typ `double` in eine Fließkommadarstellung um, welche *mindestens* 7 Zeichen lang ist (inlusive Dezimalunkt) und 3 Stellen hinter dem Dezimalpunkt anzeigt.

5.5 Positionierung im File

Bei Files, welche mit read/write geöffnet wurden (also: `"r+"`, `"w+"`, oder `"a+"`) ist es notwendig die jeweilige Lese/Schreibposition **vor** der eigentlichen Lese/Schreiboperation zu definieren.

Dazu dient die Funktion

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

Diese Funktion erlaubt die Definition der aktuellen Lese/Schreibposition. Die Position ist dabei `offset` Bytes von jener Position entfernt, welche durch den Parameter `origin` definiert ist. Dieser Parameter kann wie folgt gegeben sein:

`SEEK_SET`	`offset` ist relativ zum Fileanfang definiert.
`SEEK_CUR`	`offset` ist relativ zur gegenwärtigen Position gegeben.
`SEEK_END`	`offset` ist relativ zum Fileende gegeben.

Tritt ein Fehler auf, so hat `fseek` einen von 0 verschiedenen Wert.

Die Funktion

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

hat als Wert die gegenwärtige Lese/Schreibposition relativ zum Anfang des Files, welcher dem Filepointer `fp` zugeordnet ist. Im Fall eines Fehlers hat die Funktion den Wert `-1L`.

Die Funktion

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

setzt den File, welcher dem Filepointer `fp` zugeordnet ist, auf den Anfang zurück. Sie ist äquivalent dem Aufruf `fseek(fp,0L,SEEK_SET);`.

Programm 15:

Einlesen und Bearbeiten eines vorgegebenen Datenfiles. Dieser Datenfile enthält ein Energiespektrum. Die Daten sind wie folgt organisiert:

1. Datensatz:	Textzeile von 80 Zeichen Länge zur Datenbeschreibung.
2. Datensatz:	Erste Zahl: Zahl der Datenpunkte `int`.
	Zweite Zahl: Höchster Energiewert `double`.
Weitere Daten:	Die weiteren Daten entsprechen den Spektralwerten.
	Variablentyp `double`, beliebiges Format,
	beliebige Anzahl von Zahlen pro Zeile.

Die Datenpunkte sind dabei einem äquidistatem Energiespektrum zugeordnet, welches mit der Energie 0 meV beginnt. Es ist dynamische Memoryzuordnung zum Speichern der Spektraldaten zu verwendet. Generieren Sie zunächst einen Datenfile, welcher eine Tabelle enthält, in welcher jedem Spektralwert (y-Achse) ein Energiewert (x-Achse) zugeordnet ist. (Verwenden Sie `gnuplot` zur graphischen Darstellung!) Führen Sie im weiteren eine Kurvendiskussion durch. Es sind somit alle Maxima (Peaks) im Spektrum zu bestimmen (geht über Nullstellen der ersten Ableitung) und wenn möglich sollte auch die Halbwertsbreite der Peaks bestimmt werden. Der Name des zu bearbeitenden Files kann entweder über einen Commandline Parameter angegeben werden oder durch direktes Einlesen von `stdin`. Die Unterprogramme zum Bearbeiten des Filenamens sind zu benützen.

5.6 In-Memory Formatierung

Es wurde bereits auf Seite 49 die Funktion `sccanf` eingeführt. Diese Funktion hat die gleiche Aufgabe, wie die Funktion `fscanf`, nur daß im Fall von `sscanf` der umzuwandelnde Datensatz nicht von einem File einzulesen ist, sondern in der Zeichenkette `s` enthalten ist.

Die Umkehrfunktion lautet:

```
int sprintf(char *t,const char *format, ...);
```

mit derselben Funktionalität wie die Funktion `fprintf`, nur daß die umgewandelten Daten nunmehr in die Zeichenkette `t` übertragen werden und nicht auf einen File. Häufiger Fehler: die Dimension der Zeichkette `t` wird zu klein angegeben. `sprintf` kann aber die Größe des zur Vefügung stehenden Memory-Bereiches nicht überprüfen, sodaß es zu Überschreibungen kommen kann.

Kapitel 6

Strukturen

6.1 Definition

Strukturen sind Datenkonglomerate, welche zumeist ein Objekt beschreiben. So ist etwa ein Punkt im Raum durch drei (vier) Koordinaten beschrieben und man wird diese drei (vier) Zahlen sinnvoller Weise zu einer **Struktur** zusammenfassen. Strukturen sind also *Objekten* zugeordnet.

Das Schlüsselwort `struct` eröffnet die Definition einer Datenstruktur. Der Struktur **kann** ein *Name* gegeben werden. Die Elemente der Struktur werden durch Variablendeklarationen definiert. Diese Deklarationen sind in geschlungene Klammern eingeschlossen und stellen den *Körper* der Struktur dar, also

```
struct name {
     Variablendeklarationen
};
```

Eine so definierte Struktur führt gleichzeitig einen neuen *Datentyp* ein, wobei dieser durch `struct name` definiert wurde. Etwa:

```
struct point_3D {
    double x,y,z;
} p1,p2;
```

ist eine Variablendeklaration für zwei Variable `p1` und `p2`, welche nun vom Typ `struct point_3D` sind, also Punkten in 3D entsprechen.

Man kann hier transparenter vorgehen, indem man zunächst die Struktur (etwa in einem Header-File) definiert:

```
struct point_3D {
    double x,y,z;
};
```

und dann (etwa im selben Header-File) durch die Typendefinition

```
typedef struct point_3D POINT3D;
```

den neuen *Datentyp* `POINT3D` definiert. Die Variablen `p1` und `p2` werden dann mit

```
POINT3D p1,p2;
```

als Variable vom Typ `POINT3D` deklariert. Dies ist vollkommen äquivalent zur Deklaration:

```
struct point_3D p1,p2;
```

Es ist gute Praxis neue Datentypen, welche über `typedef` eingeführt werden mit Großbuchstaben zu bezeichnen, da sie so leicht ale neue Datentypen erkennbar sind. Ein Beispiel für einen solchen, neuen, Datentyp haben wir mit dem Filepointer `FILE` in Kapitel 5 kennengelernt, welcher im Header-File `stdio.h` definiert ist.

Man könnte nun den Raumpunkt $(1.0, 2.0, 3.0)$ wie folgt deklarieren:

```
POINT3D p1 = {1.0, 2.0, 3.0};
```

Strukturen können aber auch verschachtelt definiert werden. So definiert etwa

```
struct cube {
     POINT3D p1,p2,p3;
};
```

eine Struktur, welche jene drei Raumpunkte enthält, welche etwa einen Würfel definieren.

Wie kann man nun auf die Elemente einer Struktur zugreifen? Es wird ja erforderlich sein ihren Wert zu modifizieren oder aber auch ihren Wert auszugeben. Hiezu gibt es einen eigenen Operator, den *Element von*, oder .-Operator, welcher wie folgt definiert ist:

Variable.*Element*

wobei *Variable* eine Variable von einem Typ sein muß, welcher einer Struktur entspricht und *Element* muß der Name einer Variablen sein, welche innerhalb der Struktur deklariert ist. So weist man etwa die y-Koordinate des 3D-Punktes `p1` einer Variablen `z` zu:

```
double z;
POINT3D p1 = {1.0, 2.0, 3.0};

z = p1.y;
```

z wird dann der Wert 2.0 zugewiesen.

Von großer Bedeutung ist auch der Wertzuweisungsoperator im Zusammenhang mit Strukturen:

```
POINT3D p1 = {1.0, 2.0, 3.0},p2;

p2 = p1;
```

führt dazu, daß sämtliche Struktur-Elemente der Variablen p2 auf jenen Wert gesetzt werden, welcher in dem entsprechenden Struktur-Elementen der Variablen p1 gespeichert sind. Auch der *Ausdruck* `p2 = p1` hat einen Wert vom Typ POINT3D! Dies entspricht somit folgenden Anweisungen:

```
POINT3D p1 = {1.0, 2.0, 3.0},p2;

p2.x = p1.x;
p2.y = p1.y;
p2.z = p1.z;
```

Abschließend sei noch vermerkt, daß `sizeof(struct name)` die Größe der Datenstruktur in Bytes wiedergibt.

6.2 Pointer auf Strukturen

Da Strukturen nur einen neuen Typ von Variablen definieren, so ist es natürlich möglich auch Pointer auf Variable einzuführen, welche vom Typ einer Struktur sind, also

```
POINT3D p1 = {1.0, 2.0, 3.0};
POINT3D *pPoint;

pPoint = &p1;
```

Damit zeigt die Variable pPoint auf die Variable p1 und ist vom Typ POINT3D. Wie kann man nun wieder auf das Element einer Struktur zugreifen, wenn man nur einen Pointer auf eine Variable vom Typ `struct name` zur Verfügung hat? Die auf den bisherigen Kenntnissen aufbauende Lösung wäre:

```
POINT3D p1 = {1.0, 2.0, 3.0};
POINT3D *pPoint;
double z;

pPoint = &p1;
z = (*pPoint).y;
```

um der Variablen `z` den Wert der y-Koordinate des Punktes zuzuweisen, auf welche die Variable `pPoint` zeigt. Man beachte hier die Klammerung, welche aus der Tatsache folgt, daß der `.` Operator eine höhere Präzedenz hat als der `*`-Operator. `*pPoint.y` würde zu einer Fehlermeldung führen, da das Element `y` der Struktur `point_3D` kein Pointer auf `double` ist!

Die hier verwendete Konstruktion ist nicht sehr 'schön', weshalb man einen weiteren Operator eingeführt hat, den `->`-Operator, oder Zeiger auf Element einer Struktur Operator. Er hat die Form:

Pointer auf Struktur`->`*Element*

Man schreibt also in obigem Beispiel:

```
POINT3D p1 = {1.0, 2.0, 3.0};
POINT3D *pPoint;
double z;

pPoint = &p1;
z = pPoint->y;
```

was wesentlich besser zu lesen ist.

6.3 Strukturen und Funktionen

Strukturen können als Pointer oder als 'Wert' an eine Funktion übergeben werden. In letzterem Fall wird der Funktion eine exakte Kopie der Struktur zur Vefügung gestellt, was natürlich speicherplatzaufwendig sein kann. Es können dann natürlich die Elemente der Struktur nicht nach außen sichtbar verändert werden. (Hiezu gibt es allerdings eine interessante Alternative!) Wird hingegen ein Pointer auf eine Struktur übergeben, so können die Elemente jener Struktur, auf welche der Pointer verweist, auch für 'außen' sichtbar verändert werden.

Es ist nun wesentlich, daß auch der Funktionstyp einer Struktur entsprechen kann. Man könnte sich eine Funktion überlegen, welche zwei Fließkommazahlen zu einer komplexen Zahl zusammenfaßt, wobei wir den Typ der komplexen Zahl durch eine Struktur einführen wollen:

```
struct complex {
     double re,im;                  /* re = Realteil */
                                    /* im = Imaginärteil */
};
typedef struct complex COMPLEX;
COMPLEX Cmplx(double,double);

int main()
{
     COMPLEX cI;
     cI = Cmplx(0.0,1.0);           /* cI .. imaginäre Zahl */
     .
}

COMPLEX Cmplx(double r, double i)
{
     COMPLEX new;
     new.re = r;
     new.im = i;
     return new;
}
```

Die Funktion `Cmplx` nimmt also einen Wert an, welcher vom Typ `COMPLEX` ist. Diese Möglichkeit erlaubt es nun Veränderungen, welche in den Elementen einer Struktur ausgeführt wurden, nach 'außen' zu transportieren, wie es anhand einer Funktion zur Addition zweier komplexer Zahlen demonstriert werden soll:

```
COMPLEX Cadd(COMPLEX a, COMPLEX b)
{
     a.re += b.re;
     a.im += b.im;
     return a;
}
```

Programm 16:

Schreiben Sie ein Programm, welches Funktionen zum Initialisieren, Addieren, Subtrahieren, Multiplizieren und Dividieren zweier komplexer Zahlen definiert. Weiters ist noch eine Funktion zur Berechnung des Absolutwertes einer komplexen Zahl zu schreiben. Erzeugen Sie den zugehörigen Header-File. Testen Sie die korrekte Funktion Ihres Programmes. Wie würde unter Verwendung Ihrer Programme der Ausdruck `a = b+c*d;` zu realisieren sein, wenn alle diese Variablen vom Typ `COMPLEX` sind?

6.4 Felder von Strukturen

Es ist natürlich möglich Felder von Variablen zu deklarieren, welche dem Dateityp einer Struktur angehören. So deklariert man mit

```
COMPLEX x[10];
```

ein Feld von 10 komplexen Zahlen. Man findet dann zum Beispiel den Imaginärteil des vierten Feldes durch

```
im = x[3].im;
```

und weist seinen Wert der Variablen `im` zu. Man könnte aber auch schreiben:

```
COMPLEX x[10],*px;
px = x;              /* px zeigt auf x[0] */
im = (px+3)->im;     /* entspricht im = x[3].im; */
```

womit offensichtlich wird, daß die Inkrementierung eines Pointers auf eine Variable vom Typ einer Struktur diese um die Größe der Struktur in Bytes erhöht, sodaß sichergestellt ist, daß der Pointer die Elemente des Feldes abtastet. Die Klammerung `(px+3)->.` ist hier notwendig, da der `->`-Operator von höherer Präzedenz ist als alle anderen hier vorkommenden Operatoren.

Programm 17: (optional)

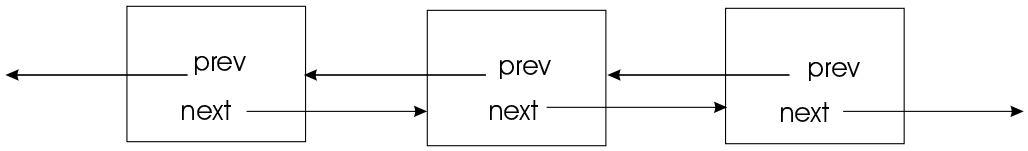
Verwenden Sie das Library-Programm `qsort` zum Sortieren von eindimensionalen Datenfeldern des Datentyps `POINT3D` aufsteigend nach ihrem Abstand vom Ursprung. Schreiben Sie die dazu erforderlichen Vergleichsfunktion entsprechend der im Manual angegebenen Spezifikationen und überprüfen Sie das Programm.

6.5 Selbstreferenzierende Strukturen

Solche Strukturen sind auch als 'linked lists' bzw. 'double linked lists' oder Binärbäume bekannt. Solche Datensammlungen beruhen auf Bauelementen, *nodes* genannt, welche etwa folgende Struktur haben:

```
struct tnode {
     Datendeklarationen
     struct tnode *prev;
     struct tnode *next;
};
```

Der Pointer `prev` zeigt auf das in der Liste vor dem aktuellen Element liegende Element und `next` zeigt auf das in der Liste nach dem aktuellen Element liegende Element der Liste. Es ergibt sich also folgende Anordnung:



Das erste Element der Liste hat `prev = NULL` gesetzt, während das letzte Element der Liste `next = NULL` gesetzt hat. Auf diese Weise kann man sich sehr leicht von einem Element der Liste zu einem Nachfolgenden bewegen und so die Liste durchsuchen.

6.6 Unions

Eine Union enthält Objekte unterschiedlichen Typs und ordnet ihnen denselben Memory-Bereich zu. Die einzelnen Objekte können dabei durchaus unterschiedlicher Länge (in Bytes) sein, sie beginnen alle an derselben Memory-Adresse. Etwa:

```
union xxx {
     char line[130];
     POINT3D p1,p2,p3;
     int x,y;
     int z;
};
```

Aufgrund dieser `union`-Defintion haben `line[0]`, `p1.x`, `x` und `z` dieselbe Memoryadresse.

Wird eher selten verwendet!

6.7 Bitfelder

Bitfelder sind von großer Bedeutung, wenn man Hardware Schnittstellen (wie etwa eine serielle Schnittstelle) programmieren muß. Einer solchen Schnittstelle ist üblicher Weise eine feste Adresse zugeordnet und jedes Bit hat eine bestimmte Bedeutung. Man könnte nun jedes Bit getrennt durch Maskierung abfragen oder setzen, man kann aber auch eine spezielle Struktur definieren:

```
struct seriell {
     unsigned int bit_0:1;
     unsigned int bit_1:1;
     .
     .
     unsigned int bit_6:8;
};
typedef struct seriell SERIELL;

int main()
{
     SERIELL *com1;
     char c;
     com1 = 0x372;                /* Setzen der Adresse */
     c = (char)com1->bit_6;
}
```

Die hier definierte Struktur besteht aus einer Reihe von Variablen, welche nur 1 Bit 'lang' sind: `bit_0` bis `bit_5`. Die Variable `bit_6` ist acht Bits lang und ihr Wert entspricht dem Zeichen, welches von der Schnittstelle empfangen wurde. Nach außen wirken die Elemente der Struktur `SERIELL` wie normale Variable vom Typ `unsigned int`.

6.8 Enumerierte Konstante

Es handelt sich dabei um eine Liste konstanter Werte, welche wie eine Struktur aufgebaut ist, aber mit dem Schlüsselwort `enum` eingeleitet wird, wie etwa

```
enum bool {
     FALSE,
     TRUE
};
```

Damit hat, definitionsgemäß `FALSE` den Wert 0 und `TRUE` den Wert 1. Man könnte nun den Typ

```
typedef enum bool BOOLEAN;
```

definieren und dann die Variable

```
BOOLEAN flag;
```

deklarieren. Eine Wertzuweisung wird dann wie folgt zu erfolgen haben:

```
flag = TRUE;
```

Man kann aber auch mehrere Werte angeben:

```
enum menu {
     FALL1 = 1,
     FALL2 = 2,
     FALL3 = 3,
     FALL4 = 5
};
typedef enum menu MENU;
```

Eine solche Variable vom Typ `MENU` kann dann vorteilhaft dazu verwendet werden eine `switch` Anweisung besser lesbar zu machen:

```
int main()
{
     MENU eMenu;
     char line[130];
     flush(stdin);
     fgets(line,130,stdin);
     eMenu = atoi(line);     /* atoi: siehe man Seiten! */
     switch (eMenu) {
     case FALL1:
          .
          break;
     case FALL2:
          .
     default:
          break;
     }
}
```

Kapitel 7

Übergang zu C++

C++ ist eine sogenannte *opjektorientierte* Programmiersprache, welche aus C abgeleitet wurde und somit alle Sprachelemente von C enthält. Die zusätzlichen Sprachelemente von C++ erlauben nun eine Strukturierung des Programmes derart, daß die Verarbeitung von Objekten in optimaler Weise erfolgen kann. Wir haben schon bei der Besprechung von Datenstrukturen (Kapitel 6, Seite 54) darauf hingewiesen, daß solche Strukturen Objekten zugeordnet werden können. Somit enthält auch die Programmiersprache C bereits eine, wenn auch minimale, Objektorientiertheit.

Während die Elemente einer Struktur in C auf Variable unterschiedlichsten Typs beschränkt sind, führt C++ eine neue Struktur ein, `class` genannt. Eine solche Objektklasse beschränkt sich nunmehr nicht nur auf Variable, wie in `struct`, sie kann nun auch Funktionen enthalten, welche die Verarbeitung von Daten, die einem Objekt zugeordnet sind, auf eine für das Objekt typische Weise ermöglichen.

Wir stellen uns nun eine komplexe Zahl als ein solches Objekt vor. Es sind dann alle arithmetischen Operationen mit komplexen Zahlen in einer für dieses Objekt typischen Weise auszuführen. Es ist daher durchaus sinnvoll die folgende (hypothetische) Definition einer Klasse von komplexen Zahlen vorzusehen:

```
class complex {
     double re,im;
     complex(void);           /* leerer Konstruktor */
     complex(double,double);  /* Konstruktor */
     complex add(complex);
     complex add(double);
};
```

Eine solche Klasse könnte dann wie folgt angewendet werden, wobei auch die entsprechenden Unterprogramme definiert werden; hier wird angenommen,

daß obige Klassendefinition in einem Header-File `complex.h` abgespeichert wurde:

```
#include "complex.h"
int main()
{
     complex a,b(1.0,2.0),c(3.0,4.0);
     double x=3.0;
     a = b.add(c); /* addiere c zu b */
     printf("Re = %lf, Im = %lf\n",a.re,a.im);
     a.add(x); /* addiere x zu a */
     printf("Re = %lf, Im = %lf\n",a.re,a.im);
     return 1;
}

complex::complex()
{
     re = im = 0.0; /* initialisieren auf Null */
}

complex complex::complex(double r,double i)
{
     re = r; /* initialisieren mit (r,i) */
     im = i;
}

complex complex::add(double r)
{
     re += a;
     return this;
}

complex complex::add(complex a)
{
     re += a.re;
     im += a.im;
     return this;
}
```

Hier wurden, um die Aufgabenstellung deutlich zu machen, eine Reihe neuer Sprachelemente eingeführt, welche auch tatsächlich in dieser Art von C++ verwendet werden, hier aber doch nur illustrativen Charakter haben sollen.

Zuallererst muß man ein Sprachelement vorsehen, welches es erlaubt Ele-

mente der Klasse zu initialisieren. Die in C vorgesehene Möglichkeit (siehe etwa Initialisierung der Elemente der Struktur `POINT3D` auf Seite 55) ist jetzt nicht mehr zielführend, da es ja nicht möglich ist Funktionen zu 'initialisieren'. Man hat also einen *Konstruktor* einzuführen, also eine Funktion, welche angibt, wie die Elemente der Klasse zu initialisieren sind. Der einfachste Fall ist der *leere* Konstruktor, welcher (in unserem Fall) die Elemente `re` und `im` auf Null setzt. (Ein Konstruktor hat immer den Namen der Klasse!). Es ist dann ein zweiter Konstruktor angegeben, welcher es erlaubt die Elemente `re` und `im` auf bestimmte vorgegebene Werte zu setzen. In der vorliegenden Deklaration würden die Variable `a` in ihren Elementen `re` und `im` auf Null gesetzt werden, die Elemente `re` und `im` der Variablen `b` werden auf 1.0 und 2.0 gesetzt, usw.

Wir sehen weiters, daß nunmehr eine Funktion, welche Mitglied einer Klasse ist, wie folgt zu definieren ist:

```
ftyp class_name::function_name(vtyp p1, vtyp p2, ...)
{

}
```

also, wie in unserem Fall für die Addition zweier komplexer Variabler: `complex complex::add(complex a)`. In der Deklaration der Funktion als Mitglied der Klasse `complex` kann dann `complex::` entfallen. Sehr häufig wird auch von einem `::`-Operator als *Member of*-Operator gesprochen. `complex::` dient zur Identifikation der Funktion `add` als Mitglied der Klasse `complex`. Man kann sich ja zum Beispiel auch eine weitere Klasse eines dreidimensionalen Vektors `vec3D` vorstellen, in welcher auch eine Funktion `add` sinvoll zu definieren wäre. Diese kann dann durch `vec3D::add` von `complex::add` unterschieden werden.

Wir sehen weiters, daß es auch möglich ist mehrere Funktionen gleichen Namens innerhalb einer Klasse zu definieren, wenn sie sich durch die Zahl der Parameter oder durch den Parametertyp voneinander unterscheiden. Wie im Beispiel gezeigt, eine durchaus sinnvolle Vereinbarung, welche die 'Lesbarkeit' des Programmes erhöht.

Schließlich wurde mit `return this` ein hypothetischer Befehl eingebaut, welcher nur symbolisieren soll, daß das Ergebnis der Datenmanipulation wieder an die 'Außenwelt' zurückgegeben werden soll, da ja die Funktion `complex::add` vom Typ `complex` ist.

Die im hier besprochenen Beispiel vorgesehene Realisierung der Addition zweier Variabler vom Typ `complex` mit `a = b.add(c);` ist noch sehr unschön. Um dies zu beheben, erlaubt die Sprache C++ die Definition von Funktionen mit Operatorcharakter. Man wird also eine Funktion mit dem Namen `+` einführen, welche ein `operator` ist, und welche zwei Variable vom

Typ `complex` verknüpft, oder auch eine Variable vom Typ `complex` mit einer Variablen vom Typ `real`. Man spricht vom *Überladen* von Operatoren. Dies erlaubt dann schließlich den Befehl `a = b+c;`. Der Programmierer ist dadurch aber nicht der Aufgabe enthoben Funktionen mit Operatorcharakter zu definieren, welche alle möglichen Verknüpfungsoperationen, die in der behandelten Klasse denkbar sind, auch zu programmieren. Hier muß man auch Typumwandlungen berücksichtigen, da sonst vom Compiler versucht wird auf die Standardtypen zurückzugreifen!

Die hier als Beispielklasse besprochene Klasse vom Typ `complex` ist in C++ bereits in der Standard-Template-Library vordefiniert. Diese Klasse enthällt alle denkbaren Operationen, welche mit komplexen Zahlen ausführbar sind, inklusive aller Standardfunktionen, für welche komplexe Argumente zugelassen sind, wie etwa `sqrt`, `sin`, `cos`, usw.

Man kann die Elemente der Standard-Template-Library leicht auch innerhalb normaler C Programme nutzen ohne deshalb das Programm vollständig auf Objektorientiertheit umzustellen. Man muß dann das Programm nur 'formal' als C++ Programm deklarieren (File-Extension `cpp` oder `c++` anstelle von `c`) und mit einem C++ Kompiler kompilieren (`g++` im Fall von LINUX an Stelle von `gcc`).

Programm 18:

Schreiben Sie Programm 16 auf C++ um, indem Sie die `complex`-Klasse mit `#include <complex>` importieren und komplexe Variable mit Elementen vom Typ `double` mit Hilfe der Deklaration

```
complex <double> c;
```

einführen.

Kapitel 8

Literatur

1. B.W. Kerningham und D.M. Ritchie
   The C Programming Language, Second Edition
   Prentice Hall Software Series (1988)
   ISBN 0-13-110362-8 {PBK}

2. A.R. Feuer
   The C Puzzle Book
   Prentice Hall Software Series (1982)
   ISBN 0-13-109926-4 {PBK}

3. S.B. Lippman
   C++ Primer, Second Edition
   Addison-Wesley (1991)
   ISBN 0-201-54848-8

4. T.L. Hansen
   The C++ Answer Book
   Addison-Wesley (1990)
   ISBN 0-201-11497-6

5. J.O. Coplien
   Advanced C++
   Addison-Wesley (1992)
   ISBN 0-201-54855-0

6. U. Breymann
   C++ Eine Einführung, 5. Aufl.
   Hanser Verlag (1999)
   ISBN 3-446-21272-8

Index

A

B

C

D

F

T

U

V

W

Z