# C

Werner Benger

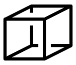
27. Mai 1997

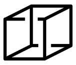
# Inhaltsverzeichnis

# Legende

Die im Text auftretenden Randsymbole haben folgende Bedeutung:

 ........................................ Achtung, Schleudergefahr!
— dieser Text behandelt eine häufige Fehlerquelle.

 ........................................ Reale Box
— dies ist die gebräuchliche und elegante Form eines Programmkonstruktes.

 ........................................ Irreale Box
— diese Version des Programmkonstruktes sollte besser vermieden werden.

 ........................................ kleine Schweinerei - hier ist irgendwo der Hund begraben.
(entsprechende Textstellen sind mit Vorsicht zu genießen!)

 ........................................ Hoppala - so eine Schweinerei! Das geht auch?
(Die hier behandelten Konstrukte sind keine besonders gefinkelten Programmtricks, sondern nur eine Konsequenz der Möglichkeiten, die C bietet. Dennoch sollte man sie nur verwenden, wenn man sie wirklich versteht.)

 ........................................ Hilfe ! Bitte nicht so etwas !
(Auch wenn der Text auf den ersten Blick vollkommen unverständlich erscheint, sollte man sich zumindest einmal im Programmiererleben die Mühe machen, ihn durchzudenken. Bitte keinesfalls die vorgestellten Konstrukte in Programmen verwenden!!)

 ........................................ Was denn, was denn !? Das gibt's doch gar nicht !
(Solche Konstrukte treten praktisch nie auf und entsprechende Probleme müssen erst einmal explizit konstruiert werden. Ihr einziger Sinn und Zweck besteht darin, ursprünglich triviale Programme zu codieren und für alle menschlichen Lebewesen — einschließlich Programmierer — unlesbar zu machen. Derartige Programmiergewohnheiten sollten keineswegs Schule machen - am besten die entsprechende Textstelle sofort wieder vergessen, außer ... )

# 1 Von 0 auf C in 8 Sekunden

... ist leider nicht möglich. Die Zahl C stand zwar schon bei den Römern für den Zahlenwert 100, aber als Programmiersprache gibt es sie erst seit dem Anfang der 70er Jahre. Sie wurde ursprünglichen von Kernighan und Ritchie aus ihren Vorgängern A und B erfunden, um damit Betriebssysteme unabhängig von der verwendeten Hardware schreiben zu können. Bis zu diesem Zeitpunkt war alle betriebssysteminterne Software in Maschinensprache (Assembler) geschrieben worden und mußte infolgedessen zwangsläufig für jede Hardeware neu erfunden werden. Um dieses Manko zu beheben, wurde C so konstruiert, daß alles möglich ist, was auch in Assembler möglich ist, nur eben auf eine maschinenunabhängige Art und Weise.

Heute ist C eine der am meisten verbreiteten Sprachen, auf fast allen Großrechnern sowie Workstations, und in zunehmendem Maße auch auf PC's, läuft UNIX, ein hardwareunabhängiges Betriebssystem, das nahezu ausschließlich in C geschrieben ist. Gerade dieses Konzept des „**maschinenunabhängigen Assemblers**" hat den Erfolg des untrennbar miteinander verbundenen Paares UNIX/C in den letzten 20 Jahren ausgemacht.

## 1.1 Grundtypen von Variablen

Grundsätzlich: Alle Namen sind Case-Sensitive. Es hat sich in C eingebürgert, generell klein zu schreiben, da der Namen einer Funktion so leichter zu merken ist.

In moderneren Anwendungen, insbesondere für die Programmierung graphischer Oberflächen, wird meist durch Groß/Kleinschreibung versucht, die Zusammenhänge von Variablen und Funktionsnamen deutlicher herauszustellen. Auf die verschiedenen Varianten und insbesondere das Konzept der „Ungarischen Notation" aus der MS-Windows-Programmierung soll aber hier nicht näher eingegangen werden. Im Wesentlichen ist die Schreibweise eine Freiheit des Programmierers, die ihm nach seinem eigenen Geschmack frei wählbar bleiben sollte.

### 1.1.1 `int`

Der Typ *int* bezeichnet eine Ganzzahl und ist der Struktur des Prozessors, auf dem das Programm läuft, optimal angepaßt. Durch die unterschiedlichen Prozessortypen ist der Typ *int* **abhängig** von der jeweiligen Programmierumgebung (16-Bit oder 32-Bit Compiler). Je nach Compiler und CPU hat ein *int* eine Größe von 2 oder 4 Bytes, also einen Wertebereich von -32768 bis 32767 oder -2147483648 bis 2147483647. Daher sollte man den Typ *int* nur in Situationen verwenden, in denen der Wertebereich nicht von Bedeutung und in denen es mehr auf Ausführungsgeschwindigkeit des Programmcodes ankommt. *int* ist der Standardtyp in verschiedenen Situationen (mehr dazu später).

### 1.1.2 `short,long`

Um die Größe einer *int*-Variablen eindeutig festzulegen, ist der Vorsatz *short* oder *long* möglich. Ein *short int* hat 2 Bytes, ein *long int* 4 Bytes. Die Bezeichnung „*int*" kann auch jederzeit weggelassen werden, der Typ *short int* kann einfach mit *short* abgekürzt werden, analog *long*.

### 1.1.3 `signed,unsigned`

Ein weiterer möglicher Zusatz ist *signed* (Vorgabe) oder *unsigned*, je nachdem, ob negative und positive oder nur positive Zahlenwerte gewünscht sind.

### 1.1.4 Übersicht: Ganzzahltypen

Somit können also folgende Typen gebildet werden:

| Name | Bytes | Wertebereich |
|---:|:---:|:---:|
| unsigned short int | 2 | 0 bis 65535 |
| signed short int | 2 | -32,768 bis 32,767 |
| unsigned long int | 4 | 0 bis 4,294.967,295 |
| signed long int | 4 | -2,147.483,648 bis 2,147.483,647 |
| int | 2/4 | wie short oder long |
| signed int | 2/4 | wie signed short oder signed long |
| unsigned int | 2/4 | wie unsigned short oder unsigned long |
| signed | 2/4 | wie signed int |
| unsigned | 2/4 | wie unsigned int |
| short | 2 | wie signed short int |
| signed short | 2 | wie signed short int |
| unsigned short | 2 | wie unsigned short int |
| long | 4 | wie signed long int |
| signed long | 4 | wie signed long int |
| unsigned long | 4 | wie unsigned long int |

ÜBUNGSAUFGABE:

1. Markiere in obiger Tabelle alle jene Typen, die identisch sind.
2. Suche in der Tabelle die jeweils kürzesten Namen für die vier eigentlichen Grundtypen.

### 1.1.5 Portabilität

Der Typ *int* sollte immer nur dann verwendet werden, wenn der Wertebereich nicht relevant ist, sonst ist die Verwendung von *short* und *long* anzuraten, um nicht unliebsame Überraschungen zu erleben, wenn das Programm mit einem anderen Compiler übersetzt wird.

### 1.1.6 `float`, `double`

Für nichtganzahlige Rechnungen stellt C die Typen *float* und *double* zur Verfügung. Ein *float* besteht aus 4 Bytes, ein *double* aus 8. Rechnungen mit *double* sind genauer als mit *float* und bringen, vor allem am PC, nur in Ausnahmesituationen einen Geschwindigkeitsverlust. Der einzige Grund, *float*s zu verwenden, liegt im geringeren Speicherplatzbedarf.

### 1.1.7 `long long`, `long double`

Es ist auch möglich, einen Typ namens *long long int* zu bilden, dieser besteht dann aus 8 Bytes und hat einen entsprechenden Wertebereich. Allerdings wird dieser Typ nicht von allen Compilern unterstützt.

Analog ist ein Typ *long double* (10 Bytes) möglich, der am PC dem eigentlichen Wertebereich des Prozessors entspricht. Obwohl dies ein Standardtyp in C++ ist, wird er jedoch nicht von jedem Compiler gleich behandelt, bei manchen Compilern ist er identisch mit *double*.

### 1.1.8 `char`

Ein weiterer Grundtyp ist *char*, immer definiert als ein Byte. Hierbei ist zwar die Größe definiert, nicht aber das Vorzeichen. Je nach Compiler und Einstellungen desselben ist *char* vorzeichenbehaftet oder nicht. Klarheit verschafft der Typ *signed char* (-128 bis 127) bzw. *unsigned char* (0 bis 255). Die meisten Compiler bieten die Möglichkeit, per Compilerschalter das Vorzeichen des Typs *char* festzulegen. Im Allgemeinen ist „signed“ die Vorgabe, dennoch sollte man sich nicht darauf verlassen und in Fällen, in denen das Vorzeichen eine Rolle spielt, explizite *signed* oder *unsigned* angeben.

### 1.1.9 `char[]` ≡ strings

Es gibt in C – im Gegensatz zu den meisten anderen Programmiersprachen – keinen Typ „String". Dennoch gibt es etwas Ähnliches, denn der Typ *char* ist zwar für sich genommen nicht allzu bedeutsam, wichtig hingegen sind die aus *char* aufgebauten *Arrays*, die anstelle eines eigenen Typs *string* treten. Zeichenketten sind somit nicht als eigener Typ vorhanden, sondern nur ein Spezialfall eines abgeleiteten Typs. Daraus erklärt sich auch, daß „Strings" in C nicht einfach addiert, kopiert o.ä. werden können, was man beim Umstieg von einer anderen Programmiersprache erwarten würde, sondern daß hierfür entsprechende Libraryfunktionen verwendet werden müssen (die etwas später behandelt werden).

### 1.1.10 `bool`

In C gibt es keinen eigenen Typ *bool* für Variablen, die nur wahr oder falsch sein können. Als Äquivalent fungiert der Typ *int*, der entweder 0 (falsch), oder ungleich 0 (wahr) sein kann. In C++ ist *bool* ein eigener Typ mit den zusätzlichen Schlüsselwörtern *true* und *false*. Intern ist *false* gleich 0 und *true* ein Zahlenwert ungleich 0, sodaß aus Kompatibilitätsgründen die Typen *int* und *bool* jederzeit ausgetauscht werden können. Nur in speziellen Situationen in C++ spielt diese Unterscheidung eine Rolle.

### 1.1.11 `sizeof`

Mithilfe des Operators *sizeof* kann die Größe eines Typs in Bytes erfragt werden. So liefert `sizeof short` 2, `sizeof long` 4. Mit `sizeof int` kann unterschieden werden, ob das Programm mit einem 16-Bit oder einem 32-Bit Compiler übersetzt wird - je nachdem ist `sizeof int` entweder 2 oder 4.

Meist wird ein *sizeof*-Ausdruck mit Klammern geschrieben, damit er wie ein Funktionsaufruf erscheint, z.B.: `sizeof(short)`, dies ist aber nicht notwendig.

Recht praktisch ist *sizeof* in Verbindung mit Arrays (siehe 1.2), da so deren Größe ermittelt werden kann, ohne extra eine Konstante dafür definieren zu müssen.

## 1.2 Definition von Variablen

Variablen werden definiert, indem man **zuerst** ihren Typ angibt, und dann die Namen der Variablen:

```
int      i,j,k,l;
unsigned short ushort;
double   x,y;
```

Bei Arrays wird hinter dem Namen der Variablen die Anzahl der Elemente angegeben. Der Index startet immer bei 0.

```
int      m[10],i,n[80];  /*  i ist kein Array ! */
double   data[12];
```

Die Definition von *Arrays* und gewöhnlichen Variablen kann beliebig gemischt werden.

Achtung: Ein C-Array „`int m[10];`“ hat genau **10** Elemente, wobei die Indizierung bei 0 beginnt! Das C-Array darf daher mit maximal 9 indiziert werden (`m[9]`)! C-Compiler bieten üblicherweise keine Bereichsüberprüfung, auf `m[10]` kann daher i.a. problemlos zuegriffen werden, der Wert ist aber undefiniert und kann sich unvorhersehbar ändern!

Variablen können entweder **außerhalb** von Funktionen oder in Funktionen selbst, jeweils zu Beginn eines *beliebigen Programmblockes* – näheres siehe Abschnitt 3.1 (Speichertypen von Variablen) – definiert werden. Ein Programmblockes ist ein Teil einer Funktion, der mit geschwungenen Klammern „{“, „}“ eingeleitet bzw. beendet wird, analog dem Funktionsrumpf (siehe 1.4.1). Innerhalb eines Programmblockes definierte Variablen sind nur in diesem Block gültig.

In C++ können Variablen in einer Funktion an jeder beliebigen Stelle definiert werden.

## 1.3 Initialisierung von Variablen

Sofort mit der Definition einer Variablen kann ihr ein Wert zugewiesen werden. Dabei können auch konstante Ausdrucke, d.h. solche, die bereits der Compiler auswerten kann, verwendet werden.

```
int     i = 12, j = 3, k = 97,l;
double  x = 127.45/89.1;
```

In C++ kann auch die sog. Konstruktorsyntax zur Initialisierung verwendet werden, die einem Funktionsaufruf entspricht:

```
int     i(12), j(3), k(97), l;
double  x(127.45/89.1);
```

Initialisierungsaufrufe von Funktionen wie `sin()` oder `cos()` sind in C nicht gestattet, in C++ ist derartiges möglich.

Bei Arrays können die ersten paar Elemente, wenn gewünscht auch alle, in geschwungenen Klammern angegeben werden:

```
double  data[12] = { 0.2, 0.4, 0.5, 0.6, 0.7, 0.99, 10.0 };
```

Unvollständig initialisierte Arrays werden mit 0 aufgefüllt.

Bei initialisierten Arrays kann auch die Dimensionsangabe weggelassen werden, die Größe des Arrays entspricht dann der Anzahl der Initialisierungselemente.

```
double  data[] = { 0.2, 0.4, 0.5, 0.6, 0.7, 0.99, 10.0 };
```

Bei *char*-Arrays (Strings) wird ein abschließendes 0-Byte angehängt, das das Ende der Strings signalisiert. Eine zusätzliche Längenangabe bei Strings existiert nicht. Auf diese Art können Strings nahezu beliebig lang sein und sind keiner Grenze unterworfen.

## 1.4 Funktionen

### 1.4.1 Parameterdefinition

Funktionen werden wie Variablen definiert, nur daß zusätzlich runde Klammern angegeben werden müssen. Diese können die Funktionsparameter enthalten:

```
int     i;                     /* Definition einer Variablen */

int     f(int i,int j,double d) /* Definition einer Funktion */
{
}
```

Werden keine Parameter angegeben, kann die Funktion mit **beliebig vielen** Parametern aufgerufen werden und die Kontrolle der richtigen Parameter durch den Compiler fällt weg. Da die Auswertung der Funktionsparameter dann allerdings etwas komplizierter ist, wird dies erst später im Abschnitt 8.2 behandelt.

In C++ hingegen ist eine Funktion ohne Parameterangabe – wie intuitiv zu erwarten wäre – parameterlos. Dies ist eine – historisch bedingte – Inkompatibilität zwischen C und C++ , die sich allerdings nur selten auswirkt.

### 1.4.2 Rückgabewert (`return`)

Der Typ einer Funktion ist ihr Rückgabewert, dieser wird mit dem Schlüsselwort *return* angegeben.

```
int     add23(int i)
{
        return i+23;
}
```

Wird **kein** Rückgabewert angegeben, so ist der Rückgabewert vom Typ *int*.

### 1.4.3 `void`

Soll eine Funktion keine Parameter haben, wird in der Parameterliste *void* angegeben (oder einfach nichts in C++ ). „void“ bedeutet „nichts“, die Funktion ist parameterlos:

```
double  pi(void)
{
        return 3.14159265;
}
```

Eine Funktion, die „nichts“ zurückliefern soll, hat als Rückgabewert ebenfalls den Typ *void*. Um die Funktion zu verlassen, kann *return* ohne Parameter verwendet werden; am Ende einer *void*-Funktion ist dies aber nicht explizit notwendig.

```
int     icounter = 0;
void    count(int increment)
{
        icounter = icounter + increment;
}
```

### 1.4.4 Aufruf

Beim Aufruf einer Funktion müssen immer Klammern verwendet werden, auch dann, wenn die Funktion keine Parameter hat. Dadurch werden Funktionsaufrufe immer eindeutig von Variablen unterschieden:

```
double  d;
        d = pi();         /*  Aufruf der Funktion pi() */
        d = epsilon;      /*  Auswertung der Variablen epsilon */
```

Es ist kein Fehler, eine Funktion ohne Klammern auszuwerten, dann allerdings wird nicht die Funktion aufgerufen, sondern ihre Speicheradresse ermittelt, d.h. anstelle des Funktionsaufrufes wird ein mehr oder weniger sinnvoller Zahlenwert eingesetzt.

### 1.4.5 `main()`

Unter allen C-Funktionen gibt es eine spezielle, die dadurch ausgezeichnet ist, daß sie beim Programmstart zuerst aufgerufen wird. Dies ist normalerweise die Funktion `main()`. Ein C-Programm sieht also so aus, daß der Programmierer eine Funktion mit dem Namen „`main()`“ schreibt und von dort aus ggf. andere Funktionen aufruft. `main` ist **kein** Schlüsselwort und es ist reine Konvention, daß diese Funktion zu Beginn aufgerufen wird. Dabei ist es völlig gleichgültig, *wie* diese Funktion definiert wird (Parameter, Rückgabewert), sie wird auf jeden Fall aufgerufen. Dennoch sollte man sich dabei an gewisse Regeln halten.

### 1.4.6 Rückgabewert von `main()` - Errorlevel

Von der Funktion `main` wird erwartet, daß der Benutzer sie so definiert, daß sie einen *int* als Funktionswert liefert:

```
int     main()
{
          ...
        return 0;
}
```

Der Rückgabewert wird dann als *Errorlevel* des Programmes weiterverwendet. Obiges Programm liefert also den Errorlevel 0. Diese Information kann dann außerhalb des Programmes, z.B. in Batch-Dateien, weiterverwendet werden. Eine bisweilen verwendete Konvention ist:

| | |
|---|---|
| 0 | fehlerfreier Programmablauf |
| 1 | ein vorhersehbarer Programmfehler ist aufgetreten (z.B. falsche Benutzereingabe) |
| 2 | ein mittelschwerer Programmfehler ist aufgetreten (z.B. Datei nicht gefunden) |
| 3 | ein unvorhersehbarer, fataler Programmfehler ist aufgetreten (z.B. eine Variable hat einen Wert, den sie eigentlich niemals hätte bekommen dürfen) |

Wie bereits erwähnt, kann die Funktion `main()` auch anders definiert werden, beispielsweise als `void`:

```
void    main()
{
         ...
        return;
}
```

In diesem Fall ist der Errorlevel **undefiniert** und rein zufällig. Normalerweise spielt dies keine Rolle, aber manche Programme reagieren sehr empfindlich auf bestimmte Errorlevels (z.B. „`make`“); „Saubere“ Programme liefern immer einen definierten Errorlevel.

Übungsaufgabe: Schreiben eines Batch-Files, das den Errorlevel eines Programmes abfragt und eine Meldung ausgibt. Zu beachten ist dabei, daß die DOS-Errorlevel-Abfrage einen „Größer gleich“ Vergleich darstellt. Dieses Batchfile könnte z.B. so aussehen:

```
@echo off
test.exe
IF ERRORLEVEL 3 echo "Der Errorlevel ist >= 3 !"
IF ERRORLEVEL 2 echo "Der Errorlevel ist >= 2 !"
IF ERRORLEVEL 1 echo "Der Errorlevel ist >= 1 !"
IF ERRORLEVEL 0 echo "Der Errorlevel ist >= 0 !"
```

Zudem wird erwartet, daß die Funktion `main()` mit bestimmten Parametern definiert wird, die dann die Kommandozeile enthalten. Solange man die Kommandozeile aber nicht auswerten will, ist das nicht nötig und wird daher erst im Abschnitt 8.1 durchgenommen.

## 1.5 Library - Funktionen

Da ein Programm nur selten „aus sich selbst“ besteht, ist es notwendig, Library - Funktionen, wie beispielsweise zur Ein/Ausgabe, einzubinden.

### 1.5.1 Deklaration von Funktionen und `#include`

Um dem Compiler mitzuteilen, daß eine Funktion irgendwo als Libraryfunktion existiert und aufgerufen werden kann, muß sie *deklariert* werden. Dies wird an späterer Stelle in Abschnitt 3.5 ausführlich behandelt. Für das erste genügt es zu wissen, daß mit der `#include` - Anweisung einzelne Funktionen deklariert und damit im Programm verwendet werden können. Mit der `#include` - Anweisung werden die sogenannten Header-Dateien eingebunden, die die notwendigen Deklarationen enthalten. Der Name der Header-Datei wird nach `#include` in spitzen Klammern angegeben (oder in Anführungszeichen - mehr dazu in Abschnitt 3.5).

KLASSISCHES ÜBUNGSBEISPIEL:

```
#include <stdio.h>

int     main()
{
        puts("Hello, world!");
        return 0;
}
```

## 1.6 Standardfunktionen zur Ein/Ausgabe

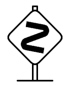
- `puts()` - „Put String“: Ausgabe eines Strings (`char[]`):

```
#include <stdio.h>

int     main()
{
char    text[] = "Hello, world!";
        puts(text);
        return 0;
}
```

- `gets()` - „Get String“: Einlesen eines Strings bis zum Zeilenende

```
#include <stdio.h>

main()
{
char    text[128];
        gets(text);
        puts(text);
        return 0;
}
```

Warnung: Bei `gets()` muß das *char*-Array, das den Eingabetext aufnehmen soll, groß genug sein! Die Funktion `gets()` überprüft **nicht** die Länge der eingegeben Zeichen mit dem zur Verfügung stehenden Speicherplatz!



- `putchar()` - „Put Character“: Ausgabe eines einzelnen Zeichens.

```
#include <stdio.h>

int     main()
{
char    text[] = "123!";
        putchar(text[0]);
```

```
        putchar(text[1]);
        putchar(text[2]);
        putchar(text[3]);
        return 0;
}
```

- `atoi()` - „Ascii to integer“: Mit dieser Funktion kann ein String ausgewertet und einem *int*-Wert zugewiesen werden.

```
#include <stdlib.h>

int     main()
{
int     i;
char    s[] = "123";
        i = atoi(s);

        return i;
}
```

- `printf()` - „print formatted“

  Diese Funktion ist eine der komplexesten, aber auch am meisten gebrauchten Funktion in C. Ihr erster Parameter besteht aus einer Zeichenkette, in der das Format der Ausgabe festgelegt wird; die weiteren, variablen (!) Parameter werden dann je nach dem Formatstring ausgewertet und ausgegeben. Im Formatstring spielt das „%“-Zeichen eine besondere Rolle: In Abhängigkeit von dem auf das „%“-Zeichen folgenden Zeichen werden die Parameter ausgegeben. Die Parameter, mit denen *printf()* aufgerufen wird, müssen daher unbedingt mit dem Formatstring übereinstimmen - dies wird vom Compiler nicht überprüft und liegt zumeist[1] vollständig in der Verantwortung des Programmierers.

  Jedem Zeichencode muß ein Parameter entsprechen, andernfalls funktioniert *printf()* nicht ! Die gebräuchlichsten Codes sind:

| | | | |
|---|---|---|---|
| `%d` | *int* | `%u` | *unsigned* |
| `%ld` | *long int* | `%lu` | *long unsigned* |
| `%f` | *float* | `%lf` | *double* („long float“) |
| `%s` | Zeichenkette | `%c` | *char* |

VERWENDUNG VON PRINTF:

```
int     main()
{
int     a,b;
        printf("Reiner Text...\n");
        printf("Dies ist ein 'int':  %d !\n",23);
        printf("Die Zahl PI ist ungefähr:  %lf \n",3.1415);
        a=4;
        b=5;
```

[1] Manche Compiler bieten hierbei etwas Hilfestellung an.

```
        printf("%d und %d ergeben in Summe %d \n",a,b,a+b);
        return 0;
}
```

## 1.7 Typisierte Konstanten

### 1.7.1 Ganzzahlige Konstanten

Zahlenwerte verhalten sich parallel zu den Typen von Variablen. Eine Zahl (z.B. „1") ist normalerweise vom Typ *int*; wenn eine Zahl nicht mehr durch *int* - angenommen, `int≡short` - darstellbar ist (z.B. 33000, da größer als 32767), wird automatisch der nächste passende Typ genommen (bei einem 16-Bit-Compiler ist 33000 also ein *unsigned*, 100000 ein *long*). Der Typ einer Zahl kann aber auch explizit angegeben werden werden: für *long*-Zahlen wird einfach ein „L" angehängt, für *unsigned* ein „U". Wichtig ist dies bei Vergleichen und Ausdrücken, wenn entweder das Vorzeichen ignoriert werden soll oder mit höherer Genauigkeit gerechnet werden soll (10U ist ein *unsigned*, 20L ein *long*, 30UL ein *unsigned long*). So ergibt der Ausdruck `1000*1000/1000` bei einem 16-Bit Compiler den Wert **16** (!), da `1000*1000` eigentlich zwar `1000000` ergeben sollte, diese Zahl wird aber als *int* behandelt und der Rest der Division durch 65536, respektive 16960, weiterverwendet. Um dies zu verhindern, muß die Multiplikation mit *long*-Werten ausgeführt werden; `1000*1000L/1000` oder `1000L*1000/1000` ergibt korrekt `1000`, das Ergebnis ist vom Typ *long*. Dieses Problem tritt natürlich so nicht auf, wenn *int* vom Compiler als mit *long* identisch verwendet wird. Kritische Situationen entstehen erst dann, wenn ein mit einem solchen Compiler entwickeltes Programm mit einem anderen Compiler übersetzt werden soll.

Hexadezimalwerte werden durch die Zeichenfolge `0x` eingeleitet (z.B. 0xFFFF, 0x01B8), Oktalwerte durch eine führende `0` - aus diesem Grunde ist ein Zahlenwert wie `08` **nicht** zulässig und führt zu einer Fehlermeldung des Compilers!

ÜBUNGSBEISPIEL: Konstanten und einfache Operationen mit Variablen.

```
#include <stdio.h>
main()
{
int     a,b,c,d;
        a = 1000;
        b = 1000;
        c = 1000;

        d = a*b/c;

        printf("Die Rechnung %d*%d/%d ergibt %d !\n",a,b,c,d);
        return 0;
}
```

### 1.7.2 Fließkommazahlen

Sie werden durch ein Komma in der Zahlenangabe oder eine Exponentialangabe unterschieden. Beispiele: `1.0,.8,12.,5E10`. Eine führende 0 vor oder hinter dem Komma ist nicht notwendig. Fließkommazahlen sind generell vom Typ *double*.

Fließkommazahlen, die vom Type *long double* sein sollen, werden mit anschließendem „`L`" gekennzeichnet, z.B.:

`0.7071067811865475244008443621048L` ($\sqrt{2}$). Gerade bei Konstanten hat dies den Vorteil größerer Genauigkeit, da sonst für den Typ *double* überzählige Stellen einfach abgeschnitten würden.

### 1.7.3 Asciizeichen und Zeichenketten (Strings)

Konstanten vom Typ *char* unterscheiden sich von *int*-Konstanten dadurch, daß sie mit einfachem Hochkomma („`'`“) eingeschlossen werden (`'a'`,`'b'`,`'c'` usw.). Eine spezielle Rolle in den *char*-Konstanten spielt der Backslash „`\`“, er leitet eine Zeichenfolge ein, die für bestimmte bei der Bildschirmausgabe relevante Werte steht. So steht

| | | |
|---|---|---|
| `\b` | für | 8 (Backspace, Befehl „ein Zeichen zurück“), |
| `\t` | für | 9 (Tabulator), |
| `\n` | für | 10 (New Line, Neue Zeile), |
| `\r` | für | 13 (Return, an den Zeilenanfang springen), |
| `\'` | für | das einfache Hochkomma selbst, |
| `\x23` | für | das Zeichen mit dem **hexadezimalen** Wert 23 |
| `\23` | für | das Zeichen mit dem **oktalen** Wert 23. |

Es gibt leider keine Möglichkeit, dezimale Zahlenwerte anzugeben.

Konstante Zeichenketten sind aus *char*-Konstanten aufgebaut, aber insgesamt mit doppeltem Hochkomma abgeschlossen.
BEISPIELE:`"abcdef"`, `"Eine Zeile !\n"` oder `"Das Zeichen 0x23 ist '\x23' !"`).
Einfache Hochkommas brauchen nicht mehr mit Backslash kodiert werden - stattdessen muß jetzt das doppelte mit `\"` geschrieben werden (z.B. `"Dies ist eine \"3\"!"`).

Strings können auch aus mehreren einzelnen Strings zusammengesetzt werden (allerdings **nur** bei der Initialisierung) und so über mehrere Zeilen gehen, z.B.:

```
char    text[] = "Dies ist ein "
                 "ausgesprochen langer, "
                 "beziehungsweise über mehrere Zeilen "
                 "reichender Text.";
```

Soll der Text allerdings auch Zeilenumbrüche enthalten, darf man nicht vergessen, an den entsprechenden Stellen ein `\n` einzufügen - obiger Text wird nämlich nur als eine einzige Zeile ausgegeben, und nicht mehrzeilig !

## 1.8 Typkonversionen

Alle Typen sind *implizit* ineinander konvertierbar. So kann einem *int* ein *double*-Wert genauso zugewiesen werden wie einem *char* ein *int*-Wert. Sogar die Zuweisung eines *double*-Wertes an einen *char* ist ohne Probleme möglich (`char c=9.86;`). Wenn ein Wert nicht in einen anderen Typ paßt (Fließkomma → Ganzahl, long → short), werden überzählige Stellen abgeschnitten. Manche Compiler geben bei Fällen, in denen ein Datenverlust erfolgen könnte, beim Übersetzen eine Warnung aus.

Ungefährliche Konversionen sind:

```
double  d = 1, e = 99, g = 512;
long    l = 1024, m = -8192;
int     i = '\n';
```

Gefährliche Konversionen, hier eindeutig mit Datenverlust verbunden, sind:

```
long    l = 1E24;
short   a = 1.056, b = 0.99, c = 5.12, d = 100000L;
char    c = 284, d = -99.76;
```

Ein häufiger Fehler ist es, Rechnungen mit *int*'s durchzuführen und dann einem *double* zuzuweisen:

```
double  ein_drittel = 1/3;
```

In diesem Falle wird die Rechnung selbst mit *int*'s durchgeführt und dann das Ergebnis, nämlich 0, in einen *double* konvertiert und der Variable zugewiesen. Korrekterweise muß der Rechenausdruck zumindest eine Fließkommakonstante aufweisen:

```
double  ein_drittel = 1./3;
double  ein_drittel = 1/3.;
double  ein_drittel = 1./3.;
```

So wird zuerst der verbleibende *int* in einen *double* konvertiert und dann die Rechnung selbst vollständig in *double*'s ausgeführt.

Dasselbe gilt natürlich auch für Rechnungen mit Variablen anstelle von Konstanten.

Neben der teilweise etwas gefährlichen *impliziten* gibt es noch die Möglichkeit der *expliziten* Typkonversion. Dabei wird vor den Ausdruck, der konvertiert werden soll, der gewünschte Typ in runden Klammern geschrieben.

Beispiele:

```
double  ein_drittel = (double)1/(double)3;
int     i = (int)(5.96 + 4.56);
```

Auf diese Weise kann beispielsweise in double-Ausdrücken ein Abschneiden erzwungen werden:

```
double intervall_0_1(double d)
{
        return d-(int)d;
}

/*  Liefert den Rest von a bei der Division durch b */
double fmod(double a,double b)
{
        return a-b*(int)(a/b);
}
```

In C++ kann zur expliziten Typkonversion auch die Syntax eines Funktionsaufrufes verwendet werden:

```
double intervall_0_1(double d)
{
        return d-int(d);
}
```

was allerdings bei Typen, die länger als ein Wort sind (z.B. *unsigned short*), nicht möglich ist. In moderneren C++ -Compilern gibt es die Typecast-Syntax, die an dieser Stelle zwar etwas komplex wirkt, aber aufgrund komplexerer analoger Fälle (es gibt in C++ mehrere Varianten eines Typecasts) ihre Berechtigung hat.

```
double intervall_0_1(double d)
{
        return d - static_cast<int>(d);
}
```

## 1.9 Übungsaufgaben

### 1.9.1 Compilertyp

Eine erste Aufgabe soll darin bestehen herauszufinden, welche Variablen der aktuell verwendete Compiler unterstützt. Konkret sollen die folgende Fragen beantwortet werden:

- Handelt es sich um einen 16- oder 32-Bit Compiler?
- Unterstützt er den Typ *long long* als 64 Bit Ganzzahl?
- Unterstützt er den Typ *long double* als 80 Bit Fließkommazahl?

### 1.9.2 Programmblöcke

Ein Beispiel zum Thema „Programmblöcke“. Welcher Text wird beim letzten *puts()*-Aufruf ausgegeben?

```
main()
{
char    s[] = "Ein Ausgabetext";

        puts(s);

        {
        char    t[] = "Ebenfalls ein Ausgabetext";
                puts(t);

                {
                char    s[] = "Ein lokaler Ausgabetext...";
                        puts(s);
                }

                puts(s);/*  Welcher Text wird hier ausgegeben ? */
        }
        return 0;
}
```

Anmerkung: Zum Einrücken sind unbedingt Tabulatoren, die acht Leerzeichen entsprechen, zu empfehlen, da durch die kompakte Schreibweise von C sonst leicht unübersichtlich tiefe Verschachtelungen enstehen können. Natürlich kann es dann vorkommen, daß der Programmcode über den rechten Bildschirmrand hinausreicht – dies ist dann ein gutes Kriterium, daß die aktuelle Funktion bereits zu komplex ist und besser in kleinere, übersichtlichere Funktionen aufgeteilt werden sollte.

### 1.9.3 Umlaufszeiten der Planeten & Titius-Bode-Reihe

Ein Beispiel mit Eingabe via *gets()*, Zeichenumwandlung via *atoi()* und Ausgabe via *printf()*. Das Programm soll mittels *gets()* die Nummer eines Planeten des Sonnensystems erfragen und dann dessen Umlaufszeit um die Sonne ausgeben. Dazu sind folgende Informationen nötig: Für die Umlaufszeiten zweier Planeten $T_1$, $T_2$ und deren Abstände von der Sonne $a_1$, $a_2$ gilt:

$$\frac{T_1^2}{a_1^3} = \frac{T_2^2}{a_2^3}$$

(3. Keplersches Gesetz). Die Herleitung ist recht einfach, die Beschleunigung eines Planeten ist $m\omega^2 a$, die Gravitationskraft ist $GM_\odot m/a^2$, wobei $G$ die Gravitationskonstante ist ($G = 6.672 \cdot 10^{-11} m^3/kg/s^2$) und $\omega = 2\pi/T$ die Kreisfrequenz. Daraus ergibt sich, daß der Wert

$$\frac{a^3}{T^2} = \frac{GM_\odot}{(2\pi)^2} = \text{const.}$$

für alle Planeten des Sonnensystems eine Konstante ist. Dieser Zahlenwert kann anhand der Erdbahn direkt berechnet werden (Umlaufszeit $T$ ein Jahr, Abstand von der Sonne eine astronomische Einheit, entsprechend ca. 149.6 Mio. km).

Weiters kann zur Berechnung des Abstandes eines Planeten von der Sonne die *Titius-Bode-Reihe* verwendet, eine rein empirische Formel, die auf keiner physikalischen Ursache beruht und

„rein zufällig“ in etwa die korrekten Planetenabstände von der Sonne in astronomischen Einheiten angibt:

$$AE = 0.4 + 0.3 \cdot 2^n$$

Dabei ist $n = -\infty$ für den Merkur (daher bitte diesen Planeten in diesem Übungsprogramm nicht behandeln), sonst 0,1,2,3 … für die restlichen mit freiem Auge sichtbaren Planeten des Sonnensystems. Für die äußersten Planeten gilt die Titius-Bode-Reihe nicht mehr.

Als Zusatzaufgabe könnte gleichzeitig die Masse der Sonne mit ausgerechnet werden, das Ergebnis sollte ca. $2 \cdot 10^{30} kg$ sein.

Zum Vergleich: Die *wahren* Werte sind:

| Planet | n | Sonnenabstand(AE) | Umlaufszeit(Jahre) |
|---|---|---|---|
| Merkur | $-\infty$ | 0.39 | 0.24 |
| Venus | 0 | 0.72 | 0.62 |
| Erde | 1 | 1.00 | 1 |
| Mars | 2 | 1.52 | 1.88 |
| Jupiter | 4 | 5.2 | 11.86 |
| Saturn | 5 | 9.54 | 29.46 |
| Uranus | 6 | 19.18 | 84.01 |
| Neptun | — | 30.06 | 164.79 |
| Pluto | $\approx 7$ | 39.6 | 248 |

### 1.9.4 Anlagekurse

Ein weltlicheres Beispiel: Eingegeben werden soll eine Geldsumme, der Zinssatz, den man in der Bank dafür bekommt und die Kapitalertragssteuer, die der Staat abzieht. Das Programm soll dann die ein Jahr später zur Verfügung stehende Summe berechnen und ausgeben, sowie die Zinsen und Steuern als Zahlenwert, ebenso die effektive Rendite in Prozent.

Weitere Ausbaustufen: Beim An/Verkauf werden jeweils Bankspesen verrechnet, wie sieht das Ergebnis dann aus? Wie verändert sich der Geldwert mit der Inflation?

### 1.9.5 Ausklang

Was tut das folgende Programm? (Nur hier im Text ist das Programm auf zwei Zeilen aufgeteilt, das Original ist aber einzeilig!)

```
main(){char p[]="main(){char p[]=%c%s%c;printf(p,34,p,34,10);}%c";
printf(p,34,p,34,10);}
```

# 2 Einfache Operatoren und Programmstrukturen

## 2.1 `if,else`

Die *if*-Anweisung testet eine Bedingung und führt ggf. die folgende Anweisung oder den folgenden Programmblock aus. Als Bedingung gilt dabei alles, was sich irgendwie in *int* konvertieren läßt (vgl. 1.1.10). Die Bedingung muß mit runden Klammern abgeschlossen sein, dafür entfällt aber die Notwendigkeit, „`then`“ oder ähnliche *if*-abschließende Schlüsselwörter verwenden zu müssen. Der Programmblock folgt unmittelbar auf die Bedingung. Auf den *if*-Block kann noch ein *else* mit entsprechendem Block folgen.

```
if (x>120)
        y=12;

if (x>8)
        y=1;
else
        y=2;

if (x>100)
{
        x=0;
        y=y+1;
}
else
        x=x+1;
```

Ein spezielles *elseif* ist nicht nötig, da der *else* -Block ja selbst wieder eine *if*-Bedingung sein kann.

```
if (val>0)
        sign = +1;

else if (val<0)
        sign = -1;

else
        sign = 0;
```

Folgende Unklarheit kann anfangs leicht auftreten: Ist diese Einrückung die richtige:

```
if (x>100)
        if (y>100) return;
        else
        {
                x=0;
                y=y+1;
        }
```

oder diese ?

```
if (x>100)
        if (y>100) return;
else
{
        x=0;
        y=y+1;
}
```

Auf welches *if* bezieht sich das *else* ? Das nächste Beispiel sollte dies klarstellen:

```
if (x>100)
        if (y>100) return;
        else
        {
                x=0;
                y=y+1;
        }
else
        x=x+1;
```

Wie man hieran sehen kann, können Verwechslungen auftreten, wenn man verschachtelte *if*s verwendet. Obwohl es eigentlich nicht notwendig wäre, ist es in solchen Fällen, denen man nicht sofort sieht, welches *else* zu welchem *if* gehört, günstig, die zweite *if*-Bedingung als eigenen Block zusammenzufassen:

```
if (x>100)
{
        if (y>100) return;
        else
        {
                x=0;
                y=y+1;
        }
}
```

Zwingend wird dies, wenn man etwas ganz anderes will, vergleiche daher Obiges mit der folgenden Zeile:

```
if (x>100)
{
        if (y>100) return;
}
else
{
        x=0;
        y=y+1;
}
```

ÜBUNGSBEISPIEL: Das Programm aus 1.9.3, Umlaufszeiten der Planeten, soll ausgebaut werden, indem auch der Planet Merkur ($n = -\infty$) behandelt werden kann und außerdem die Eingabe überprüft wird. Wenn die Eingabe ungültig ist, d.h. der eingebene Text nicht mit „+“, „-“ oder einer Zahl beginnt, soll das Programm mit einer Fehlermeldung (und Errorlevel> 0) beendet werden. Statt $-\infty$ kann man für den Merkur einfach jede beliebige Zahl kleiner 0 als Eingabe akzeptieren.

Mithilfe einer Reihe von *if*-Bedingungen kann man nun eine Funktion schreiben, die den Planetennamen ausgibt, d.h. eine Funktion des Typs

```
void planet_name(int n)
```

Die Funktion soll dabei mit „kleiner“-Abfragen arbeiten, d.h. wenn der Zahlenwert kleiner als 0 ist, dann Merkur, sonst wenn kleiner 1, dann Venus u.s.w.

Die Berechnung des Sonnenabstandes soll eine eigene Funktion übernehmen, die einen *int* als Parameter enthält und daraus den Abstand berechnet - unter Berücksichtigung des Merkurs. Diese Funktion wäre also vom Typ:

```
double AE(int)
```

Weiters soll der Abstand ebenfalls mit einer eigenen Funktion ausgegeben werden, und zwar in Astronomischen Einheiten, in km und in Lichtzeit. Die Lichtzeit ist diejenige Zeit, die das Licht braucht, um eine Entfernung zurückzulegen und wird aus einer Entfernung $r$ berechnet mittels

$$t = r/c$$

wobei $c$ die Lichtgeschwindigkeit ist (299 792 456 m/s). Der Erdmond beispielsweise mit einer Entfernung von ca. 300 000km ist ca. eine Lichtsekunde entfernt. Die Ausgabefunktion soll dabei selbst entscheiden, in welcher Einheit die Lichtzeit ausgegeben werden soll, d.h. entweder Jahre, Tage, Stunden, Minuten, oder Sekunden anzeigen. Zur Ausgabe der Lichtzeit soll die gleiche Funktion verwendet werden, die die Umlauszeit ausgibt, da es sich ja in beiden Fällen um eine Zeiteinheit handelt, also sind zwei Funktionen nötig:

```
void print_time(double t)
```

und

```
void print_dist(double ae)
```

wobei `print_dist()` die Funktion `print_time()` aufruft.

ZUSATZAUFGABE: Das 3. Keplersche Gesetz gilt auch für Kometenbahnen, allerdings ist als Entfernung nicht die minimale Entfernung von der Sonne einzusetzen, sondern die. sog. große Halbachse. Beim Kometen Hale-Bopp beträgt die Umlaufszeit ca. 3000 Jahre (woraus sich direkt die Länge der großen Halbachse $a$ berechnen läßt), der minimale Abstand von der Sonne (die „Perihel-Distanz“) ist ca. eine Astronomische Einheit. Kometen und Planeten bewegen sich auf Ellipsenbahnen. Den Abstand des Mittelpunktes der Ellipse vom Brennpunkt, in dem die Sonne steht, nennt man die „lineare Exzentrizität“, sie ist die Differenz von großer Halbachse $s$ und der Periheldistanz. Damit kann man nun den maximalen Abstand eines Kometen von der Sonne (die „Aphel-Distanz“) berechnen als Summe von großer Halbachse und linearer Exzentrizität. Somit erweiterte Aufgabenstellung: Eingabe der Umlaufszeit eines Kometen und der Periheldistanz, Ausgabe der maximalen Entfernung des Kometen von der Sonne, wie zuvor in Astronomischen Einheiten, Kilometern und Lichtzeit.

## 2.2 Operatoren

Operatoren spielen in C eine essentielle Rolle, viele Schlüsselwörter aus anderen Programmiersprachen sind in C durch Operatoren ersetzt.

Die gewöhnlichen, von anderen Programmiersprachen her bekannten Operatoren `+,-,*,/` u.a. wurden bereits in den vorhergehenden Beispielen verwendet. Besondere Vorsicht im Umgang mit Operatoren ist geboten, da ein Operator nicht unbedingt eindeutig ist, sondern seine Bedeutung möglicherweise erst aus dem Umfeld erhält. Wesentlich ist es zudem, die **Rangfolge** zu kennen, so hat selbstverständlich „`*`“ eine höhere Priorität als „`+`“, aber bei der Masse an Operatoren kann man leicht den Überblick verlieren; dennoch entspricht die Rangfolge weitgehend dem, was man erwartet und wünscht, sodaß Klammern eher vermieden werden können. In Zweifelsfällen sollte man aber besser zuviel als zuwenig Klammern setzen.

### 2.2.1 Unäre Operatoren

Unäre Operatoren sind solche mit nur einem „Argument“ und haben nach den Klammern die höchste Priorität. Zu ihnen zählen: `+`,`-`,`~` (bitweises NICHT/NOT) und `!` (logisches NICHT/NOT). (Der Unterschied zwischen bitweisen und logischen Operatoren wird in 2.2.4 klar.)
BEISPIELE:

```
int     a, b = 3, error = 0;
        a = +b;
        b = -a;
        a = ~3;
        if (!error)     puts("Keine Fehler aufgetreten.");
```

### 2.2.2 Binäre Operatoren

Neben den gewöhnlichen arithmetischen Operatoren `+`,`-`,`*` und `/` gibt es den *Modulo*-Operator `%`, der allerdings nur für Ganzzahltypen verwendet werden kann, für *float* und *double* muß die Libraryfunktion *fmod* verwendet werden. Die folgenden Bit-Operationen sind verständlicherweise ebenso nur für *int*- und *char*-Typen definiert (nach aufsteigender Rangfolge geordnet):

| Operator | Bedeutung | Beispiel | ergibt |
|---|---|---|---|
| \| | oder / or | 1 \| 2 | 3 |
| ^ | exkl. oder / xor | 1 ^ 3 | 2 |
| & | und / and | 1 & 3 | 1 |
| << | Bits Linksschieben | 1<<5 | $32 \equiv 2^5$ |
| >> | Bits Rechtsschieben | 32>>5 | $1 \equiv 32/(2^5)$ |

WEITERE BEISPIELE:

| Ausdruck | entspricht | ergibt | | | ist ungleich | ergibt | | |
|---|---|---|---|---|---|---|---|---|
| 1>>2+3 | 1>>(2+3) | 1>>5 | ≡ | 0 | (1>>2)+3 | 0+3 | ≡ | 3 |
| 1+3&5 | (1+3)&5 | 4&5 | ≡ | 4 | 1+(3&5) | 1+1 | ≡ | 2 |
| 10\|3&5 | 10\|(3&5) | 10\|1 | ≡ | 11 | (10\|3)&5 | 11&5 | ≡ | 1 |
| 1<<2\|8 | (1<<2)\|8 | 4\|8 | ≡ | 12 | 1<<(2\|8) | 1<<10 | ≡ | 1024 |
| 1<<2&8 | (1<<2)&8 | 4&8 | ≡ | 0 | 1<<(2&8) | 1<<0 | ≡ | 1 |

Gerade die Operatoren `<<` und `>>` können leicht Anlaß zu Irrtümern geben, da sie als quasi „Potenz“-Operatoren einen geringeren Rang haben als z.B. `+` oder `-`. Einige Compiler geben Warnungen aus, wenn die Gefahr einer Verwechslung besteht. In solchen Fällen ist es angebracht, Klammern zu verwenden, auch wenn eigentlich keine Notwendigkeit besteht.

### 2.2.3 Die Zuweisung und die Vergleichsoperatoren

Die Zuweisung liefert den Wert des zugewiesenen Objektes. Der Ausdruck `a=15` kann also weiterverwendet werden: `b=(a=15)`. Dies kann beliebig weit fortgesetzt werden, vorausgesetzt, die jeweiligen Typen sind jeweils konvertierbar (siehe 1.8). Mehrere Variablen können so leicht mit dem gleichen Wert belegt werden: `a=b=c=d=0;`.

Die Zuweisung kann auch als Teil eines Ausdruckes auftreten:

```
b=(a=15)+1;
```

und diesen somit beliebig komplex und unlesbar erscheinen lassen ...

Der Operator `=` ist in **allen** Fällen ein Zuweisungsoperator, also auch in *if*, *while*(2.3.1) und anderen Bedingungsabfragen. Zum Vergleich zweier Werte dient der Operator `==`, weitere Operatoren sind `!=` für Ungleichheit, `>`,`>=`,`<` und `<=` für Größenvergleiche.
Eine korrekte *if*-Bedingung sieht dann so aus:

```
x = 12;
if (x==25)      puts("Variable x hat den Wert 25 !");
else            puts("Variable x ist nicht 25 !");
```

Dieser Programmcode vergleicht völlig korrekt und erwartungsgemäß die Variable `x` mit dem Zahlenwert `25`.

ACHTUNG! Folgendes Programm läuft anders, als es auf den ersten Blick erscheint:

```
x = 12;
if (x=25)       puts("Variable x hat den Wert 25 !");
else            puts("Variable x ist nicht 25 !");
```

Bei dieser *if*-Abfrage wird nämlich der Variablen `x` der Wert `25` zugewiesen und das Ergebnis dieser Zuweisung (`25`) auf Wahrheit geprüft - also wahr, da `25` ungleich 0.

### 2.2.4 Logische Operatoren

Für logische Operationen gibt es die zu den bitweisen Operatoren analogen: `&&` für logisches *Und*, sowie `||` für logisches *Oder*; ein logisches *Xor* existiert nicht. Zur Wiederholung: Jeder *int*-Wert ungleich 0 wird als „wahr" interpretiert, 0 selbst als „falsch". Infolgedessen muß zwischen logischem und bitweisem *Und* bzw. *Oder* unterschieden werden. Zum Beispiel ergibt `1&6` *falsch*, da 0, `1&&6` hingegen *wahr*, da 1 und 6 jeweils wahr sind.

Das Ergebnis einer logischen Operation ist, im Gegensatz zu arithmetischen oder bitweisen, immer 0 oder 1. Die *Und*-Operation wird **vor** der *Oder*-Operation ausgeführt, beide werden **nach** den Vergleichsoperatoren ausgewertet, sodaß in vielen Fällen eine Klammerung unnötig ist.

```
if (x>=0 && x<MaxX || y==23) return;
```

Obwohl diese Schreibweise für den C-Compiler vollkommen ausreicht und auch für erfahrene C-Programmierer übersichtlicher ist, sollte man eigentlich überflüssige Klammern verwenden um in nicht unmittelbar einsichtigen Fällen keinerlei Zweifel aufkommen zu lassen:

```
if ((x>=0 && x<MaxX) || y==23) return;
```

oder sogar:

```
if (((x>=0) && (x<MaxX)) || (y==23)) return;
```

MERKE: Unnötige Klammern sollten nur dann nicht gesetzt werden, wenn die Funktionsweise eines Ausdrucks ohne Nachdenken sofort ermittelt werden kann.

Eine logische Operation wird abgebrochen, sobald sie eindeutig ausgewertet werden kann. Das heißt, bei einem Ausdruck wie „ `if (func1()||func2()) ...` “ wird die Funktion `func2()` nicht aufgerufen, wenn `func1()` bereits *wahr* zurückliefert, weil ja die Gesamtbedingung unabhängig von Funktion `func2()` dann *wahr* ist. Analoges gilt für `&&`.

### 2.2.5 Die Zuordnungsreihenfolge und die Auswertungsreihenfolge

Je nachdem, ob bei *gleichwertigen* Operatoren Klammern von links oder von rechts gesetzt würden, unterscheidet man Operatoren, die *von links zuordnen* (z.B. „+“, da `a+b+c` identisch ist mit `(a+b)+c`, so wie auch alle anderen gewöhnlichen Operatoren) und solche, die *von rechts zuordnen*, wie die *Zuweisungsoperatoren* - z.B. „=“ - (weiteres folgt in 4.1).

Bei assoziativen Operatoren wie `+` oder `*` ist die Zuordnungsreihenfolge irrelevant, da ja `(a*b)*c` identisch ist mit `a*(b*c)`. Sie tritt aber wesentlich in Erscheinung bei nichtassoziativen Operatoren wie `-` oder `/`: `a-b-c` entspricht `(a-b)-c` und natürlich nicht `a-(b-c)`. Auch das erscheint trivial, da man diese Operatoren bereits gewöhnt ist. Das gleiche gilt aber ebenso für den Modulooperator `%`: `a%b%c` entspricht `(a%b)%c` und nicht `a%(b%c)`.

Die Zuordnungsreihenfolge ist **nicht** identisch mit der Auswertungsreihenfolge, bei einem Ausdruck `f()+g()+h()` ist es nicht gewährleistet, daß `f()` vor `g()` und `g()` vor `h()` aufgerufen wird — auch dann nicht, wenn explizit Klammern gesetzt werden. Die Zuordnungsreihenfolge legt nur fest, wie die einzelnen Zahlenwerte kombiniert werden, wann diese ermittelt und die entsprechenden Funktionen aufgerufen werden, liegt ganz im Ermessen des Compilers. So kann er beispielsweise alle Funktionen in einer beliebigen Reihenfolge aufrufen und die Zahlenwerte dann gemäß der Zuordnungsreihenfolge in den Ausdruck einsetzen. Genausogut kann er die Funktionen aber auch erst dann aufrufen, wenn der Zahlenwert wirklich gebraucht wird.

Dies wird sehr wichtig in C++ !

TESTBEISPIEL: Wie wertet der aktuell verwendete Compiler diese beiden Ausdrücke aus:

`f()+g()+h()` und `f()+(g()+h())` ?

Ist diese Vorgangsweise logisch ?

Programmcode, der von der Auswertungsreihenfolge eines Ausdrucks abhängt, ist in höchstem Maße nicht portabel und kann sogar bei ein und demselben Compiler von den Optimierungsparametern abhängen. Dennoch gibt es eine einzige Ausnahme von dieser Regel:

ÜBUNGSFRAGE: Bei welchen beiden Operatoren ist die Auswertungsreihenfolge definiert ?

### 2.2.6 Übungsaufgabe

Eingabe einer Ganzzahl und Ausgeben der Bitstruktur dieses Zahlenwertes. Desgleichen umgekehrt: Eingabe einer Bitstruktur und Ausgabe des Zahlenwertes.

Als weitere optionale Ausbaustufe des Programmes: Eingabe zweier Werte (binär oder dezimal), Anwenden eines binären Operators, Ausgabe des Ergebnisses, binär und dezimal.

## 2.3 Programmschleifen

### 2.3.1 `while`

Mit *while* können Schleifen gebildet werden, die jeweils **am Anfang** getestet werden.
BEISPIEL:

```
unsigned long fakultät(unsigned long i)
{
unsigned long n = 1;
        while(i>0)
        {
                n = n * i;
                i = i - 1;
        }
        return n;
}
```

### 2.3.2 `do-while`

Mit *do-while* können Schleifen gebildet werden, die jeweils **am Ende** getestet werden.
BEISPIEL:

```
char    buf[128];
int     i;
        do
        {
                printf("Bitte die Zahl 10 eingeben:  ");
                gets(buf);
                i = atoi(buf);
        }
        while(i!=10);
```

### 2.3.3 `for` in seiner einfachsten Form

Die *for*-Schleife ist in C bedeutend allgemeiner und damit mächtiger als in anderen Programmiersprachen. Dies wird noch ausführlicher in Abschnitt 4.2 behandelt.

Die übliche *for*-Konstruktion sieht wie folgt aus:

```
int     array[100];
int     i;
        for(i=0;i<100;i=i+1)
                array[i] = i;
```

Die *for*-Konstruktion enthält drei Ausdrücke, der erste enthält den Startwert (Initialisierung), der zweite den Endwert (Abbruchsbedingung) und der dritte den Inkrementwert (Schleifenanweisung). Wird kein Inkrementwert angegeben, dann wird auch die Schleifenvariable nicht erhöht und die Schleife niemals beendet! Die obige Schleife wird genau 100 mal durchlaufen, die Variable `i` nimmt alle Werte von 0 bis 99 an. Nach Verlassen der Schleife hat die Variable den Wert 100 und kann ohne Bedenken weiterverwendet werden.

Der Vorteil des „kleiner"-Vergleiches liegt darin, daß als Obergrenze der *for*-Bedingung genau die Größe eines Arrays angegeben werden kann, ohne sich darüber viele Gedanken machen zu müssen. Zudem kann diese Größe mit *sizeof* automatisch erfragt werden, der Zahlenwert braucht damit nur bei der Definition des Array selbst angegeben werden. Zu beachten ist dabei, daß *sizeof* die Größe des Arrays in Bytes liefert, nicht in Elementen.

```
int     array[100];
int     i;
        for(i=0;i<sizeof(array)/sizeof(int);i=i+1)
                array[i] = i;
```

In C++ kann **in** der *for*-Anweisung die Schleifenvariable definiert werden:

```
int     array[100];
        for(int i=0;i<100;i=i+1)
                array[i] = i;
```

In diesem Fall ist die Variable $i$ nur innerhalb des *for*-Blockes gültig. Vorteil dieser Variante ist, daß die benötigte Variable nur dort existiert, wo sie gebraucht wird, sodaß erstens im Weiteren eine Variable gleichen Namens definiert werden kann, ohne daß dadurch ein Namenskonflikt entsteht, zweitens kann der Compiler den Code so besser optimieren und die Schleifenvariable beispielsweise direkt in einem Prozessorregister ablegen.

### 2.3.4 Übungsaufgabe

Beispiel 1.9.4 (Anlagekurse) soll soweit ausgebaut werden, daß es für einen eingegebenen Zeitraum jährlich die aktuellen Zahlenwerte ausgibt.

## 2.4 Einfache Stringfunktionen

Bei dem Ausdruck

```
char text[128] = "Dies ist ein Text !\n";
```

handelt es sich um eine *Initialisierung*, nicht um eine Zuweisung. Soll einem *char*-Array im weiteren Programmverlauf ein Wert zugewiesen werden, muß explizit die Libraryfunktion `strcpy()` aufgerufen werden:

```
strcpy(text,"Dies ist ein anderer Text !\n");
```

Dabei muß aber darauf geachtet werden, daß im Zielarray (hier: `text`) genügend Platz vorhanden ist, die Libraryfunktion `strcpy()` selbst hat keine Möglichkeit, dies zu überprüfen. Die maximale Länge läßt sich mit `sizeof` ermitteln, die momentane Länge des Textes bis zum abschließenden 0-Byte liefert die Libraryfunktion `strlen()`. Bei char-Arrays, deren Größe über die Initialisierung festgelegt wird, liefert `sizeof` einen um 1 größeren Wert als `strlen()`, da bei `sizeof` das abschließende 0-Byte mitgezählt wird (das ja auch Speicherplatz belegt).

Will man von einen String nicht von Anfang an verwenden, sondern nur von einer bestimmen Position an, so kann man zu der Stringvariablen einfach die Position hinzuzählen (Details und Hintergründe hierzu folgen in Abschnitt 6), also:

```
char    text[] = "Ein Text mit Zahlen:  123456789";

        puts(text+21);          /*  gibt nur 123456789 aus */
```

An einen bestehenden String kann mittels `strcat()` ein weiterer angehängt werden. Wenn eine bestimmte Größe dabei nicht überschritten werden soll, kann die Funktion `strncat()` verwendet werden, bei der zusätzlich angegeben werden kann, wieviele Zeichen maximal angehängt werden sollen. Analog existiert eine Funktion `strncpy()`, die einen String nur bis zu einer bestimmten Länge kopiert. Diese Funktion ist aber mit Vorsicht zu genießen, da unter gewissen Umständen im Zielstring das 0-Byte verloren gehen kann. In diesem Fall sind die üblichen Stringfunktionen nicht mehr anwendbar und können sich sehr zerstörisch auswirken. Daher muß also ggf. das Nullbyte explizit gesetzt werden.

Neben diesen einfachen Stringfunktionen gibt es noch jede Menge komplexerer Werkzeuge (wie strchr(), strstr(), strtok(), strpbrk()), die aber Erfahrung im Umgang mit Zeigern voraussetzt und daher vorerst nicht verwendet werden sollen.

### 2.4.1 Übungsaufgabe

Bei der Programmierung von `cgi`-Anwendungen (Programme, die von einem WWW-Server über das Internet aufgerufen werden) werden die Daten einer Eingabeform über die Environmentvariable `QUERY_STRING` übergeben. Eine solche WWW-Eingabeform in HTML könnte beispielsweise so aussehen:

```
<HTML>
<HEAD>
<TITLE>Eingabeformular</TITLE>
</HEAD>
<BODY>
<H1>Planetensystem</H1>
<FORM ACTION="planet.exe">
Erster Planet:  <INPUT TYPE="TEXT" NAME="START"><BR>
Zweiter Planet:  <INPUT TYPE="TEXT" NAME="ENDE"><BR>
<INPUT TYPE="SUBMIT" VALUE="Berechnen">
</FORM>
</BODY>
</HTML>
```

Nach Bestätigung der Eingabe mit einem WWW-Browser wie Netscape wird das Programm „`planet.exe`“ aufgerufen, die Environment-Variable `QUERY_STRING`, die mit der Funktion

```
getenv("QUERY_STRING")
```

aus <stdlib.h> ermittelt werden kann, enthält dann den Text `START=1&ENDE=5`, d.h. alle Variablen sind hintereinander aufgereiht, jeweils durch das Zeichen „`&`“ getrennt.

Aufgabenstellung: Aus diesem Text sollen die Werte der Variablen `START` und `ENDE` extrahiert werden.

# 3 Speichertypen von Variablen und Modultechnik

## 3.1 Speichertypen von Variablen

In C wird zwischen zwei Speicherbereichen unterschieden, dem *Code-* und dem *Daten*-Segment. Auch wenn Funktionen und Variablen im C-Quellcode in beliebiger Reihenfolge definiert werden, so werden die Funktionen zu einem gemeinsamen Block, dem Code-Segment, zusammengefügt, analog die Variablen zum Daten-Segment. Manche Prozessoren unterscheiden sogar hardwaremäßig zwischen diesen beiden Segmenttypen, sodaß Variablen nicht als Programmcode ausgeführt werden können und Programmcode nicht überschrieben werden kann. Während das Code-Segment keine weitere Struktur hat, läßt sich das Daten-Segment in drei Sparten unterteilen: Der *statische* und der *dynamische* Bereich sowie das *Stack*-Segment. Bei manchen Compilern gibt es auch noch einen eigenen Bereich für konstante Daten, die, entsprechende Hardware vorausgesetzt, vor Schreibzugriffen geschützt werden können (siehe 3.2).

- **Statische Daten** werden zugleich mit ihrer Definition durch einen bestimmten Wert initialisiert (0, wenn nicht anders angegeben), und sind während des gesamten Programmablaufes verfügbar. Wenn ihnen ein Wert zugewiesen wird, behalten sie ihn bei, bis er verändert wird; sie „vergessen“ ihn also nicht. Statische Daten, die speziell vorbelegt werden, sind Teil des fertigen Programmes und vergrößern das Programm auf der Festplatte.

- **Dynamische Daten** müssen speziell angefordert werden. Sie sind dann zu verwenden, wenn man erst im Laufe des Programmes feststellen kann, wieviel Speicher benötigt wird. Einmal angefordert, verhalten sie sich so wie statische Daten, sie können aber auch wieder „vernichtet“ werden, um Platz für andere Daten zu schaffen und sind dann nicht mehr verfügbar. Die dynamischen Daten sind nicht direkt Teil der Sprache C und nur über *Zeiger* zugänglich, daher werden sie erst im Abschnitt 6.8 behandelt.

- **Stack-Daten** sind quasi Teil einer Funktion, obwohl sie sich effektiv in einem anderen Speicherbereich befinden. Sie werden beim Aufruf einer Funktion angelegt und beim Verlassen derselben Funktion wieder vernichtet. Nach dem Ende der Funktion sind sie also nicht mehr zugänglich und haben beim Wiedereintritt einen undefinierten Wert; ihnen sollte also möglichst immer am Anfang einer Funktion etwas zugewiesen werden. Manche Compiler geben eine Warnung aus, wenn eine Stack-Variable ausgelesen wird, ohne daß ihr vorher Wert zugewiesen wurde. Stack-Variablen werden oft auch *automatische* Variablen genannt, da sie mit jedem Funktionsaufruf automatisch erzeugt werden und nach Verlassen automatisch vernichtet werden.

Die **Speichertypen** legen fest, wo eine Variable angelegt wird.

### 3.1.1 Speichertypen von lokalen Variablen

„Lokale Variablen“ sind alle Variablen, die innerhalb einer Funktion definiert werden. Es gibt dabei die Möglichkeiten *auto* und *static*. Lokale Variablen sind normalerweise automatische Variablen, das bedeutet, daß eine Variable im *Stack*-Segment abgelegt wird und diese somit nur innerhalb einer Funktion gültig ist. Um dies explizit auszudrücken kann, muß aber nicht, *auto* angegeben werden.

Das Gegenteil dazu ist *static*, damit wird der entsprechenden Variablen im statischen Datensegment ein Platz reserviert.

```
void f()
{
auto    int     k;      /* eine Stack-Variable  */
static  int     l;      /* eine statische Variable */
        double  x,y,z;  /* ebenfalls automatische Variablen */
static  i = 15;         /* eine statische int Variable, */
                        /* die mit 15 vorbelegt wurde  */
         ...
}
```

Übungsaufgabe: Erfinden eines Demonstrationsprogrammes, in dem der Unterschied zwischen automatischen und statischen lokalen Variablen ersichtlich wird.

### 3.1.2 Speichertypen von globalen Variablen

Globale Variablen werden außerhalb einer Funktion definiert und sind daher nicht mit einer Funktion gekoppelt, sodaß sie von allen Funktionen gleichwertig verwendet werden können. Globale Variablen sind immer statische Variablen. Da automatische Variablen außerhalb einer Funktion nicht möglich sind, ist die Angabe *static* hier unnötig (und würde etwas anderes bedeuten, siehe dazu 3.4.1).

```
int     a;

void    setze_a(int i)
{
        a=i;
}

int     lies_a(void)
{
        return  a;
}
```

### 3.1.3 Initialisierung von statischen Variablen

Eine statische Variable kann in C nur mit einem Wert initialisiert werden, den der Compiler bereits bestimmen kann, denn sie muß zur Zeit der Ausführung des fertigen Programmes bereits von Anfang an den richtigen Wert haben.

```
double pi_4 = 3.141592/4.;
```

Eine solche Variable kann also nicht durch den Aufruf einer Funktion, z.B.

```
double sin_pi_4 = sin(3.141592/4.);
```

initialisiert werden, denn die Funktion `sin()` müßte bereits vor dem Start des Programmes aufgerufen werden. Der Compiler selbst kann und darf diese Funktion nicht von vorneherein auswerten, da die Annahme, die Funktion `sin()` liefere immer den Sinus des Argumentes, auch falsch sein kann - ein Programmierer muß ja auch eigene Funktionen gleichen Namens schreiben können.

In C++ können Funktionen zur Initialisierung statischer Variablen benutzt werden. Das Problem, den Wert der Variablen „rechtzeitig“ zu ermitteln, wurde dabei so gelöst, daß die Initialisierungsfunktionen **vor** dem Aufruf der Funktion *main()* aufgerufen werden. Zum Ausführungszeitpunkt von *main()* sind daher schon alle statischen Variablen mit ggf. benutzerdefinierten Initialisierungsfunktionen korrekt belegt worden.

Die Ausführungsreihenfolge der Initialisierungsfunktionen in C++ ist undefiniert und projektabhängig, d.h. alle Teile eines größeren Programmprojektes, inkl. Art und Gestalt einer Library, können dafür ausschlaggebend sein, wann welche Funktion aufgerufen wird. Die zu initialisierenden Daten sind zum Zeitpunkt der Ausführung der Initialisierungsfunktionen ebenfalls undefiniert (i.a. vor der Initialisierung gleich 0), sodaß der in den Initialisierungsfunktionen vorhandene Programmcode keinesfalls bestimmte Werte für statische Daten voraussetzen darf oder von der Reihenfolge der Initialisierungsfunktionen abhängen darf (außer, ein Initialisator erkennt selbst, ob eine Initialisierung notwendig ist und führt diese, wenn möglich, in einer Regie selbständig durch).

In C++ (und **nur** dort) möglich:

```
void f(int i)
{
static  double  sin_pi_4 = sin(3.141592/4.);
static  int     first_i = i;
         ...
}
```

Die Variablen `sin_pi_4` und `first_i` werden hier nicht beim Programmstart initialisiert, sondern erst mit dem **ersten** Aufruf der Funktion **f()**.
VORSICHT: Die statischen lokalen Variablen werden **ausschließlich** mit dem **ersten** Aufruf der Funktion initialisiert! Das folgende Beispiel verhält sich vollkommen konträr zum vorigen:

```
void f(int i)
{
static  double  sin_pi_4;
static  int     first_i;

        sin_pi_4 = sin(3.141592/4.);
        first_i = i;
         ...
}
```

Während der Aufruf der Funktion `sin()` zum Berechnen von $\sin\frac{\pi}{4}$ mit jedem Aufruf der Funktion **f()** nur ineffizient ist, enthält die Variable `first_i` nicht wie im vorigen Beispiel den Wert des Parameters `i` vom erstmaligen `f()`-Aufruf, sondern *immer* den Wert von `i`!

### 3.1.4 `register`-Variablen

Während die vorhergehenden Speichertypen sehr wohl einen Einfluß auf den Programmablauf haben, ist `register` nur eine spezielle Angabe um die Geschwindigkeit erhöhen zu können. Im Wesentlichen entspricht `register` dem Speichertyp `auto`, nur mit dem Unterschied, daß die Variable sich nicht im Speicher befindet, sondern in einem CPU-Register abgelegt werden soll. Da man aber im allgemeinen mehr Variablen benötigt, als ein Prozessor Register hat, ist es nicht sinnvoll, `register` für alle verwendeten Variablen zu verwenden, da dann der Compiler permanent Variablen ein- und auslagern müßte. Ein guter Compiler findet selbständig heraus, welcher Variablen zur Optimierung in CPU-Registern abgelegt werden können. Die explizite Angabe ist quasi ein „Wink mit dem Zaunpfahl" und sollte nur in kurzen Programmteilen, wie kleinen Schleifen, angewandt werden. Eine globale oder statische `register`-Variable zu definieren, hat natürlich keinen Sinn[2] und ist nicht zulässig.

Auch für Funktionsparameter kann `register` verwendet werden. Dies bedeutet dann, daß die Funktionsparameter nicht auf dem Stack, sondern in Prozessorregistern übergeben werden sollen (Ob der Compiler das dann auch wirklich macht, ist nicht unbedingt gesagt).

### 3.1.5 `volatile`-Variablen

Eine spezielle und sehr selten gebrauchte Zusatzangabe ist `volatile`. Sie bedeutet, daß eine Variable nur in dem für sie definierten Speicherplatz angesprochen wird und nicht intern woanders, z.B. aus Optimierungsgründen in einem CPU-Register, zwischengespeichert werden darf - `volatile` ist also das genaue Gegenteil von `register`. Wichtig ist dies in Multitasking - Umgebungen, wenn mehrere Tasks gemeinsame Variablen verwenden, oder wenn eine Variable in einem Speicherbereich liegt, die für irgendwelche Hardwarebausteine relevant ist. Für den normalen Programmieralltag ist `volatile` ohne Bedeutung - jedoch: **wenn** man *volatile* braucht, dann ist es unverzichtbar. `volatile`-Variablen können sowohl statisch als auch automatisch sein.

Folgende Kombinationen sind (unter anderen) möglich:

```
volatile int j;

void f()
{
int volatile auto a;
auto volatile int b;
volatile static c;
static volatile d;
register volatile e;
         ...
}
```

Übungsaufgabe: Stelle die Beziehung der Schlüsselwörter `auto`, `static`, `register`, `volatile` und des Gültigkeitsbereiches von Variablen (lokal, global) graphisch dar.

[2] Manche Compiler bieten jedoch die Möglichkeit, daß eine Funktion ein spezielles CPU-Register nicht verändert.

## 3.2 Die `const`-Spezifikation

Um Variablen vor unbeabsichtigten Veränderungen zu schützen, können sie als `const` spezifiziert werden. `const` wird als Vorsatz vor dem Variablennamen oder Variablentyp verwendet. Eine `const`-Variable kann nur initialisiert werden:

```
const int i = 56;
```

Jede versuchte Änderung (`i=i+1` etc.) bewirkt einen Fehler beim Compilieren des Programmes.

Während in C *const*-Variablen unveränderliche gewöhnliche Variablen darstellen, können in C++ derartige Variablen vollständig wegoptimiert werden, d.h. an der Stelle ihrer Verwendung im Programmcode wird nicht auf den Speicherplatz einer Variablen zugegriffen um den dortigen Zahlenwert auszulesen, sondern stattdessen direkt der Zahlenwert vom Compiler an die entsprechende Stelle eingesetzt (womit möglicherweise schon ganze Rechenausdrücke vom Compiler zum Übersetzungszeitpunkt ausgerechnet werden können).

## 3.3 `enum`-Typen

Eine weitere Möglichkeit, (allerdings nur ganzzahlige) Konstanten zu definieren, sind die *Aufzählungstypen*. Ein solcher Aufzählungstyp wird mit `enum` erzeugt; die Syntax ist dabei sehr vielfältig. Im einfachsten Fall wird nur eine Reihe von *int*-Konstanten definiert:

```
enum { APFEL, BIRNE, ERBSE, LINSE ... };
```

Der Konstanten „`APFEL`“ wird der Wert 0 zugewiesen, alle folgenden Konstanten erhalten einen jeweils um 1 höheren Wert (also `BIRNE`=1, `ERBSE`=2 etc. ).

Dabei kann aber auch ein bestimmter Wert vorgegeben werden, die Numerierung aller nachfolgenden Konstanten wird dann mit diesem Wert fortgesetzt:

```
enum { APFEL, BIRNE, ERBSE=100, LINSE ... };
```

Die Konstante „`LINSE`“ erhält also den Wert 101. Dabei können auch bereits definierte Konstanten verwendet werden:

```
enum { APFEL, BIRNE, GEMÜSE = 100, ERBSE = GEMÜSE, LINSE ... };
```

Umgekehrt ist **nicht** möglich:

```
enum { APFEL, BIRNE, GEMÜSE = ERBSE, ERBSE = 100, LINSE ... };
```

da ja bei der Definition von „`GEMÜSE`“ die Konstante „`ERBSE`“ noch nicht definiert war.

Zusätzlich zur Definition der Konstanten kann ein Variablentyp (intern immer ein *int*) definiert werden, der dann zur Verwendung mit den zugehörigen Konstanten vorgesehen ist.

```
enum Obst { APFEL, BIRNE, ... };
```

Variablen sind dann vom Typ „`enum Obst`“:

```
enum Obst Obst_variable = BIRNE;
```

Eine Funktion kann dann diesen Typ als Parameter haben

```
void iß(enum Obst welches)
{
        if (welches==APFEL)     puts("Esse einen Apfel.");
        else if (welches==BIRNE)puts("Esse eine Birne.");
        else                    puts("Esse irgendwas.");
}
```

oder als Funktionswert liefern:

```
#include <stdlib.h>

enum Obst Dessert(void)
{
enum Obst Eßbares[] = { APFEL, BIRNE, GEMÜSE };

        return Eßbares[ rand() % ( sizeof(Eßbares) /sizeof(Eßbares[0]) ) ];
}
```

Ein „sattes“ Hauptprogramm könnte dann etwa lauten:

```
main()
{
int     i;
        for(i=0;i<100;i++)
                iß(Dessert());

        return 0;
}
```

Gleichzeitig mit der Definition des `enum`-Typs und der Konstanten können auch Variablen vom entsprechenden `enum`-Typ definiert werden:

```
enum Obst { APFEL, BIRNE } Obst_variable_A, Obst_variable_B = BIRNE;
```

Dabei kann auch die Typendefinition weggelassen werden:

```
enum { APFEL, BIRNE } Obst_variable_A, Obst_variable_B = BIRNE;
```

Die erzeugten Variablen sind dann von einem **namenlosen** `enum`-Typ.

Eine Typendefinition ist, wie eine Variablendefinition, auch lokal möglich; ein `enum`-Typ kann also auch lokal in einer Funktion oder einem Programmblock definiert werden:

```
void    f(void)
{
enum Obst { APFEL, BIRNE, ... };
         ...
}
```

unabhängig davon, ob außerhalb ein Typ oder `enum`-Konstanten gleichen Namens bereits definiert wurden.

Da es sich bei den `enum`-Typen um ganze Zahlen handelt, ist die Zuweisung einer `enum`-Konstanten oder -Variablen an einen `int` möglich. Umgekehrt ist die Zuweisung eines `int`-Wertes an eine `enum`-Variable ein programmtechnischer Fehler, da diese Variable dann Werte annehmen könnte, für die keine zugehörige Konstante existiert. Da dies aber leider in C eine gängige Praxis ist, geben manche Compiler dabei nicht einmal eine Warnung aus, in C++ ist dies jedoch mindestens mit einer Warnung, wenn nicht sogar mit einer Fehlermeldung verbunden, da `enum`-Typen dort als unterschiedliche Typen behandelt werden, auch wenn sie intern als *int* dargestellt werden. Dies soll die Programmsicherheit erhöhen, da so gewährleistet wird, daß `enum`-Typen nur mit den ihnen zugedachten Zahlenwerten bzw. Konstanten arbeiten.

Anmerkung: Nach ANSI-C ist es gestattet, einen `enum`-Typ bei der Definition von Funktionsparametern zu erzeugen, etwa so:

```
void    f(enum { APFEL, BIRNE } Obst_variable)
{
         ...
}

void g()
{
        f(BIRNE);
}
```

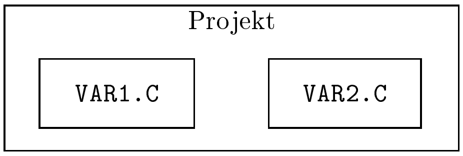
Dies kann aber leicht zu Verwirrungen führen, insbesondere dann, wenn die Konstante „`BIRNE`“ im obigen Beispiel als globale `enum`-Variable bereits definiert wurde. Diese Verwirrung besteht nicht nur auf seiten der Programmierer, sondern auch bei den Compilerbauern (unterschiedliche Auslegung des ANSI-C Standards).

**Solche Konstrukte sollten daher auf jeden Fall vermieden werden.**

## 3.4 Module

Es ist in C möglich, mehrere Programmteile (sogenannte Module) separat zu übersetzen und dann zusammenzufügen. Eine in einem Programmteil global definierte Variable wird dort angelegt; wenn also mehrere Module eine Variable gleichen Namens beinhalten würden, käme es später zu Konflikten, da dies hieße, daß im fertigen Programm die gleiche Variable mehrmals vorhanden sein müßte. Es ist daher nötig, in allen bis auf ein Modul die entsprechende Variable als *extern* zu definieren. Dadurch wird die entsprechende Variable nur **deklariert**, aber nicht **definiert**. Beim Zusammenfügen (Linken) des Programmes muß dann allerdings auch unbedingt dasjenige Modul vorhanden sein, das die Variable wirklich beinhält.

Das gleiche gilt für Funktionen, wobei auch deren Parameterliste angegeben werden muß, anstelle des Funktionsrumpfes (`{ ... }`) steht dann ein einfaches Semikolon. Da es sich so um keine Funktionsdefinition handeln kann, darf *extern* auch weggelassen werden.




Beispiel: Modul „`VAR1.C`“:

```
int     a;                        /*  Definition der Variablen a */
extern  int     lies_a(void);     /*  „extern“ ist hier nicht nötig */

void    setze_a(int i)            /*  Definition der Funktion setze_a() */
{
        a=i;
}
```

Modul „VAR2.C“:

```
extern  int     a;       /* Deklaration der Variablen a */
void    setze_a(int i); /* Deklaration der Funktion setze_a() */

int     lies_a(void)
{
        return a;
}
```

Werden in C bei der Funktionsdeklaration, auch **Funktionsprototyp** genannt, keine Parameter angegeben, so kann diese Funktion mit **beliebigen** Parametern aufgerufen werden; es findet somit auch keine Überprüfung auf falsche Parameter statt. Funktionsprototypen wurden erst mit ANSI-C eingeführt, frühere C-Versionen kannten nur Funktionen mit beliebigen Parametern. In C++ ist eine derartige Funktion parameterlos („`void`“, vgl. 1.4.3).

Um den Programmierern die Arbeit der Erstellung von Prototypen zu ersparen, kann in C eine Funktion auch dann aufgerufen werden, wenn sie **nicht** vorher deklariert wurde. Als Typ wird dann eine Funktion mit beliebigen Parametern angenommen, die einen *int* zurückliefert. Da Parameter bei Funktionen nur in ANSI-C angegeben werden können, bestand in früheren C-Versionen der einzige Sinn einer Funktionsdeklaration im Festlegen des Rückgabewertes, falls es sich dabei nicht um *int* handelt. Aus historischen Gründen können daher auch in ANSI-C nicht deklarierte Funktionen verwendet werden. C++ ist hierbei empfindlicher: Dort führt die Verwendung nicht deklarierter Funktionen je nach verwendetem Compiler entweder mindestens zu einer Warnung oder gar zu einer Fehlermeldung, die eine Funktionsdeklaration erzwingt.

### 3.4.1 Variablen innerhalb eines Modules

Eine Variable, die einmal global definiert wurde, kann kein zweites Mal innerhalb des gesamten Programmes definiert werden. Wenn erwünscht ist, daß eine Variable nur innerhalb eines Modules bekannt ist und somit gegebenfalls in einem anderen abermals definiert werden kann, ohne daß dies Probleme erzeugt, kann eine globale Variable mit dem Vorsatz *static* definiert werden.

Hier hat *static* eine etwas andere Bedeutung als bei lokalen Funktionsvariablen, denn „statisch“ ist eine globale Variable ohnehin. Der Zusatz bedeutet in diesem Zusammenhang, daß die Variable nur innerhalb des momentanen Moduls zugänglich sein soll - sie ist dann vor Zugriffen aus anderen Modulen „geschützt“. Andere Module können globale Variablen gleichen Namens dann ohne weiteres verwenden. Gleiches gilt für Funktionen, die ebenfalls *static* sein können, d.h. nur innerhalb des aktuellen Modules aufgerufen werden können.

Eine globale statische Variable ist somit das genaue Gegenteil einer externen: Sie ist nur innerhalb eines Modules bekannt, eine externe hingegen verweist auf eine außerhalb gelegene.

Modul „VAR1.C“:

```
static  int     a;

static  void Setze_a(int i)
{
        a=i;
}
```

Die Variable `a` ist nur innerhalb des Modules `var1.c` bekannt - jeder Versuch, sie von außerhalb mittels `extern` anzusprechen, schlägt beim Linken fehl. Ebenso kann die Funktion `Setze_a` nicht von außen aufgerufen werden.

ANMERKUNG: Ein C-Modul, das **nur** statische Variablen und Funktionen enthält, ist vollständig von der Außenwelt abgekoppelt und somit absolut unbrauchbar ...

### 3.4.2 Lokale externe Variablen

Es ist kein Widerspruch, eine lokale externe Variable oder Funktion zu haben. Eine externe Variable oder Funktion, die nicht global, sondern innerhalb einer Funktion deklariert wird, ist dann nur innerhalb dieser Funktion bekannt, obwohl sie sich außerhalb des momentanen Moduls befindet. Lokale externe Variablen sind in dem Modul, in dem sie definiert wurden, immer global und dürfen dort natürlich nicht statisch sein.

```
void f()
{
extern  int b;          /* eine lokale externe statische Variable */
int     lies_a(void);   /* eine lokal deklarierte externe Funktion */
         ...
}
```

ÜBUNGSAUFGABE: Erfinden einer Situation, in der eine lokale Deklaration sinnvoll ist.

### 3.4.3 „forward"-Deklaration

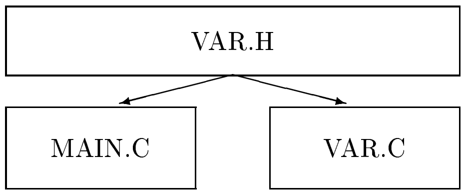
Eine externe Variable/Funktion muß sich nicht unbedingt außerhalb des aktuellen Moduls befinden, mithilfe von „*extern*" kann man auch anzeigen, daß die entsprechende Variable/Funktion erst später im Programmtext definiert wird, dennoch aber bereits vorher verwendet werden soll. Es ist kein Fehler, wenn auf eine *extern*-Deklaration, unmittelbar oder nicht, eine Definition folgt.

## 3.5 Headerdateien

Oft kommt es vor, daß eine besonders praktische Funktion von mehreren Modulen aus aufgerufen werden soll. Eigentlich müßte dann deren Funktionsprototyp in jedem dieser Module vorhanden sein. Es gibt aber einen einfacheren Weg: Die Headerdateien.

Alle zu einem bestimmten Modul gehörenden Deklarationen können in einer separaten Datei zusammengefaßt werden, der üblicherweise die Endung „`.h`" gegeben wird. Mit der `#include`-Anweisung kann diese dann zur Compilierzeit als Teil des C-Modules eingelesen werden, die Headerdatei erscheint dem Compiler als Teil des C-Programmes selbst. Auf diese Art ist gewährleistet, daß alle C-Module die gleiche Deklaration erhalten. Wird in dem Modul, das die eigentliche Definition enthält, die Header-Datei includiert, entsteht eine zusätzliche Kontrolle durch den Compiler, da Deklaration und Definition ja übereinstimmen müssen.

Der Dateiname kann entweder mit spitzen Klammern („`<`" und „`>`") oder doppelten Anführungszeichen angegeben werden, je nachdem, wo die Datei gesucht werden soll. Bei der Version mit doppelten Anführungszeichen wird die Datei erst im aktuellen Verzeichnis gesucht, wenn sie dort nicht vorhanden ist, wird der *Includepfad* „abgegrast", der durch die Compilereinstellungen definiert ist. Mit Spitzklammern wird *nur* der Includepfad abgesucht, lokale Headerdateien werden ignoriert. Diese Version wird daher üblicherweise für System-Headerdateien verwendet, die die Deklarationen von Libraryfunktionen beinhalten, die erste Variante für vom Programmierer erstellte Headerdateien.




BEISPIEL: Headerdatei „VAR.H“:

```
extern  int     lies_a(void);
extern  void    setze_a(int i);
```

Modul „MAIN.C“:

```
#include "var.h"        /*  „Interface“ zu var.c */

main()
{
        setze_a(5);
        printf("a hat den Wert %d !\n",lies_a());
        return 0;
}
```

Modul „VAR.C“:

```
#include "var.h"        /*  nur zur Typüberprüfung */

static  int a;          /*  a ist aus main.c nicht direkt ansprechbar */

void    setze_a(int i)
{
        a = i;
}

int     lies_a(void)
{
        return a;
}
```

Man kann Module als eine Art „Black Box“ ansehen, auf die von außen mit einem Interface in Form der Headerdatei zugegriffen wird. Was innerhalb des Moduls geschieht ist von außen nicht sichtbar, etwaige Änderungen oder Verbesserungen werden im Hauptprogramm sofort wirksam, ohne daß dort etwas geändert werden müßte.

Es kommt zuweilen vor, daß sich eine Includedatei weder im momentanen Pfad, noch direkt im Includepfad, sondern in einem weitern Unterverzeichnis befindet, zum Beispiel befindet sich die Standard-Includedatei stat.h üblicherweise im Unterverzeichnis sys. Diese Datei kann nun einfach mit #include <sys/stat.h> angesprochen werden.

## 3.6 Übungsaufgabe: „Dateiverschlüsselung“

Unter Verwendung der in Aufgabe 2.2.6 (Ein/Ausgabe von binären Zahlen) erstellten Funktionen

```
int     decode(const char bits[], const char bit0, const char bit1);
void    encode(const int value,
                char bits[], const char bit0, const char bit1);
```

soll ein Programm geschrieben werden, daß alle von der Standardeingabe gelesenen Zeichen in eine Reihe von Binärzahlen umwandelt und ausgibt:

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

main()
{
int     c;
        while( (c=getchar()) != EOF)
        {
        char    bits[9];        /*  8 Zeichen + 0-Byte */
                encode( c, bits, '0', '1');
                printf("%s",bits);
        }
        return 0;
}
```

Das Programm kann dann benutzt werden, um eine Datei in eine „unlesbare“ Variante in Form von Binärzeichen zu bringen, indem das compilierte Programm unter DOS mit der Ein/Ausgabeumleitung aufgerufen wird:

```
C:\> encode.exe < encode.c > encode.001
```

Natürlich wird dazu noch ein zweites Programm benötigt, das diese Codierung wieder rückgängig machen kann. Beide Programme sollen dabei das gleiche Modul verwenden, in dem die Codierungsfunktionen enthalten sind.

Eine derartige „Verschlüsselung“ ist allerdings noch recht einfach zu „knacken“, da es relativ einfach zu erkennen wäre, daß nur die entsprechenden Bits wieder in Ascii-Zeichen zurückzukonvertieren wären. Etwas schwieriger wird es allerdings schon, wenn die Zeichen bei der Konvertierung mit einem bestimmten Zeichen `XOR`-verknüpft werden. Die `XOR`-Operation mit einem konstanten Zahlenwert dreht ja alle dort gesetzten Bits um, eine zweimale `XOR`-Operation mit dem gleichen Zahlenwert ergibt wieder den Originalwert. Daher eignet sich eine solche zusätzliche Operation hervorragend zu einer weiteren Verschlüsselung. Um die Sache noch etwas trickreicher zu machen, kann das Verschlüsselungszeichen bei jedem Aufruf der Verschlüsselungsfunktion geändert werden, beispielsweise könnte man dabei alle Zeichen des Namens des Programmierers durchgehen und dann wieder von vorne anfangen. Es muß nur gewährleistet sein, daß die Entschlüsselungsfunktion genau den gleichen Vorgang in umgehrter Reihenfolge ausführt.

# 4 Komplexe Operatoren und Programmstrukturen

## 4.1 Die Zuweisungsoperatoren

Die Zuweisungsoperatoren sind eine praktische Erweiterung der bereits vorhandenen Operatoren und bringen programmtechnisch im Grunde nichts Neues. Ihr Vorteil liegt darin, gewisse Ausdrücke einfacher und effizienter schreiben zu können. Relevant wird dies allerdings, wenn Funktionen anstatt von Variablen verwendet werden; diese werden dann seltener aufgerufen.

Die Zuweisungsoperatoren bestehen entweder aus dem Gleichheitszeichen selbst (2.2.3) oder dem Gleichheitszeichen in Verbindung mit einem binären Operator (2.2.2). Das Ergebnis der binären Operation wird dann direkt dem linken Operanden zugewiesen. So läßt sich `a=a+b` als `a+=b` schreiben. Der gesamte Ausdruck läßt sich wiederum verwenden, beispielsweise: `a+=b+=c;` , in konventioneller Schreibweise: `b=b+c; a=a+b;` . Die Zuweisungsoperatoren ordnen offensichtlich von **rechts** zu, denn `a=b=1` entspricht `a=(b=1)`, ebenso wie auch `a+=b+=1` ≡ `a+=(b+=1)` (vgl. 2.2.5). Folgende Zuweisungsoperatoren sind in C bekannt:

`+=`, `-=`, `*=`, `/=`, `%=`, `&=`, `|=`, `^=`, `>>=` und `<<=`

Damit lassen sich beispielsweise `for`-Schleifen eleganter formulieren:

```
int     i;
        for(i=0; i<100; i+=10)
        {
                ...
        }
```

Für die Addition (Subtraktion) um 1 kann der *Inkrement*(*Dekrement*)- Operator `++`(`--`) eingesetzt werden; `a++` ist identisch mit `a+=1`. Ein feiner Unterschied, der manchmal relevant sein kann, besteht darin, ob `a++` oder `++a` geschrieben wird (*Postfix*- oder *Prefix*-Inkrement): Im ersten Fall wird zuerst a zugewiesen und dann inkrementiert, im zweiten wird erst a inkrementiert, also das ursprüngliche a + 1 zugewiesen.

```
int     a,b;
        a=2;
        b=a++;          /* b erhält den Wert 2, a den Wert 3. */

int     a,b;
        a=2;
        b=++a;          /* a und b erhalten den Wert 3. */
```

Zumeist werden die Increment/Decrement - Operatoren in Verbindung mit `for`-Schleifen gebraucht:

```
        for(i=0; i<100; i++)
        {
                ...
        }
```

Ein bisweilen ebenfalls zu findendes Konstrukt ist das folgende:

```
        b &= ~((1<<12)-1);      /* Alignment auf Vielfache von 2^12 */
```

Aufgrund der vielfältigen Möglichkeiten kann man mithilfe der Zuweisungsoperatoren leicht Ausdrücke erfinden, die nicht definiert sind:

```
b = a++ + a;
```

ist nicht definiert, da der Postfixoperator nur festlegt, daß die Variable `a` nach der Auswertung erhöht wird, aber nicht, ob diese sofort oder nach Ende des gesamten Ausdruckes stattfindet. Zur Wiederholung: **Die Zuordnungsreihenfolge ist nicht die Auswertungsreihenfolge!** Die Auswertungsreihenfolge ist bei den meisten Operatoren nicht definiert.

Ein Ausdruck ist meistens dann undefiniert, wenn ein und dieselbe Variable darin mehrmals verändert wird, da die Reihenfolge dieser Veränderungen nicht festgelegt ist. Jeder Ausdruck, der von der Auswertungsreihenfolge abhängt, ist nicht portabel und kann sogar beim gleichen Compiler von den Optimierungseinstellungen abhängen.

Um nicht in Begriffsverwirrungen mit Definition-Deklaration zu kommen, sollte man eigentlich nicht von definierten und undefinierten Ausdrücken sprechen, den programmieren läßt sich beides. Ein besseres Wort für einen Ausdruck, der einen voraussagbaren Wert liefert, wäre „wohldefiniert“.

Natürlich sind auch „wilde“ Konstruktionen möglich wie im folgenden Beispiel. In Programmen, die auch jemand anderer lesen können soll, muß man sich daher gegebenfalls auf Wiederverstehbares beschränken.

Übungsaufgabe: Wie lauten die folgenden Ausdrücke in konventioneller Schreibweise?

```
b>>=a++;        c+=b*=a++;     c*=b+=++a;     b^=a-=++c;
b=(a=b-a)+b;    b=(a=b)-a+b;   b=a=b-a+b;     b-=a+=b-a+b;
b>>=++a+1;      b|=c++-++a;    b&=++c+--a;    b&=!(c++&a);
a+=b+++-c;      a+=++b+--+c;   a-=--b-++-c;   c+=+a++ + ++b;
```

Dabei wird auffallen, daß einer dieser Ausdrücke nicht kompilierbar ist, derjenige nämlich, für den man auch keine konventionelle Schreibweise finden kann. Man muß also unterscheiden zwischen solchen Ausdrücken, denen etwas zugewiesen werden kann, sogenannte *lvalues* ( **L**inks stehende Werte), und solchen, die nur einen Wert liefern können, sogenannten *rvalues*, ( **R**echts stehende Werte). Nicht jeder *rvalue* ist ein *lvalue*, aber jeder *lvalue* kann als *rvalue* verwendet werden - oder, anders ausgedrückt: Jeder Ausdruck kann ausgelesen werden, aber nicht jeder kann beschrieben werden.

Ein weiterer Ausdruck ist nicht wohldefiniert. Welcher?

## 4.2 Allgemeines `for`

In 2.3.3 wurde die `for`-Schleife in Analogie zur `for`-Konstruktion in anderen Programmiersprachen vorgestellt. Die `for`-Schleife an sich ist aber in C viel allgemeiner und hat mit dem, was normalerweise als `for`-Schleife bezeichnet wird, kaum etwas gemeinsam — die in 2.3.3 dargestellte Konstruktion ist nur ein Spezialfall. Üblicherweise wird *for* als Zählschleife verwendet (`for(i=0;i<100;i++) ...` ), es ist aber keineswegs erforderlich, daß sich die in `for` enthaltenen Anweisungen auf dieselbe Variable beziehen.

Eine allgemeine `for`-Anweisung enthält drei Ausdrücke, die keineswegs irgendwie in Beziehung zueinander stehen müssen:

| Die Startanweisung, die Schleifenbedingung und die Schleifenanweisung. |
|---|

**Jeder** dieser drei Ausdrücke ist optional.

Die Startanweisung kann genausogut vor `for` stehen und dann in der `for`-Anweisung selbst weggelassen werden, aus Gründen der Übersichtlichkeit wird sie jedoch meist als Teil der `for`-Anweisung betrachtet. Gleiches gilt für die Schleifenanweisung: im Prinzip kann die Schleifenanweisung in den folgenden Programmblock hineingezogen werden und durch eine Leeranweisung ersetzt werden (und auch umgekehrt). Gelegentlich kommen derartige Konstrukte vor, was aber

aus Gründen der Lesbarkeit besser vermieden werden sollte. Die Schleifenanweisung wird jeweils am **Ende** der `for`-Schleife ausgeführt. Die Schleifenbedingung funktioniert analog zu `while`: solange sie ungleich 0 ist, wird die `for`-Schleife ausgeführt. Auch sie kann weggelassen werden - es wird dann immer *wahr* angenommen und die Schleife wird niemals beendet. Insgesamt bietet `for` keine Vorteile gegenüber `while`, der einzige Nutzen liegt in der Zusammenfassung von Start-,Ende- und Schleifenanweisung zu einer einzigen, kompakten Anweisung.

VERGLEICH ZWISCHEN `for` UND `while`

```
for(i=0;i<100;i++)
 ...
```

```
i = 0;
while(i<100)
{
 ...
    i++;
}
```

Das Konzept der *for*-Schleife ist also in C bedeutend allgemeiner als in anderen Programmiersprachen.

Das Fakultätsbeispiel aus 2.3.1 läßt sich auch folgendermaßen schreiben:

```
unsigned long fakultät(unsigned long i)
{
unsigned long n;
        for(n=1;i>0;n*=i--);
        return n;
}
```

Dies ist wohl die kürzestmögliche (und damit unverständlichste, aber auch effizienteste) Schreibweise ...

VORSICHT: Die `for`-Schleife selbst ist im obigen Beispiel **leer** und besteht nur aus einem Semikolon (sogenannte Null-Anweisung) ! Da dies leicht übersehen werden kann, ist folgende Schreibweise dringendst anzuraten:

```
 ...
for(n=1;i>0;n*=i--)
        ;
 ...
```

So nämlich fällt sofort auf, daß die `for`-Schleife leer ist und die eigentliche Aktion in der Schleifenanweisung stattfindet.

Eine relativ oft gebrauchte Anwendung, mit der eine Endlosschleif erzeugt wird, ist die **leere** `for`-Anweisung (`for(;;)`). `while(1) { ... }` ist nicht üblich und manche Compiler generieren dabei angeblich schlechten Code (d.h. sie testen jedes Mal die Konstante 1 auf Wahrheit ... ).

## 4.3 Der Sequenzoperator

Jede Anweisung wird in C mittels Semikolon abgeschlossen. In Ausdrücken kann in C der Beistrich „`,`“ verwendet werden um mehrere Ausdrücke in einem zu schreiben. Der Wert und Typ des gesamten Ausdruckes ist derjenige des letzten Ausdruckes, die Unterausdrücke werden dabei von links nach rechts ausgewertet:

```
d = (a+=b, b*=c, c/=a);
```

Die Typen aller verwendeten Ausdrücke müssen dabei ineinander konvertierbar sein. Ist einer der Unterausdrücke ein `void`-Ausdruck (z.B. der Aufruf einer Funktion, die `void` als Rückgabewert liefert), dann ist der gesamte Ausdruck `void` und kann keiner Variablen zugewiesen werden. Umgekehrt kann aber einer der Ausdrücke einen Wert liefern, da jeder beliebige Wert nach `void` konvertiert werden kann (d.h., er wird nicht ausgewertet). Dies kann verwendet werden, um mehrere Ausdrücke zusammenzufassen, **ohne** dafür einen eigenen Block erstellen zu müssen:

```
if (x<0) puts("Illegaler x-Wert !"), x=0;
```

In diesem Beispiel ist `puts` zwar keine `void`-Funktion und liefert einen `int`-Wert, es funktioniert aber ebenso mit einer `void`-Funktion.

Oft wird der Sequenzoperator in Verbindung mit der `for`-Schleife gebraucht:

```
for(i=0,j=99;i<100 && j>=0;i++,j--)
```

(`i` zählt aufwärts, `j` zählt abwärts)

Um bei Funktionsaufrufen den Sequenzoperator von den Funktionsargumenten zu unterscheiden, **müssen** hierbei Klammern verwendet werden:

```
func(i, (j = 1, j + 4), k);
```

Die Funktion `func(int,int,int)` wird mit **drei** Argumenten aufgerufen, das zweite Argument ist 5.

ÜBUNGSAUFGABE: Was geschieht im folgenden Beispiel? Wie kommt das Ergebnis (der Wert von `y`) zustande?

```
#include <stdio.h>

int     main()
{
int     x = -1,y;

        y = puts("Zuweisung von x an y!"), x++;

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

        return 0;
}
```

Was ändert sich, wenn

```
        y = (puts("Zuweisung von x an y!"), x++);
```

geschrieben wird?

## 4.4 break

Bei der unendlichen `for(;;)`—Schleife stellt sich dann natürlich die Frage, wie sie abgebrochen werden kann.

Dazu dient das Schlüsselwort `break`. Hiermit kann jede beliebige Schleife (`while`, `do` ... `while`, `for`) verlassen werden. Bei verschachtelten Schleifen wird dann die nächstäußere weiter ausgeführt.

```
int     i;
        for(i=0;i<10000;i++)
        {
static  unsigned long old_fak = 1;
        unsigned long fak = fakultät(i);
                if (fak<old_fak) break;
                old_fak = fak;
        }
        printf("Die maximal berechenbare Fakultät ist %d !\n",i);
```

MAN BEACHTE: Da die Variablen `old_fak` und `fak` nur innerhalb der `for`-Schleife gebraucht werden, sind sie auch nur dort gültig. Abgesehen davon, daß die beiden Namen weiter unten auch in anderer Bedeutung verwendet werden können, läßt diese Methode dem Compiler die Möglichkeit, die Rechnung mit beiden Variablen besser zu optimieren, denn eine lokale Optimierung ist immer einfacher zu erreichen als eine globale.

## 4.5 continue

Ähnlich wie `break` funktioniert das Schlüsselwort `continue`, das die Ausführung der Schleife ebenso abbricht, sie aber nicht verläßt, sondern wieder an den Anfang der Schleife springt. Der restliche Teil zwischen `continue` und dem Ende der Schleife wird einfach ignoriert.

Besonders praktisch ist dies beispielsweise bei mathematischen Ausdrücken wie dem folgenden:

$$\sum_{i=0}^{N} \sum_{\substack{j=0 \\ j \neq i}}^{N} \cdots$$

Das sieht in C dann so aus:

```
int     i,j;
        for(i=0;i<=N;i++)
        for(j=0;j<=N;j++)
        {
                if (i==j) continue;
                 ...
        }
```

## 4.6 goto

... und nicht unerwähnt bleiben darf, daß es auch in C ein `goto` gibt. `goto` ist jedoch eine Brachiallösung und kann gerade in C durch `break` und `continue` meist umgangen werden. Dennoch: Bevor unzählige Flag-Variablen eingesetzt werden, ist es besser, ein `goto` zu verwenden, denn obwohl es vielleicht nicht „schön" ist, so ist es doch wenigstens verständlich (vgl. mit obigen `for`-Konstruktionen). Zudem: Ein *goto* hat keinen negativen Einfluß auf die Geschwindigkeit des Programmes, sondern bildet nur händisch etwas nach, was der Compiler/die Programmiersprache (noch) nicht bietet.

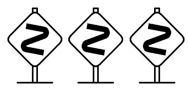
Als Sprungziel muß ein „Label" definiert werden, das dann mit `goto` angesprungen werden kann:

```
         ...
        for(y=0;y<1000;y++)
        for(x=0;x<1000;x++)
        {
                if (kbhit()) goto terminate;
                putpixel(x,y,fraktale_Berechnung(x,y));
        }
terminate:
         ...
```

## 4.7 Funktionsübergreifendes goto

Für besonders ausgefeilte Programmiertricks bietet C standardmäßig die Libraryfunktionen `setjmp()` und `longjmp()` an. Eine genauere Beschreibung findet sich i.a. in der jeweiligen Online-Hilfe des verwendeten Compilers, hier soll nur der prinzipielle Nutzen beschrieben werden.

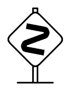
Mit diesen Funktionen ist es möglich, ein „`goto`" über mehrere Funktionen hinweg zu realisieren. Das kann man beispielsweise verwenden, um mehrere Aufgaben quasi-parallel auszuführen, indem die jeweils zuständigen Funktionen untereinander hin- und herspringen. Genau dies geschieht normalerweise in Multitasking-Betriebssystemen, nur daß dort das Betriebssystem bzw. die Hardware selbst den Wechsel ausführt. Die zu einer Funktion parallel ausgeführte(n) Funktion(en) bezeichnet man auch als Coroutinen. Auch in echten Multitaskingumgebungen können sie sinnvoll sein, da der Taskwechsel schneller als über einen Betriebssystemaufruf erfolgt und außerdem gezielt ausgelöst wird.



Dieser Mechanismus ist aufgrund seiner Komplexität sehr anfällig für Programmierfehler und sollte daher nur von wirklich erfahrenen Programmierern verwendet werden.

## 4.8 switch

`switch` erwartet einen `int`-Parameter und verzweigt dann je nach dessen Wert zu der entsprechenden `case`-Anweisung. Sollte keine `case`-Anweisung einen passenden Wert aufweisen, wird die optionale `default`-Anweisung angesprungen. Pro `case`-Anweisung darf nur ein Wert, kein Wertebereich, angegeben werden[3].



ACHTUNG: Die `case`-Anweisungen werden nicht bis zum nächsten `case` ausgeführt, sondern bis zum nächsten `break`!!! Dadurch können mehrere Werte gruppiert werden und den gleichen Effekt bewirken oder sogar einige Werte eine Untermenge bilden.

[3]Der manche Compiler, wie GNU C, bietet diese Möglichkeit jedoch als Erweiterung zum C-Sprachenstandard an

```
switch(getchar())
{
case '0':       i=0;            break;
case '1':       i=1;            break;
case '2':       i=2;            break;
 ...
case 'a':
case 'A':       i=10;           break;
case 'b':
case 'B':       i=11;           break;
 ...
default:        puts("Ungültiges Zeichen !");
}
```

Gerade für `switch`-Verzweigungen sind die Aufzählungstypen 3.3 sehr praktisch. Mithilfe der `switch`-Anweisung kann das Beispiel von Seite 34 folgendermaßen geschrieben werden:

```
void iß(enum Obst welches)
{
        switch(welches)
        {
        case APFEL:     puts("Esse einen Apfel.");
                        break;

        case BIRNE:     puts("Esse eine Birne.");
                        break;

        default:        puts("Esse irgendwas.");
                        break;
}
}
```

Eine wertvolle Hilfe bieten manche Compiler, indem sie bei `switch`-Blöcken, in denen nicht alle Möglichkeiten einer `enum`-Variablen behandelt werden, eine Warnung ausgeben.

## 4.9 Der Konditionaloperator

Der Konditional– (Bedingungs–)operator ist der einzige **ternäre** Operator in C, d.h. er hat drei Operanden. Der erste Operand enthält einen logischen Ausdruck, der entscheidet, ob das Ergebnis des gesamten Ausdrucks der zweite oder dritte Operand sein soll. Der folgende Ausdruck beispielsweise liefert das Minimum der Variablen von `a` und `b`:

```
c = a<b?a:b;
```

Der Konditionaloperator arbeitet völlig parallel zur `if`-Abfrage, nur kann das Ergebnis sofort weiterverwendet werden und die Abfrage an einer beliebigen Stelle innerhalb eines Ausdruckes auftreten. Seine Rangfolge ist eine der niedrigsten, Klammern sind infolgedessen selten nötig.

Das obige Beispiel würde mit `if` in etwa so erscheinen:

```
if (a<b) c = a; else c = b;
```

Allerdings ist die `if`-Version nicht direkt vergleichbar, denn der Ausdruck mit Konditionaloperator kann weiter verwendet werden:

```
d = (tmp=a<b?a:b)<c?tmp:c;
```

und so beispielsweise das Minimum der drei Variablen `a`,`b` und `c` auf einmal berechnen (die `tmp`-Variable ist aus Effizientgründen nötig).

ÜBUNGSAUFGABE: Suche nach dem Minimum/Maximum in einem randomisierten (Funktion `rand()`) *int*-Array.

## 4.10 Übungsaufgaben

### 4.10.1 cgi-Programmierung

In Erweiterung der Aufgabe 2.4.1 soll ein Modul samt Header mit einer Sammlung praktischer Funktionen zur Auswertung von cgi-Parametern geschrieben werden. Dabei soll die Environmentvariable `QUERYSTRING` ausgewertet werden. Mehrere Definitionen sind durch das Zeichen `&` getrennt, Leerzeichen werden als `+` codiert, beliebige Zeichen durch deren hexadezimale Darstellung mit führendem `X`, z.B. `2E`. Folgende Funktionen sind zweckmäßig:

```
/* Wert der Variablen NAME in das char-Array VALUE der Länge n einlesen. */
void    getparam(const char NAME[], char VALUE[], int n);

/* Alle existierenden Variablen der Reihe nach formatiert ausgeben. */
void    listparams();
```

Falls ein WWW-Server zur Verfügung steht, soll das Planetenprogramm 1.9.3 damit WWW-tauglich gemacht werden.

### 4.10.2 Graphik-Programmierung

Da die Verwendung von direkter Graphikausgabe hochgradig von der Programmierumgebung und den zur Verfügung stehenden Libraries abhängt, außerdem zumeist die Verwendung von Zeigern bedingt, die erst später behandelt werden sollen, ist es einfacher, mit Standardbefehlen eine Graphikdatei zu erstellen, die dann mit anderen Programmen betrachtet werden kann.

Als Dateiformat eignet sich dazu das **Targa**-Format ausgezeichnet, da es sehr simpel ist. Es besteht aus einem Header zu 18 Bytes mit folgender Struktur:

| Offset | Bezeichnung | Typ | Wert |
|---:|---|---|---|
| 0 | commentlen | unsigned char | 0 |
| 1 | maptype | unsigned char | 0 für 24-Bit rgb-Format |
| 2 | filetype | unsigned char | 2 für 24-Bit rgb-Format |
| 3 | maporg | unsigned short | 0 |
| 5 | maplen | unsigned short | 0 |
| 7 | mapsize | unsigned char | 0 |
| 9 | xorigin | unsigned short | 0 |
| 11 | yorigin | unsigned short | 0 |
| 13 | MaxX | unsigned short | Pixel in X-Richtung |
| 15 | MaxY | unsigned short | Pixel in Y-Richtung |
| 16 | databits | unsigned char | 24 für 24-Bit rgb-Format |
| 17 | flags | unsigned char | 0 |

Danach folgen die Daten für jedes Pixel, jeweils ein `unsigned char` für den zugehörigen Blau-, Grün-, Rot-Wert. Eine Datei im 24-Bit Format (16 Mio. Farben) mit einer Auflösung von 320x200 ist somit $18 + 3 \cdot 320 \cdot 200 = 192018$ Bytes lang.

Zum Schreiben einer Datei ist ein kleiner Vorgriff auf die Zeigersyntax notwendig. Dieser Vorgriff ist allerdings nicht allzu elementar, sodaß an dieser Stelle einfach ein „Kochrezept" angegeben werden kann. Eine Datei wird mit der Funktion `fopen()` geöffnet, diese Funktion liefert einen Typ `FILE*`, Parameter sind der Dateiname und der Zugriffsmodus. Der Zugriffsmodus ist ein String, in dem das Zeichen „`w`" für Schreibzugriff, „`r`" für Schreibzugriff und unter DOS „`b`" für zeichenweisen („binären") Zugriff sowie „`a`" („ascii") für Zeichenumsetzung `\n` → `\n\r` steht (unter anderen Betriebssystemen als DOS, z.B. Unix, gibt es diese Unterscheidung zwischen Binär und Ascii nicht, dort werden Dateien immer so geschrieben wie sie gelesen werden. Die für DOS notwendige Zusatzangabe „`a`" oder „`b`" wird dann ignoriert.): Die von `fopen()` gelieferte `FILE*`-Variable ist dann der für Funktionen wie `fputc()` (einzelnes Zeichen schreiben), `fputs()` (Zeichenkette schreiben), `fprintf()` (wie `printf()`, schreibt in eine Datei) und `fclose()` (Datei schließen) benötigte Dateiparameter. Ein Anwendungsbeispiel mit typischen Dateifunktionen, das die Zeichen 0-255 in eine Datei schreibt, sieht folgendermaßen aus:

```
#include <stdio.h>

int     main()
{
int     i;
FILE*outfile = fopen("table.txt", "wb");

        if (!outfile)
        {
                perror("Datei \"table.txt\" konnte nicht geöffnet werden");
                return 1;
        }

        for(i=0;i<256;i++)
                fputc(i, outfile);

        fclose(outfile);
        return 0;
}
```

Dabei kann mit „`!outfile`" abgefragt werden, ob die gewünschte Datei geöffnet werden konnte. Falls nicht, gibt die Funktion „`perror()`" den Grund für den Fehler aus, wobei der Fehlermeldung der angegebene Textvorangestellt wird.

Die Targa-Datei kann nun zeichenweise mit der Funktion `fputc()` geschrieben werden. Dabei ist zu beachten, daß bei den Datenelementen vom Typ `unsigned short` zuerst das niederwertige, dann das höherwertige Byte geschrieben werden muß. ERSTE ÜBUNG: Schreibe ein Programm, das eine Targa-Datei mit verschiedenfarbigen waagrechten Linien enthält. Die Helligkeit der Linien soll von links nach rechts ansteigen.

### 4.10.3 Apfelmännchen

Das „Apfelmännchen" ist die berühmteste Anwendung der sog. Fraktale. Ein Fraktal ist ein Gebilde, das beliebig vergrößert werden kann und dennoch immer wieder neue Strukturen hervorbringt, die zwar den großräumigen ähneln, aber nicht ganz identisch sind. Das Apfelmännchen (auch unter dem Namen „Mandelbrotmenge" bekannt) entsteht, indem man die Konvergenz der

Reihe

$$z_{i+1} = z_i^2 + c$$

betrachtet, wobei $z$ und $c$ komplexe Zahlen sind. Man kann zeigen, daß, sobald der Betrag von $z_i$ für ein bestimmes $i$ größer als 2 wird, die Reihe divergiert, d.h. gegen $\infty$ geht. Die Anzahl der Schritte, die für ein bestimmtes $c$ benötigt werden, bis dieses Kriterium erreicht ist, wird in den gängigen Anwendung als **Farbe** desjenigen Punktes gesetzt, mit dem die komplexe Zahl $c$ identifiziert wird.

Konkret wird jede komplexe Zahl $a$ durch zwei Komponenten dargestellt, dem reellen und dem imaginären Anteil:

$$a = a_r + i \cdot a_i$$

wobei $i = \sqrt{-1}$ die imaginäre Einheit ist. $z$ wird anfänglich 0 gesetzt, $c$ sind die Graphikkoordinaten $x$ und $y$, allerdings mit einem Skalierungsfaktor multipliziert und verschoben, sodaß diese Werte zwischen $-2$ und $+2$ liegen (für größere Werte kann man sofort sagen, daß die Reihe divergiert). Wenn die Graphikdatei linear geschrieben wird und eine Variable `linear` bei jeder Pixelausgabe erhöht wird, können die Pixelkoordinaten daraus mit Division und Modulo der Breite `MaxX` berechnet werden:

```
PixelX = linear % MaxX;
PixelY = linear / MaxX;
```

Die Umrechnung in den Bereich $-2, +2$ ist ebenfalls einfach:

```
x = double(PixelX) /MaxX;        /*  0, +1 */
x *= 4;                          /*  0, +4 */
x -= 2;                          /*  -2, +2 */
```

analog in y. Dies definiert die aus den zwei Variablen `x,y` bestehende Variable $c$. Das Quadrat einer komplexen Zahl $z = a + ib$ ist

$$z^2 = (a+ib)^2 = a^2 + 2iab + i^2b^2 = a^2 - b^2 + i2ab$$

d.h. der Realteil ist $a^2 - b^2$, der Imaginärteil $2ab$. Damit braucht nun im Programm nur noch die Rechnung $z = z^2 + c$ für jedes Pixel komponentenweise, d.h. getrennt in Realteil und Imaginärteil, da man in C nicht mit komplexen Zahlen rechnen kann, solange ausgeführt werden, bis der Betrag $|z| \equiv \sqrt{a^2 + b^2}$ den Wert 2 überschreitet.

# 5 Benutzerdefinierte Typen

## 5.1 Datenstrukturen - `struct`

Mit einer `struct` können mehrere Variablen zu einer Gesamtheit zusammengefaßt werden. Eine Strukturdefinition kann verwendet werden, um einen *Typ*, eine oder mehrere *Strukturvariablen* oder *beides gleichzeitig* festzulegen (wie bei `enum`). Die Strukturelemente müssen aus bereits definierten Variablentypen bestehen und werden völlig analog zur Definition von Variablen festgelegt:

```
struct  punkt   { int x,y; };     /* Definition eines Typs struct punkt */
struct  punkt   p,q;              /* Definition zweier Variablen vom   */
                                  /* Typ struct punkt    */
struct  { int   x,y; } p,q; }     /* Definition zweier Variablen, die beide */
                                  /* jeweils aus zwei int's bestehen */
                                  /* (Anonyme Struktur) */

struct  punkt { int x,y; } p,q;   /* Definition zweier Variablen, die beide */
                                  /* jeweils aus zwei int's bestehen  */
                                  /* und gleichzeitig eines Typs struct punkt */
```

Im Unterschied zu `enum`, wo sowohl der Typname als auch die zu erzeugenden Variablen optional sind, muß bei `struct` eines von beiden angegeben werden (andernfalls hätte die Strukturdefinition ja auch keinen Sinn).

Auf die einzelnen Elemente wird über den Punkt „." zugegriffen:

```
struct  punkt   pkt;
        pkt.x = 1;
        pkt.y = 0;
```

Strukturen werden ähnlich wie Arrays initialisiert, nur können hier die Typen unterschiedlich sein (unter Berücksichtigung etwaiger Typenkonversion):

```
struct  info
{
        double  d;
        char    s[10];
        int     i;
};

struct  info info_var = { 3.67, "text", 97 };
```

Die Initialisierung muß exakt in der Reihenfolge der Strukturelemente erfolgen, eine explizite Angabe des zu initialisierenden Elementes, ist in C **nicht** möglich[4].

Die Größe einer Struktur (`sizeof struct ...` ) ist nicht notwendigerweise die Summe der Größen der Elemente dieser Struktur. Je nach Compiler und -einstellungen können die einzelnen Datenelemente auf ganzzahlige Vielfache von 2 „aligned" (ausgerichtet) werden, da dies für manche Prozessoren Geschwindigkeitsvorteile bringt oder sogar notwendig ist.

Eine Funktion kann eine Struktur als Funktionswert liefern

[4] GNU C/C++ bietet hierfür eine Erweiterung des Sprachenstandards an

```
struct punkt Kreis_koordinate(int r, float phi)
```

Strukturen können auch als Funktionsparameter eingesetzt werden; dies ist allerdings recht ineffizient, da die gesamte Struktur dazu kopiert werden muß. Manche Compiler geben daher beim Aufruf der Funktion eine Warnung aus. In den seltensten Fällen ist eine lokale Kopie wirklich notwendig, meistens genügt ein Zeiger oder in C++ eine Referenz auf eine außerhalb der aktuellen Funktion existierende Strukturvariable — dies wird daher erst im Abschnitt 6.6 (Zeiger) behandelt.

Da die Definition der Strukturelemente völlig analog zur Variablendefinition erfolgt, können auch verschachtelte Strukturen in einem generiert werden; dabei sind anonyme Strukturen recht sinnvoll.

Definition einer Struktur, die drei Punkte enthält, welche jeweils aus zwei *int*'s bestehen:

```
struct Trigon
{
        struct
        {
                int x,y;
        }
        P0,P1,P2;
};

struct Trigon T = { {1,0}, {1,1}, {0,1} };
```

Hierbei kann gleichzeitig ein globaler Strukturtyp (aus der inneren verschachtelten Strukturdefinition) erzeugt werden, es ist jedoch in C nicht zulässig, **nur** einen Strukturtyp zu erzeugen.

Natürlich läßt sich auch dieses Beispiel noch kompaktifizieren:

```
struct
{
        struct
        {
                int x,y;
        }
        P0,P1,P2;

}
        T = { {1,0}, {1,1}, {0,1} };
```

Bei dieser Unmenge an Möglichkeiten ist verständlicherweise auch hier wieder eine angemessene Selbstbeherrschung angebracht.

In einer Struktur können auch enum-Typen definiert werden:

```
struct  Obst
{
        enum { APFEL, BIRNE, ... } welches;
        double  Preis;
};

struct  Obst    A = { APFEL, 3.4},
                B = { BIRNE, 5.6};
```

Eine derartige enum-Definition ist dann mit einer globalen Definition gleichwertig und bewirkt einen Fehler, wenn eine *enum*-Konstante gleichen Namens global definiert wurde oder noch wird (im Gegensatz zur lokalen Definition eines enum-Typs).

Auch eine Strukturdefinition kann lokal sein. Dabei kann es jedoch geschehen, daß man sich selbst den Ast absägt:

```
struct x
{
        int     i,j,k;
};

struct x f()
{
        struct x
        {
                double  e,f,g;
        };

struct  x X;
        return X;
}
```

Dieses Beispiel kann nicht funktionieren, da die lokale Variable X zwar vom Typ einer Struktur des Namens x ist, diese Struktur unterscheidet sich aber deutlich von der gleichnamigen Struktur x, die als Funktionswert geliefert werden soll. Dasselbe kann auch mit enum-Typen geschehen, nur merkt man es dort nicht ...

Eine Struktur kann auch **deklariert** werden:

```
struct punkt;
```

Wozu das nötig ist und wann man dies braucht, wird im Abschnitt 6.6 erklärt.

## 5.2 Bitfelder

Strukturelemente müssen nicht unbedingt vollständige Variablen sein, auch einzelne Bits können ein Element bilden:

```
struct  Color_attribute
{
        unsigned char fg_color:3, intensity:1, bg_color:3, blinking:1;
};
```

Obige Struktur bildet genau die Bitstruktur nach, mit der die Farbe der Textzeichen am Bildschirm unter MSDOS codiert wird. Die Strukturelemente „`fg_color`“ und „`bg_color`“ bestehen dabei jeweils aus 3 Bits und können damit einen Wertebereich von 0 bis 7 annehmen. „`intensity`“ und „`blinking`“ sind einzelne Bits und stellen damit „echte“ Boolean-Variablen dar. Bei der Zuweisung eines höheren Wertes werden überzählige Bits verworfen. Die gesamte Struktur hat `3+1+3+1 = 8` Bits (`sizeof(struct Color_attribute)` $\equiv$ `1`).

Übungsaufgabe: Als Erweiterung von Beispiel 4.10.2 soll dort die Behandlung des Headers von Targa-Dateien mittels `struct` ausgeführt werden. Es soll eine Funktion erstellt werden, die aus den Parametern Länge und Breite eine Struktur für eine Targa-Datei erstellt, eine andere Funktion soll diese Struktur byteweise ausgeben.

## 5.3 `union`'s

Eine `union` wird genau wie eine Struktur definiert und auch gleich behandelt. Der Unterschied besteht darin, daß die Elemente einer `union` (auf Deutsch etwas holprig „Variante“ genannt) nicht wie bei einer Struktur **nebeneinander**, sondern **übereinander** liegen, also im gleichen Speicherbereich.
Damit ist es möglich, ein und denselben Speicherbereich beispielsweise als `int`, `double` oder `char[]` anzusprechen:

```
union   Float_und_Long
{
        float   f;
        long    l;
        char    c[sizeof(float)];
};
```

Die Größe einer `union` entspricht der maximalen Größe eines Elementes. Bei jeder Veränderung eines Elementes werden auch alle anderen geändert.

Vom internen Speicheraufbau abgesehen, verhält sich eine `union` in allen Belangen genau gleich wie eine `struct`.

Übungsbeispiel: Ausgeben der Bitstruktur eines Fließkommawertes.

## 5.4 `typedef`

Bei allen selbstdefinierten Typen (`enum`, `struct`, `union`) muß in C bei der Angabe des Typs zur Variablendefinition oder Typenkonversion immer explizit der Vorsatz `enum`, `struct` oder `union` mitangegeben werden. Da dies recht lästig ist, kann mittels `typedef` ein neuer Typname definiert werden, der dann wirklich mit den Standardtypen gleichberechtigt ist. Die Definition erfolgt parallel zur Definition einer Variablen, diese Definition wird jedoch mit `typedef` eingeleitet. Anstelle einer Variablen wird dann ein Typ definiert:

```
typedef struct  punkt   Punkt;  /*  Definition des neuen Typs Punkt */
Punkt   P = {8,9};
```

Manchmal werden auch Standardtypen aus Bequemlichkeit als eigener Typ definiert:

```
typedef unsigned short u_short;

typedef unsigned char byte;
```

Das Problem dabei ist, daß bei unsachgemäßer Anwendung zwar viele schöne neue Typnamen entstehen, bei der Definition von Variablen wird aber dann verschleiert, von welchem Typ sie eigentlich sind. Das Ergebnis ist ein scheinbar schönes Programm, in dem sich niemand mehr auskennt.

Recht häufig wird gleichzeitig mit der Definition einer Struktur auch ein Strukturtyp erstellt, ohne daß ein Name für die Struktur selbst vergeben wird:

```
typedef struct
{
        int x,y;
}
        Punkt;          /*  Definition des Typs Punkt */
```

Achtung! Trotz des Namens „`typedef`“ wird damit nicht wirklich ein neuer Typ definiert, sondern nur ein Alias-Name festgelegt. Das heißt, im folgenden Fall:

```
typedef int     type_a;
typedef int     type_b;

type_a  a;
type_b  b;
```

sind die Variablen `a` und `b` jeweils vom Typ `int` und können ohne Warnung einander zugewiesen werden.

In C++ ist der Vorsatz `enum`, `struct` oder `union` bei selbstdefinierten Typen nicht mehr nötig und kann weggelassen werden. Die ursprüngliche Existenzberechtigung für `typedef` ist daher in C++ fortgefallen und nur in komplexen Situationen hat es einen Sinn, `typedef` zu verwenden.

## 5.5 Übungsbeispiel

Das Beispiel 4.10.2 soll soweit ausgebaut werden, daß in einem Modul alle notwendigen Funktionen zur Behandlung von Targa-Dateien untergebracht werden. Das Header-Modul soll in etwa der folgenden Struktur entsprechen:

```
typedef struct
{
        ...
}
```

```
        Targa;

typedef struct
{
        unsigned char r, g, b;
}
        rgb;

Targa   Header(int MaxX, int MaxY);     /*  Header erzeugen */
void    write_Header(Targa Hdr, FILE*outfile);  /*  Dateiheader schreiben */
void    write_rgb(rgb color, FILE*outfile);     /*  rgb-Pixel schreiben */
```

### 5.5.1 Bresenhams Kreisalgorithmus

Um einen Kreis zu zeichnen, könnte man einfach mit Sinus und Cosinus für alle Winkel zwischen 0 und $2\pi$ die Koordinaten ausrechnen und einen Punkt setzen. Schneller sind jedoch optimierte Algorithmen, die ausschließlich mit `int`'s rechnen, wie die folgende Implementation des Kreisalgorithmus von Bresenham:

```
rgb C[320][200];

void    Circle(int xc, int yc, int r, rgb I)
{
int     x=0,y=r,d=2*(1-r);
        while(y>x)
        {
                C[xc+y][yc+x] = C[xc+x][yc+y] =
                C[xc-x][yc+y] = C[xc-y][yc+x] =
                C[xc-y][yc-x] = C[xc-x][yc-y] =
                C[xc+x][yc-y] = C[xc+y][yc-x] = I;

                if (d+y>0)
                {
                        y--;
                        d+=-2*y+1;
                }
                if (x>d)
                {
                        x++;
                        d+=2*x+1;
                }
        }
}
```

Da mehrere Punkte gleichzeitig angesprochen werden müssen, ist der Zwischenspeicher `C[][]` notwendig. Nach dem Aufruf aller im Hauptprogramm verwendeten `Circle()`-Funktionen muß der gesamte Zwischenspeicher in die Graphikdatei geschrieben werden. Aufgabe: Schreibe ein Programm, daß mithilfe des Zufallsgenerators verschiedene Kreise in eine Targa-Datei zeichnet. Modifiziere dann den Kreisalgorithmus dahingehend, daß sich die Farbe innerhalb eines einzigen Kreises ohne scharfen Farbsprünge ändert.

# 6 Einfache Zeiger und dynamischer Speicher

## 6.1 Das Konzept des Zeigers

Jede Variable steht an einer bestimmten Stelle im Speicher. Diese Speicherstelle wird von dem Prozessor über einen Zahlenwert (der **Adresse** der Variablen) angesprochen. Die Adresse einer Variablen erhält man mit dem **unär** gebrauchten *Operator* `&`, so ist z.B. `&i` ist die Adresse (oder: *Referenz*) der Variablen `i`. Um die Adresse einer Variablen auszugeben, ist bei der Funktion `printf()` das Kürzel `%p` vorgesehen:

```
int     i;
        printf("Die Adresse der Variablen i ist %p !  \n",&i);
```

Der Adreßoperator `&` ist nur erlaubt für

- Funktionen
- LValues, die nicht `register`-Variablen oder Bitfelder sind

Es ist nicht möglich (weil nicht sinnvoll), die Adresse einer `enum`-Konstanten oder einer Zahl zu bilden (etwa: `&1`) oder die Adresse einer Adresse (`&&i`).

Ein Zeiger (engl. *Pointer*) ist eine Variable, die auf eine andere Variable oder Funktion zeigt und daher die Adresse dieser anderen beinhalten muß.

`int`-Variablen werden bekanntermaßen wie folgt definiert:

```
int i, j, k;
```

Die Definition von Zeigervariablen erfolgt vollkommen analog:

```
int *p, *q;
```

Damit wird eine Variable vom Typ `int` „mit dem Namen `*p`“ definiert. Diese Variable kann im Prinzip so wie jede andere auch behandelt werden:

```
        *p = 42;
        i = 18 + *p - j;
        j /= *p;
```

Im Gegensatz zu einer gewöhnlichen Variablen handelt es sich bei `*p` jedoch um keine echte, sondern um eine *virtuelle* Variable, d.h. `*p` existiert nicht als Variable für sich, sondern ist dadurch an die *Zeigervariable* `p` gekoppelt, daß `p` die Adresse von `*p` enthält:

$$\texttt{*p} \Longleftrightarrow \texttt{p}$$

Inhalt des Zeigers     Adreßwert

Die Variable `p` selbst enthält die Adresse, die angibt, wo sich die assoziierte `int`-Variable `*p` befindet. Der hier **unär** auf die Zeigervariable `p` angewandte *Operator* `*` liefert den **Inhalt** der Speicherstelle, auf die `p` verweist. Der **unäre** Operator `*` heißt Dereferenzierungsoperator, mit seiner Hilfe kann ein Zeiger ausgewertet werden.

Um nicht irgendeine völlig beliebige Speicherstelle zu verändern, ist es notwendig zu wissen, wo eine bestimmte Variable vom Compiler im Speicher abgelegt wurde. Diese Information liefert der schon erwähnte Adreßoperator `&`. Nach der Zuweisung `p=&i;` kann `*p` als gültige Variable verwendet werden. Dabei gilt: `*p`≡`i`, jede Veränderung der „virtuellen“ Variablen `*p` bewirkt eine Veränderung der „realen“ Variablen `i`.

Zeigervariablen, die auf einen `int` zeigen, sind vom Typ `int*`. Mit den Variablen

```
int i;
int *p;
```

gilt:

| Ausdruck | Typ | LValue | Bezeichnung |
|---|---|---|---|
| `i` | `int` | Ja | Eine `int`-Variable |
| `&i` | `int*` | Nein | Adresse der `int`-Variablen i |
| `p` | `int*` | Ja | Zeiger auf eine Variable vom Typ `int` |
| `*p` | `int` | Ja | Derefenzierter Zeiger auf einen `int` |
| `*i` | `---` | | Nicht möglich |
| `&p` | `int**` | Nein | Adresse eines Zeigers auf einen `int` |

Die unären Operatoren `&` (Referenz) und `*` (Dereferenz) heben sich gegenseitig auf. `*&i` ist identisch mit `i`, `&*p` identisch mit `p`.

ACHTUNG! Auch ohne die Zuweisung `p=&i;` kann `*p` ohne Einschränkung verwendet werden, wodurch aber irgendwelche undefinierten Speicherbereiche verändert werden können - ein Programmabsturz ist dann eine relativ harmlose[5] Konsequenz daraus.

Bei der Definition von Zeigern können Zeigervariablen mit gewöhnlichen Variablen beliebig gemischt werden:

```
int a, *p, b, *q;
```

Dabei ist aber zu beachten, daß „`*`“ einen quasi-Teil des Variablennamens darstellt:

```
int* p,a;
```

... definiert einen `int a` und **nicht** einen `int * a`! Eine derartige Mischung spricht für schlechten Programmierstil und ist ein erster Schritt zu unverständlichen Programmen. Manche Programmierer ziehen es daher vor, einen eigenen Zeigertyp zu definieren:

```
typedef pointer_to_int        *p;
```

Dies vermeidet Fehler wie sie bei gemischten Definitionen vorkommen können:

```
pointer_to_int        p, a;
```

Sowohl `p` als auch `a` sind vom Typ `int*`. Nachteil der Verwendung von Typdefinitionen ist jedoch, daß für jeden vorkommenden Datentyp auch die Definition eines Zeigertyps notwendig ist und diese Definitionen in allen verwendeten Programmodulen identisch sein müssen. Zudem geht die Assoziation „`*`“⇔*Zeiger* verloren.

## 6.2 Zeiger als Funktionsparameter

Eine Funktion kann Zeiger als Funktionsparameter besitzen:

```
void f(int *ptr, struct value *data);
```

Diese Verwendung als Funktionsparameter ist eine der häufigsten Anwendungen von Zeigern, denn damit wird der „call by reference“ in C realisiert. Über den als Parameter übergebenen Zeiger kann die Funktion auf eine außerhalb gelegene Variable zugreifen. Wird die Variable selbst übergeben („call by value“), kann die Funktion zwar den Parameter selbst verändern, nicht aber die eigentliche

[5]Ein Programmabsturz fällt wenigstens sofort auf, im schlimmsten Fall jedoch können betriebssysteminterne Daten zur Festplatten- oder Diskettenverwaltung zerstört werden und somit irgendwann später, nach Beendigung des eigentlichen fehlerverursachenden Programmes, zu Datenverlust führen. Derartige Fehler sind praktisch unauffindbar.

Variable außerhalb der Funktion. Als Parameter müssen dieser Funktion dann die Adressen von `int`-Variablen übergeben werden.

Ein klassische Anwendung des „call by reference" in C ist die zur Ausgabefunktion `printf()` parallele Eingabefunktion `scanf()` (scan formatted). Ihr erster Parameter ist ein zu `printf()` ähnlicher Formatstring, danach folgen die Adressen der einzulesenden Variablen:

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

Anhand des Rückgabewertes von `scanf()` kann festgestellt werden, **wieviele** Variablen korrekt eingelesen wurden. Nicht eingelesene Variablen behalten ihren Wert von vorher.

WARNUNG! Die Funktion `scanf()` arbeitet nicht unbedingt so, wie man es erwarten könnte! Im folgenden Beispiel wird nicht etwa von `scanf()` ein Text ausgegeben, der zur Eingabe auffordern würde, sondern es **muß** der Text „`int:`" eingegeben werden, damit der folgende `int` korrekt eingelesen wird:

```
int     i;
        scanf("int:%d",&i);
        printf("int:%d",i);
```

WICHTIGE WARNUNG! Ein schwerer Fehler, der auch erfahrenen Programmieren passiert, ist es, nicht die Adresse einer Variablen, sondern die Variable selbst als Parameter von `scanf()` anzugeben! In diesem Fall wird der Wert der im allgemeinen noch nicht initialisierten Variablen von der Funktion `scanf()` als Adreßwert interpretiert und irgendeine Speicherstelle unkontrolliert überschrieben!

Die Parameter werden, wie bei `printf()`, nicht auf Richtigkeit überprüft. Während ein falscher Parameter bei `printf()` maximal eine falsche Bildschirmausgabe zur Folge hat, können sich falsche Parameter bei `scanf()` äußerst fatal auswirken!!

## 6.3 Gültigkeit von Zeigern

Ein dereferenzierter Zeiger ist ein LValue , ihm kann etwas zugewiesen werden:

```
double  d;
double  *d_ptr;

        d_ptr = &d;      /*  d_ptr zeigt auf d */
        *d_ptr = 245;    /*  d = 245; */
        *d_ptr -= 245;   /*  d = 0; */
```

Dies kann bei Funktionen auf den ersten Blick etwas kryptisch erscheinen:

```
double  val;
double  *f(void)
{
        return &val;
}

int     main()
{
```

```
        *f() = 5.83;      /* Nicht der Funktion f() */
                          /* wird hier etwas zugewiesen, */
                          /* sondern der globalen Variablen val ! */
        return 0;
}
```

Wichtig! Ein Zeiger ist nur solange gültig, als auch sein Inhalt gültig ist! Ein Zeiger, der auf eine lokale Variable zeigt, ist nur innerhalb des Gültigkeitskeitsbereiches dieser Variable verwendbar, auch dann, wenn die Zeigervariable selbst einen größeren Gültigkeitsbereich umfaßt. Dieser Fall tritt ebenso bei Funktionen in Erscheinung, die einen Zeiger als Funktionswert liefern:

```
int     *f(int val)
{
int     i = val;
        return  &i;       /* Hier ist der Programmfehler ! */
}

int     main()
{
int     *p;

        p = f();
        *p = 5;           /* Hier wird eine undefinierte Speicherstelle */
                          /* überschrieben ! */
        return 0;
}
```

Jeder Zugriff auf `*f()` führt zu einem sicheren Programmfehler, denn der zurückgelieferte Zeiger hat außerhalb der Funktion `f()` selbst keinen gültigen Wert mehr. Da es sich dabei um völlig legalen C-Code handelt, kann man auch keine Hilfestellung von seiten des Compilers erwarten.

## 6.4 Typkonversionen von Zeigern

Da alle Zeiger unabhängig vom Typ, auf den sie zeigen, identisch aufgebaut sind, ist die Zuweisung einer Zeigervariablen des einen Typs an eine Zeigervariable anderen Typs zwar möglich, erzeugt aber eine Compilerwarnung. Sie kann jedoch durch explizite Typenkonversion unterdrückt werden.

```
short   i;
char    *char_ptr;
char    low_byte;

        i = 0x1234;
        char_ptr = (char*)&i;           /* char_ptr zeigt auf i */
        low_byte = *char_ptr;

        printf("int: %X, low Byte: %X\n",i,(int)low_byte);
        *char_ptr = 0x12;
        printf("int: %X\n",i);
```

Übungsfrage: Warum ist im obigen Beispiel die Typenkonversion `(int)low_byte` nötig?

Die Verantwortung für die Typenkonversion eines Zeigers liegt vollständig in den Händen des Programmierers:

```
double  d;
double  *d_ptr;
int     *i_ptr;

        d = 1E-5;               /*  ... irgendeine Zahl */
        d_ptr = &d;             /*  d_ptr zeigt auf d */
        i_ptr = (int*)d_ptr;    /*  i_ptr zeigt auf d */
        *i_ptr = 25000;         /*  d = ???  */
```

Die Variable `d` erhält in diesem Beispiel einen undefinierten Wert.

ANMERKUNG: Es ist wohl jedem klar, daß sich auch hier alle Operationen bis zur Unkenntlichkeit kompaktifizieren ließen ...

Eine Besonderheit stellt der `void*`-Zeiger dar: Ein void-Zeiger zeigt auf „nichts“ (keinen Typ, nur Speicherstelle), d.h. er kann nicht dereferenziert (ausgewertet) werden. Jeder beliebige Zeiger kann in einen `void*`-Zeiger konvertiert werden, aber nicht umgekehrt, da durch die Konversion die Typinformation verloren geht. Ein `void*`-Zeiger ist also ein nicht typisierter Zeiger, die Zuweisung an einen typisierten Zeiger oder die Auswertung setzt explizite Typenkonversion voraus:

```
void*   v_ptr;
double  d,f;

        d = 1;
        v_ptr = &d;             /*  v_ptr —> d */
        f = *(double*)v_ptr;    /*  einfacher: f = d */
```

Für die Zuweisung eines expliziten Adreßwertes an einen Zeiger gilt dasselbe wie für einen `void*`-Zeiger. Die einzige Ausnahme spielt der Zahlenwert Null (`0`). Jedem Zeiger kann die Zahl 0 zugewiesen werden, um anzudeuten, daß der Zeiger keine gültige Adresse enthält, denn kein Objekt, von welchem Typ auch immer, wird an der Speicheradresse 0 angelegt. Um anzudeuten, daß die Zuweisung einer Zahl an einen Zeiger problematisch ist, wird üblicherweise statt der Zahl 0 selbst die Konstante `NULL` verwendet, die beispielsweise in `stdio.h` definiert wird. (ANMERKUNG: Obwohl zur Definition dieser Konstanten `enum` verwendet werden könnte, hat man dort eine andere Methode gewählt, die aufgrund ihrer Problematik erst später in 7 behandelt wird.)

Ein Zugriff auf den Inhalt eines Null-Zeigers bedeutet daher einen schweren Programmfehler, der von der Programmierumgebung gemeldet werden kann.

## 6.5 Zeigerarithmetik

Mit Zeigern können einfache arithmetische Operationen unternommen werden. Sinnvoll ist dies, wenn mehrere Datenelemente gleichen Typs im Speicher hintereinander stehen, also ein Array bilden. Die Adresse des ersten Elementes in einem Array läßt sich einfach ermitteln:

```
double  A[100];
double  *ptr;

                ptr = &A[0];
```

Der Zeiger auf das nächste Element kann einfach mit `ptr+1` ausgerechnet werden, er zeigt dann auf `A[1]`. Zeiger werden also nicht byteweise erhöht, sondern jeweils um die Größe desjenigen Objektes, auf das er zeigt. Da `void`-Zeiger keinem Typ zugeordnet sind, ist mit ihnen keine Zeigerarithmetik möglich. Die Addition zweier Zeiger ist nicht sinnvoll und daher ebenfalls nicht erlaubt. Für einen beliebigen `int`-Wert `i` und einen beliebigen Zeiger `ptr` gilt allgemein:

$$\boxed{\texttt{ptr + i} \equiv \texttt{(char*)ptr + i*sizeof(*ptr)}}$$

`(char*)ptr` ist ein byteweiser Zeiger, mit dem jede Speicherstelle einzeln adressiert werden kann; jedes Arrayelement ist um `sizeof(*ptr)` Bytes vom jeweils vorigen Element entfernt.

Jedes Element eines Arrays kann infolgedessen sehr einfach über einen Zeiger angesprochen werden:

$$\texttt{*(ptr+i)} \equiv \texttt{A[i];}$$

Aufgrund dieser Identität ist der Zugriff auf ein Element eines Arrays in C über den Zeigerzugriff **definiert**:

$$\boxed{\texttt{A[i] := *(A+i);}}$$

D.h. das Array entspricht einem Zeiger auf das erste Element `*A` liefert das erste Element `A[0]` und damit `&A[0]` ≡ `&*A` ≡ `A`.

Die *Adresse* eines Arrays zeigt ebenfalls auf das Array, daher gilt:

$$\texttt{A} \equiv \texttt{\&A}$$

Dies kann leicht zu Verwirrungen führen. Zwischen `A` und `&A` besteht numerisch keinerlei Unterschied, allerdings ist `A` vom Typ „*Pointer to* `double`", ein Zeiger auf **ein** Element, und `&A` vom Typ „*Pointer to Array of* `double`", ein Zeiger auf ein **Array**.

Jedes Array kann als Zeiger verwendet werden und jeder Zeiger als Array. Einem echten Zeiger kann jedoch ein Wert zugewiesen werden, ein Array ist hingegen ein konstanter Zeiger und damit kein LValue.

Wird in einer Funktion ein Array als Parameter benötigt, beispielsweise bei der Libraryfunktion `strlen()`, die die momentane Länge eines Strings (Array of char) liefert, dann ist ein Array mit einem Zeiger **vollkommen** identisch, die Funktionsdeklaration

```
int        strlen(char s[]);
```

unterscheidet sich in nichts von der Deklaration

```
int        strlen(char*s);
```

Welche von beiden Möglichkeiten gewählt wird, ist rein willkürlich.

Bei arithmetischen Operationen ist die Rangfolge von Bedeutung, denn auch die Derefenzierung stellt einen **Operator** mit einem festgelegten Rang dar (unärer Operator `*`). So bewirkt `*p+=10` die Addition des **Inhalts** `*p` des Zeigers `p` mit 10 und nicht die Erhöhung des Zeigers `p` um 10 Elemente, also `(*p)+=10`, **nicht** `*(p+=10)`. Im Gegensatz dazu bedeutet `*p++` nicht `(*p)++` sondern `*(p++)`. ÜBUNGSFRAGE: Warum ?

Die Standardlibraryfunktion zur Berechnung der Länge eines Strings kann damit folgendermaßen formuliert werden:

```
int     strlen(char sptr[])
{
int     len = 0;
        while(*sptr++)
                len++;

        return len;
}
```

Die Addition einer ganzen Zahl `i` zu einem Zeiger `p` liefert einen anderen Zeiger `q`. Mit den Variablendefinitionen

```
int     i;
```

```
int     *p, *q;
```

läßt sich dies so ausdrücken:

$$q + i = p$$

Daraus folgt, daß auch die Subtraktion zweier Zeiger möglich sein muß:

$$i = p - q$$

Sie ist nur definiert, wenn beide Zeiger sich auf das gleiche Array beziehen und liefert dann die Anzahl der Elemente zwischen diesen beiden Zeigern. Die Addition zweier Zeiger ist nicht möglich.

Die Funktion `strlen` kann damit auch so verwirklicht werden:

```
int     strlen(char*sptr)
{
char    *start = sptr;

        while(*sptr++)
                ;

        return sptr-start-1;
}
```

Diese Version ist etwas effizienter, da in der zeitkritischen `while`-Schleife ein eigentlich unnötiger Zähler wegfällt.

Bei lokalen Strings ist es wesentlich, ob die Syntax `char s[]="Text";` oder `char*s="Text";` verwendet wird. Im ersten Fall wird nämlich am Stack Platz in der Größe des gesamten Initialisierungsstrings reserviert und dieser dann bei jedem Funktionsaufruf kopiert. Im zweiten Fall wird nur ein Zeiger auf die Adresse eines zu Programmstart initialisierten Strings im statischen Datensegment gesetzt.

Andere arithmetische Operationen als `+` und `-` sind in Zeigerausdrücken nicht möglich. Allerdings ermöglicht die Subtraktion von Zeigern auch der Vergleich von Zeigern. Ist ein Zeiger `p<q`, dann bedeutet dies, daß das Element `*p` im gleichen Array vor dem Element `q` steht (wenn sich nicht beide Zeiger auf das gleiche Array beziehen, ist der Zeigervergleich sinnlos). Identität zweier Zeiger bedeutet, daß sich beide Zeiger auf die gleiche Variable beziehen, und damit auch deren Inhalte identisch sein müssen:

$$p == q \qquad \Longrightarrow \qquad *p == *q$$

ÜBUNGSAUFGABE: Klassisches `strcpy()`. Diese Funktion kopiert einen String in einen anderen und liefert die Adresse des Zielstrings. Dazu ein Testprogramm, mit dem auch festgestellt werden kann, ob das 0-Byte richtig kopiert wurde:

```
char    A[]="0123456789",
        B[]="ABCDEFGHIJKLMNOPQ";

int     main()
{
        puts(strcpy(B,A));
        return 0;
}
```

## 6.6 Zeiger auf Strukturen

Zeiger auf Strukturen werden vor allem dann sehr häufig verwendet, wenn Datenstrukturen als Funktionsparameter verwendet werden sollen. Beim Zugriff auf ein Element ist zu beachten, daß die Elementauswahl über einen Operator höchster Priorität erfolgt: `*p.x` bedeutet daher „Auswertung des Zeigers `x` in der Datenstruktur `p`" und nicht „Element `x` in der Datenstruktur `*p`". Da beides etwas völlig anderes bedeutet, kann der Compiler dies leicht unterscheiden und entsprechende Fehler melden. Um über einen Strukturzeiger auf ein Element zuzugreifen muß `(*p).x` geschrieben werden. Als praktische Abkürzung dafür gibt es in C den Zeigeroperator „`->`".

```
struct_ptr->element := (*struct_ptr).element
```

Wenn eine Struktur einen Zeiger auf eine andere Struktur enthält, ist die Verwendung des Zeigeroperators übersichtlicher:

```
struct  punkt   {       int             x,y;            };
struct  Trigon  {       struct  punkt   *p1,*p2,*p3;    };

int     f(struct Trigon*T)
{
        return T->p1->x;
}
```

Vergleiche: (nur die nötigsten Klammern sind gesetzt)

```
         ...
        return (*(*T).p1).x;
```

Manchmal kommt es vor, daß Strukturen gegenseitig aufeinander verweisen. Das folgende Beispiel beschreibt ein Wörterbuch, das eine bestimmte, dynamisch veränderliche Anzahl von Einträgen enthält, wobei jeder Eintrag zwei Versionen eines Wortes beschreibt. Jeder Eintrag enthält wiederum einen Verweis auf das Wörterbuch, sodaß mehrere Wörterbücher parallel nebeneinander existieren können und jeder Eintrag getrennt behandelt werden kann.

```
struct  entry
{
        char *from, *to;
        struct Lexikon *language;
};

struct  Lexikon
{
        int     entries;
        struct entry *translator_table;
};
```

Das Problem hierbei ist, daß zum Zeitpunkt der Definition von `struct entry` die Struktur `Lexikon` noch nicht definiert ist. Deren Definition kann aber nicht vorgezogen werden, weil dann das gleiche Problem umgekehrt auftreten würde.

Da Zeiger unabhängig von dem, auf das sie zeigen, immer die gleiche Größe haben, kann auch ein Zeiger auf eine Struktur definiert werden, die noch nicht definiert wurde. Als Typ wird dann `void*` angenommen. Wird dann später die Struktur doch noch definiert, erkennt der Compiler diese nicht als identisch und meldet einen Fehler[6].

Um dieses Dilemma zu lösen, kann eine Struktur **deklariert** werden. Wird folgende Deklaration im obigen Beispiel vor der Definition von `struct entry` eingefügt, gibt es keine weiteren Probleme:

```
struct Lexikon;
```

Da jede Definition auch eine Deklaration ist, kann eine Struktur einen Zeiger auf sich selbst enthalten:

```
struct  listelement
{
        struct  listelement *next;
         ...
};
```

## 6.7 `const`-Spezifikation und Zeiger

Um Variablen vor unbeabsichtigten Veränderungen zu schützen, können sie als `const` spezifiziert werden. `const` wird als Vorsatz vor dem Variablennamen oder Variablentyp verwendet. Eine `const`-Variable kann nur initialisiert werden:

```
const int i = 56;
```

Jede versuchte Änderung (`i++` etc.) bewirkt einen Fehler beim Compilieren des Programmes.

Eine konstante Variable kann in C über eine Pointerkonversion modifiziert werden. Da eine derartig heimtückische Programmierweise weit über das hinausreicht, was sich mit einem Schweinchen-Symbol ausdrücken ließe, steht an dieser Stelle auch keines, um nicht zusätzlich Aufmerksamkeit auf diese Möglichkeit zu lenken.

Eine sehr hilfreiche Anwendung sind die Zeiger auf konstante Objekte:

```
const int *i;
```

definiert einen Zeiger, dem die Adresse einer Variablen zugewiesen werden kann, aber dessen Inhalt nicht verändert werden kann. Die Variable, deren Adresse `i` enthält, kann hingegen sehr wohl modifiziert werden; dieses Konstrukt gewährleistet nur, daß diese Variable *nicht über den Zeiger* beschrieben wird. `*i` ist somit eine Variable, die nur ausgelesen werden kann. Als Funktionsparameter angewandt kann sichergestellt werden, daß ein Parameter, der mittels Zeiger übergeben wird (call by reference) , von dieser Funktion nicht verändert wird:

```
void f(const struct Trigon*T);
```

Ein Zeiger auf ein nicht konstantes Objekt kann implizit in einen Zeiger auf ein konstantes Objekt konvertiert werden, umgekehrt ist dies nicht möglich.

Soll nicht der Inhalt eines Zeigers, sondern die Zeigervariable selbst konstant sein, muß diese als

```
int * const iptr;
```

definiert werden. Damit sind folgende Kombinationen möglich:

[6]Borland C hat sich dabei über den ANSI-C Standard hinweggesetzt und erlaubt auch „nachträgliche“ Definitionen

| Variablendefinition | `iptr = 0` | `*iptr = 0` |
|---:|---:|---:|
| `int * iptr;` | erlaubt | erlaubt |
| `const int * iptr;` | erlaubt | nicht erlaubt |
| `int const * iptr;` | erlaubt | nicht erlaubt |
| `int * const iptr;` | nicht erlaubt | erlaubt |
| `const int * const iptr;` | nicht erlaubt | nicht erlaubt |
| `int const * const iptr;` | nicht erlaubt | nicht erlaubt |

Die Verwendung des `const`-Schlüsselwortes kann ausgesprochen mühsam sein, da, einmal angefangen, das System konsequent bis zum Ende durchgezogen werden muß. Gerade dies ist aber eine unbeschreiblich wertvolle Hilfe, oftmals wird erst hierdurch klar, welche Funktion welche Seiteneffekte bewirkt. Aus diesem Grunde ist es außerordentlich ratsam, `const` solange und so ausführlich zu verwenden, bis irgendeine Gegenindikation auftritt. Wirklich schwere systematisch Programmfehler können oft mithilfe der `const`-Spezifikation von vorneherein verhindert werden.

Ursprünglich bezeichnete ein LValue alles, dem etwas zugewiesen kannt. Mit der Einführung von `const` muß dieser Begriff allerdings etwas korrigiert werden, da nun auch Ausdrücke, die an und für sich beschreibbar wären, nicht zuweisbar sein können. Man unterscheidet daher **modifizierbare LValues** von **nicht modifizierbaren LValues**.

## 6.8 Dynamischer Speicher (Übungsaufgabe)

Zur Verwaltung des dynamischen Speichers dienen die Libraryfunktionen

| | |
|---|---|
| `malloc()` | Speicher holen |
| `free()` | Speicher freigeben |
| `realloc()` | Speichergröße nachträglich ändern |

Um das Verständnis für diese Funktionen zu erhöhen und Erfahrung im Umgang mit Zeigern, Arrays sowie Listen zu sammeln, dient die folgende ÜBUNGSAUFGABE: Schreibe eine Funktion `char*getmem(int size);`, die aus einem statischen Array `char mem[20000];` einzelne Speicherblöcke liefert.

Eine Möglichkeit diese Aufgabe zu lösen, besteht darin, für jeden Speicherblock eine Struktur zu erstellen, die alle nötigen Informationen über diesen Block enthält. Jede dieser Datenstrukturen kann über eine Liste zugänglich sein, deren Elemente ebenfalls dynamisch innerhalb des Speicherarrays angelegt werden. Hierdurch ist maximale Flexibilität gewährleistet.

In einer List enthält jede Datenstruktur einen Zeiger auf das nächste Datenelement. Man unterscheidet *offene Listen*, wo eine Datenstruktur mit einem NULL-Zeiger als Nachfolger das Ende der Liste markiert, und *geschlossene Listen*, wo das letzte Element wieder auf das erste zeigt. Die geschlossene Liste ist vollkommen symmetrisch und von jedem Element aus gleich anzusprechen, die offene Liste hat Anfang und Ende, um alle Elemente ansprechen zu können, muß bei einem bestimmten gestartet werden. In beiden Fällen ist eine zusätzliche Information von außen notwendig, um einen Einstieg in die Liste zu finden (sog. Rootpointer).

Die Funktion `getmem()` soll also bei jedem Aufruf einen neuen Speicherblock finden, eine neue Datenstruktur erstellen, diese in die Liste einfügen und einen Zeiger auf den Speicherblock zurückliefern. Ist kein Speicher mehr frei, soll die Funktion NULL liefern, damit der Aufrufer entscheiden kann, was in diesem Falle zu tun ist. Parallel zu `getmem()` ist auch eine Funktion `void freemem(char*memptr)` nötig, die einen Speicherblock wieder freigibt. Für weitergehende Ansprüche ist eine Funktion `char*changemem(char*memptr,int newsize)` praktisch, die einen bereits allozierten Speicherblock auf eine neue Größe hin verändert. Dies ist trivial, wenn die neue Größe kleiner ist als die alte, andernfalls kann es sein, daß die im Speicherblock vorhandenen Daten an einen anderen Platz kopiert werden müssen. Den Zeiger auf den neuen (ggf. alten) Speicherbereich soll `char*changemem()` zurückliefern.

# 7 Der Präprozessor

Bei dem Präprozessor handelt es sich um ein ursprünglich selbständiges Programm, das den Quellcode vorbehandelt und dann an den eigentlichen C-Compiler übergibt. Alle Kommandos an den Präprozessor beginnen mit einem `#`. Diese Präprozessorkommandos nicht Teil der Sprache C, sondern werden nur aus praktischen Gründen zusammen mit ihr verwendet. Sie haben zwar teilweise eine C-ähnliche Syntax, sind aber ausschließlich zeilenorientiert. Ein Kommando kann mehrere Zeilen umfassen, wenn das Zeilenendezeichen im Quelltext mittels \ (Backslash) entwertet wird.

## 7.1 `#include`

Diese häufigste Anwendung des Präprozessors wurde bereits in 3.5 in Zusammenhang mit der Modultechnik beschrieben. Mit dieser Anweisung wird an der aktuellen Stelle in angegebene Datei in den Quellcode eingefügt, gegebenenfalls wird dabei der Include-Pfad abgesucht.

## 7.2 `#define` ohne Parameter

`#define` definiert einen Text, der durch einen anderen ersetzt werden soll. Diese Definition ist ab dem Moment der Definition wirksam, auch ein innerhalb einer Funktion ausgeführtes `#define` ist global, da der Präprozessor keine Kenntnis vom C-Quellcode hat.

Für trauernde Pascal-Nostalgiker beispielsweise kann mittels `#define` etwas Trost gespendet werden:

```
#include <stdio.h>

#define begin           {
#define end             }
#define program         void main()
#define writeln         puts


program
begin
        writeln("Endlich wieder Pascal !");
        writeln("Oder etwa doch nicht ?");
end
```

Anmerkung: Einfach `main()` als `void` zu definieren damit kein `return`-Wert angegeben werden muß, ist eigentlich nicht gerade die feine Art, aber es sollte sowieso niemandem danach verlangen, das obige Beispiel zu programmieren ...

Etwas häufiger (und sinnvoller) ist `#define`, wenn Konstanten definiert werden sollen:

```
#define ARRAY_SIZE      100

int     array[ARRAY_SIZE];

         ...
        for(i = 0; i<ARRAY_SIZE; i++)
                array[i] = i;
         ...
```

Im Gegensatz zu enum können hier auch Fließkommakonstanten definiert werden:

```
#define PI      3.141592
```

(Diese Definition ist übrigens üblicherweise als M_PI in <math.h> bereits vorhanden.)
Eine andere „berühmte“ Konstante ist NULL:

```
#define NULL    0
```

Diese Zeile findet sich in einigen Standardincludedateien.

Der Nachteil von #define besteht darin, daß eine derartige Konstante global ist und ohne Einschränkung **alles** ersetzt:

```
#define array_size      100

int     array[array_size];

void    array_fill(int *array,int array_size)
{
int     i;
        for(i=0;i<array_size;i++)
                array[i] = i;
}
```

Dieses Beispiel bewirkt eine außerordentlich irreführende Fehlermeldung des Compilers. Um festzustellen, was wirklich geschieht, muß man sich jedoch nur die Funktionsweise des Präprozessors vor Augen halten. Im obigen Fall ist das Ergebnis allerdings auch so vorhersagbar. Was aber, wenn die Definition irgendwo in einem verschachtelten Header-File geschieht? Dann kann man sehr lange suchen ...

Um solche Fehler zu vermeiden, werden #define-Konstanten zumeist in Großbuchstaben geschrieben.

Anmerkung: Wird im obigen Beispiel die Konstante C-offiziell mit enum definiert, dann gibt es keine Probleme. Diese Methode ist allerdings - obwohl eigentlich besser - nicht üblich.

## 7.3 #if,#else,#elif,#endif

Mit #if können einzelne Blöcke bedingt compiliert werden. #if und #elif testen eine Konstante ab, die auch aus #define's aufgebaut sein können. Die Konstante muß bereits vom Präprozessor ausgewertet werden können und darf somit keine C-eigenen Werte (z.B. enum-Konstanten) enthalten. Wie in C gilt 0 als falsch, alles andere als wahr:

```
#define DEBUG   1

void    f(int *ptr)
{
#if     DEBUG
        printf("Parameter ptr=%p\n",ptr);
#endif
         ...
}
```

Durch Ändern eines einzigen Zeichens, der Konstanten 1 auf 0, kann der Debug-Mechanismus ausgeschaltet werden.

Ausdrücke in `#if` und `#elif` sind an die C-Syntax angelehnt:

```
#define DEBUG   2

void    f(int *ptr, int value)
{
#if     DEBUG==1
        printf("Parameter ptr=%p\n",ptr);
#elif   DEBUG==2
        printf("Parameter ptr=%p, value=%d\n",ptr,value);
#endif
         ...
}
```

Fast alle aus C bekannten Operatoren sind in `#if` und `#elif`-Ausdrücken erlaubt.

In C sind verschachtelte Kommentare nicht erlaubt. Dieses Manko kann man aber leicht mit den Präprozessorkommandos wettmachen, denn `#if`-Blöcke dürfen beliebig verschachtelt sein; ein `#if 0`-Block hat dabei denselben Effekt wie ein Kommentar und kann zudem schneller ein- und ausgeschaltet werden:

```
#include <stdio.h>

int     main()
{
#if     0
        puts("Das ist die main()-Funktion!\n");
# if    0
        puts("Dieser Text wird niemals (?)  ausgegeben...\n");
# endif
#endif

        return 0;
}
```

Anmerkung: Das `#`-Zeichen sollte nach dem älteren C-Standard immer am **Anfang** der Zeile stehen. Zwischen dem `#` und der eigentlichen Präprozessoranweisung kann hingegen beliebig viel Platz gelassen werden.

## 7.4 defined(), #ifdef, #ifndef

Um abzutesten, ob ein Präprozessortext bereits definiert wurde, gibt es die spezielle Präprozessorfunktion `defined()`, die „wahr“ liefert, wenn ihr Argument definiert wurde:

```
#if     !defined(ARRAY_SIZE)
#define ARRAY_SIZE      100
#endif

int     array[ARRAY_SIZE];
```

`#ifdef` bzw. `#ifndef` sind eine Abkürzung für diese recht häufige Anwendung:

```
#ifndef ARRAY_SIZE
#define ARRAY_SIZE      100
#endif
```

`#ifdef` kann eingesetzt werden, um Header-Dateien vor mehrfachem includieren zu schützen (das ist deren häufigste Anwendung). Dies ist immer dann nötig, wenn eine Header-Datei außer Deklarationen noch Definitionen enthält.
Headerdatei „`VAR.H`“:

```
#ifndef __VAR_H
#define __VAR_H
struct  entry
{
        char    *name;
        int     len;
};

extern  entry_print(struct entry*);
#endif  /*  __VAR_H */
```

Die Headerdatei „`VAR.H`“ kann nun beliebig oft includiert werden, sie wird nur ein einziges Mal im C-Quelltext erscheinen. Gerade bei mehrfach verschachtelten Include-Dateien ist es sonst fast unmöglich, nur ein einziges Includieren zu gewährleisten. Mithilfe dieser Methode kann hingegen jede Headerdatei bedenkenlos alle anderen includieren, die sie selbst benötigt. In der Verwendung spart das viel Zeit, da man im Hauptmodul nicht erst alle notwendigen `#include`'s selbst zusammensuchen muß.

Jeder Compiler definiert standardmäßig einige Konstanten, anhand derer die Programmierumgebung festgestellt werden kann. Unter DOS ist generell die Konstante `__MSDOS__` definiert, unter Unix ist die Konstante `unix` definiert, unter MS Windows üblicherweise die Konstante `_Windows`:

```
#include <stdio.h>

int     main()
{
#ifdef  __MSDOS__
        puts("Dieses Programm läuft unter MSDOS ");
#elif   defined(unix)
        puts("Dieses Programm läuft unter Unix ");
#endif
        return 0;
}
```

Welche Konstanten auf welchen Wert gesetzt sind, hängt vom jeweiligen Compiler ab.

Mithilfe dieser Compilerkonstanten ist es möglich, Programme zu schreiben, die sowohl compilerspezifischen Code enthalten (sogar Assembler), und dennoch insgesamt portabel sind!

## 7.5 #undef

Wird eine Präprozessordefinition nur für eine kurzen Bereich benötigt oder soll sie geändert werden, so kann sie mit `#undef` aufgehoben werden.

```
#ifdef  ARRAY_SIZE
#undef  ARRAY_SIZE      /* Aufheben der Definition */
#endif

#define ARRAY_SIZE      50
```

## 7.6 #define mit Parametern

Bei `#define` können auch Parameter angegeben werden, die dann in den Text eingesetzt werden. Derartige *Makros* sind immer dann praktisch, wenn ein komplizierterer Ausdrück einfach verwendet werden soll und der Aufruf einer Funktion ineffizient wäre. Da der Präprozessor keine Typüberprüfung durchführt, kann ein Macro auf alle Typen (im Gegensatz zu einer C-Funktion) gleich angewandt werden und ist damit um einiges allgemeiner.

```
#define add(a,b) a+b
#define sub(a,b) a-b

main()
{
int     i = add(2,3),
        j = sub(5,2);           /* Macro auf int's angewandt */

double  d = add(2.0,3.0),
        e = sub(5.0,2.0);       /* Macro auf double's angewandt */

        printf("2+3=%d, 5-2=%d\n",i,j);

        printf("2.0+3.0=%lg, 5.0-2.0=%lg\n",d,e);

        return 0;
}
```

Allzu oft vergessen, daß der Präprozessor einen reinen Textersetzer darstellt und **nichts** mit der Sprache C selbst zu tun hat. Problematisch wird das in Situationen wie diesen:

```
#define mul(a,b) a*b
#define div(a,b) a/b
main()
{
int a=1,b=2,c=3,d=4,
        i = mul(a+b,c+d),
        j = div(a+b,c+d);

        printf("i=%d, j=%d\n",i,j);
        return 0;
```

```
}
```

Die Ausgabe des obigen Beispiels wird stark von dem abweichen, was man erwartet. Um dem auf die Schliche zu kommen, überlege man sich, wie der Ausdruck `mul(a+b,c+d)` nach dem Einsetzen in das Makro `mul` aussieht. Zur Wiederholung: Durch `#define` wird *nur* **ein** *Text durch einen* **anderen** *ersetzt*, nicht mehr und nicht weniger.
FRAGE: Wie kann man das Makro `mul` sicher gestalten ?

## 7.7 `#error`

Mit dem Präprozessorbefehl kann eine Fehlermeldung des Compilers (eigentlich des Präprozessors) explizit erzwungen werden. Soll ein Programm beispielsweise nur unter MSDOS laufen, kann folgendes Konstrukt verwendet werden:

```
#ifndef __MSDOS__
#error  Dieses Programm ist nur fuer MSDOS geschrieben.
#endif
```

Diese Anweisung kann manche Anwender zur Verzweiflung treiben:

```
#ifdef  _Windows
#error  Dieses Programm laeuft nicht unter Windows.
#endif
```

## 7.8 Präprozessor-Operatoren

Der Präprozessor bietet zwei zusätzliche Operatoren an. Wird in einem Macro mit einem Parameter `param` der Ausdruck „`#param`“ verwendet, dann wird der Parameter **als String** eingesetzt („stringifiziert“).

```
#include <stdio.h>

#define DEBUG_EXPRESSION(param)          \
                printf("Der Ausdruck %s ergibt %d !\n",#param,param);

int     main()
{
int     a = 10, b = 20, c = 30;

        DEBUG_EXPRESSION(a+b*c);
        return 0;
}
```

Der Operator `#` ist ausschließlich in Präprozessormacros gültig.

Für größere „Schweinereien“ kann der Operator `##` verwendet werden. Damit werden die Namen zweier Argument zu einem neuen Namen zusammengefügt, sodaß beispielsweise ganze Kaskaden von ähnlichen Variablennamen erzeugt werden können. Dieser Mechanismus soll aber hier nicht weiter erläutert werden, da er wirklich selten gebraucht wird.

## 7.9 assert()

`assert()` ist ein standardmäßig vorhandenes Macro, daß in `<assert.h>` definiert wird. Sinn und Zweck dieses Macros besteht darin, an kritischen Programmstellen eine Überwachung zu bieten. Das Argument ist eine Bedingung, die auf Wahrheit getestet wird. Trifft sie nicht zu, gibt das Macro die fehlerhafte Bedingung, den Modulnamen und die Programmzeile aus. Danach wird das Programm abgebrochen. Dieses Macro ist als einfache Sicherheitshilfe gedacht an Stellen, bei denen aller Voraussicht nach kein Fehler auftreten sollte. Falls nach mehrmaligen Programmläufen wirklich niemals ein Fehler auftrat, kann der Text `NDEBUG` vor `#include <assert.h>` definiert werden; alle `assert()`-Aufrufe werden dann automatisch in Null-Anweisungen umgewandelt.

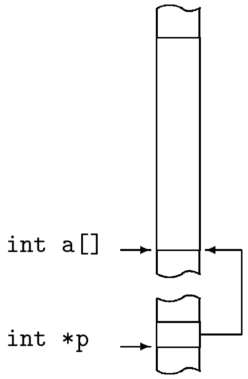
```
#include <assert.h>

int     Funktiondieimmereinsliefert(void)
{
        return 1;
}

int     main()
{
int     i;
        i = Funktiondieimmereinsliefert();

        assert(i==1);   /*  Hat i wirklich den Wert 1 ? */

        return 0;
}
```

# 8 Mehrfachzeiger und Funktionen mit variablen Parametern

## 8.1 Mehrfachzeiger

Ein Mehrfachzeiger ist ein Zeiger auf einen Zeiger auf einen Zeiger ...
Auf den ersten Blick mag dies unsinnig erscheinen, aber Mehrfachzeiger sind dennoch häufig von Nutzen, wie man am einfachsten Fall der doppelten Zeiger sehen kann.

Gemäß 6.5 läßt sich jeder Zeiger als Array interpretieren:




```
int *p;

    p[10] = 5;
```

Daher ist es naheliegend, daß ein mehrdimensionales Array einem mehrfachen Zeiger entspricht:

```
int     **p;

        p[10][4] = 5;
```

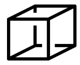
Um ein unbestimmtes Etwas zweimal indizieren zu können (damit es zumindest als Array erscheint), gibt es also folgende Möglichkeiten:

```
int     **pp;           /* pointer to pointer to int */
int     *ap[];          /* array of pointer to int */
int     aa[][];         /* array of array of int */
```

Tatsächlich kann jedes dieser Objekte zweimal indiziert werden. Aber so wie auch ein Zeiger und ein Array zwar sehr ähnlich, aber nicht identisch sind, gibt es auch hier einige nicht zu vernachlässigende Unterschiede.

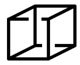
### 8.1.1 Mehrdimensionale Felder mit fixer Größe

Sie sind der am einfachsten überschaubare Fall. Bei der Definition muß jede Dimension in eigenen Arrayklammern angegeben werden:



```
double  Matrix[4][8];
```

Die in anderen Programmiersprachen häufig vorkommende Schreibweise



```
double  Matrix[4,8];
```

ist in C zwar auch möglich, bedeutet aber etwas völlig anderes (siehe 4.3 - der Sequenzoperator) und bewirkt zumeist nicht einmal eine Warnung, da es sich um gültige C-Syntax handelt.

Als Funktionsparameter werden Mehrfachfelder wie gewöhnliche Felder per Referenz übergeben :

```
void add(double dest[4][4],const double A[4][4],const double B[4][4]);
```

Da es in C keine Möglichkeit gibt, ein Array „per value“ zu übergeben, ist dies eine sinnvolle Anwendung der `const`-Spezifikation für Parameter, die nicht beschrieben werden sollen.

### 8.1.2 Ein Feld aus Zeigern

Der Nachteil eines mehrdimensionale Feldes fixer Größe besteht in der fixen Größe. Dies kann umgangen werden, wenn ein Feld aus Zeigern erstellt wird, die jeweils auf eine einzige Zeile zeigen. Auf diese Art können die einzelnen Zeilen beliebig im Speicher verteilt sein und brauchen nicht hintereinander liegen. Zudem kann jede Zeile unterschiedlich lang sein.

Aufgrund dieser ungleich vielfältigeren Möglichkeiten ist klar, daß ein Feld aus Zeigern etwas anderes ist als ein mehrdimensionales Feld fixer Größe, obwohl beide Konstrukte gleich indiziert werden können. Worin bestehen also die Unterschiede und was ist dennoch gemeinsam ?

Die folgenden Zeilen sollen dies verdeutlichen:

```
int     *iptr;          /*  Zeiger auf einen int, also ein int* */
int     p[100];         /*  Feld aus 100 int's */
int     *pz[10];        /*  Feld aus 10 int*'s */


        iptr = p;       /*  Zeiger ≡ Feld ! */

        pz[5] = iptr;   /*  der 5.te Zeiger im Feld pz */

        iptr[40] = 12;  /*  Arrayzugriff über einen Zeiger */

        pz[5][40] = 0;  /*  ebenfalls Arrayzugriff über einen Zeiger */
                        /*  pz erscheint wie zweidimensionales Feld */
```

Der **erste** Index beim Zugriff auf `pz` gehört somit zur fixen Felddimension und muß kleiner als 10 sein. Der **zweite** Index ist ein Zeigerzugriff und damit dynamisch veränderlich. Im obigen Fall darf der zweite Index maximal den Wert 99 erreichen.

Da die vom zweiten Index angesprochenen Elemente hintereinander im Speicher liegen, während der erste Index möglicherweise weit entfernte Zeilen adressiert, „läuft“ der zweite Index schneller als der erste. Das heißt, bei einer verschachtelte Schleife, die alle Element behandeln soll, sollte die innere Schleife den **zweiten** Index durchlaufen, denn nur dann kann der Compiler (oder der Programmierer selbst) diese in hocheffiziente reine Zeigerzugriffe optimieren:

```
int     i,j;

        for(j=0;j<10;j++)
                for(i=0;i<100;i++)
                        pz[j][i] = 0;
```

... ist optimierbar in:

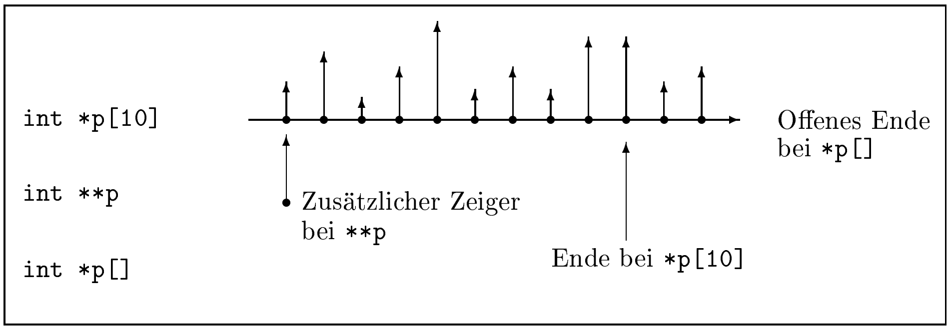
```
        for(j=0;j<10;j++)
        {
        int     *p = pz[j],
                *q = p+100;

                while(p<q)
                        *p++ = 0;
        }
```

was um einiges effizienter ist, da die aufwendige Indexberechnung `pz[j][i]` in der inneren, zeitkritischeren Schleife wegfällt.

Um ein Feld aus Zeigern ansprechen zu können, genügt ein Zeiger auf den Beginn dieses Feldes. Ein Feld aus Zeigern ist also ein Doppelzeiger, die folgenden Funktionsdeklarationen sind identisch:

```
void    f(int *p[10]);  /*  Feld mit 10 Zeigern */

void    f(int **p);     /*  Zeiger auf Zeiger */

void    f(int *p[]);    /*  Dimensionsangabe ist überflüssig! */
```

Die Speicherorganisation der verschiedenen Möglichkeiten von Variablendefinitionen (als Funktionsparameter sind alle identisch) soll folgendes Bild verdeutlichen:




Dabei ist die Syntax `int *p[]` nur möglich, wenn mit der Definition gleichzeitig eine Initialisierung erfolgt und damit indirekt die Größe festgelegt wird. Dies wird häufig verwendet, um ein *Array of Strings* zu definieren:

```
char    *string_array[] =
{
        "Basic ist schön.",
        "Pascal ist schöner.",
        "C ist am schönsten.",
        NULL
};
```

In diesem Beispiel ist keine weitere Zusatzinformation nötig, da das Ende der Elemente jeweils durch ein Sonderzeichen (`'\0'` bzw. `NULL`-Zeiger) erkenntlich ist.

Ebenfalls keine Größenangabe braucht bei der **Deklaration** angegeben werden:

```
extern  char    *s[];
```

Bei der **Definition** hingegen **muß** eine Größe angegeben werden, entweder als explizite Zahlenangabe oder als Initialisierungsarray. Die Abfrage der Größe mittels `sizeof(s)` ist nur bei Deklarationen mit expliziter Zahlenangabe möglich.

Das *Array of Strings* tritt sehr häufig auf, denn die Kommandozeile des kompilierten Programmes an `main()` wird auf diese Art übergeben:

```
#include <stdio.h>

/*  Dies ist die eigentlich korrekte Definition von main() ! */
int     main(int argc,char *argv[])
{
int     i;
        for(i=0;argv[i];i++)
                printf("Argument Nr.  %d:  %s\n",i,argv[i]);

        return 0;
}
```

Die einzelnen Kommandozeilenargumente werden in `char*argv[]` übergeben, das letzte Element ist immer `NULL`. Zusätzlich enthält der `int argc` die Anzahl der Argumente, sodaß `argv[argc] == NULL`.

ANMERKUNG: Die Namensgebung `argc` für „argument counter" und `argv` für „argument value" ist zwar allgemein üblich, aber keineswegs zwingend.

Neben diesen Kommandozeilenparametern wird der Funktion `main()` noch ein dritter Parameter übergeben, der das **Environment** („Umgebungsvariablen", die für das Betriebssystem und andere Programme relevant sind) beschreibt:

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

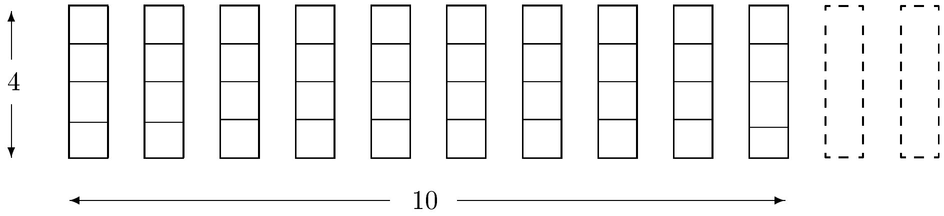
Dieser dritte Parameter ist aber nicht standardisiert, sodaß es besser ist, mit den Libraryfunktionen `getenv()` und `putenv()` auf das Environment zuzugreifen.

ÜBUNGSFRAGE: Welchen Sinn hat es, die Funktion `main()` zu deklarieren?

### 8.1.3 Ein Feld aus Feldern

Was für ein Feld aus Zeigern gilt, gilt natürlich ebenso für ein Feld fixer Größe: Der zweite Index läuft schneller. Das heißt, der Speicher muß in einem Feld fixer Größe so organisiert sein:

```
int     matrix[10][4];
```




Aus dieser Darstellung ist einsichtig, daß die gröbere, äußere Dimension (10) beliebig verändert werden kann, ohne daß dadurch die innere Struktur und damit die Indizierung beeinflußt würde. Der **erste** Index ist also eine nicht notwendige Information und darf daher als Funktionsparameter oder in einer Deklaration fehlen:

```
extern  int     matrix[][4];

void    clear(int matrix[][4],int indices)
{
register j;

        while(indices--)
                for(j=0;j<4;j++)
                        matrix[indices][j] = 0;

}
```

Da ein Array unbestimmter Größe einem Zeiger entspricht, erfüllt ein Zeiger auf ein Feld den gleichen Zweck:

```
extern  int     (*matrix)[4];

void    clear(int (*matrix)[4],int indices);
```

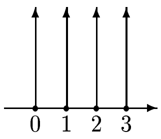
Achtung: Da der Arrayzugriffsoperator „`[]`“ eine höhere Priorität hat als der Dereferenzierungsoperator „`*`“, sind die Klammern hier zwingend!! Wird die Funktion `clear()` als

```
void    clear(int *matrix[4],int indices)
```

definiert, dann kann der Parameter `matrix` ebenfalls doppelt indiziert werden, sodaß bei der Funktion selbst nicht einmal eine Compilerwarnung auftritt. Dennoch ist ein *Feld aus Zeigern* (`int *matrix[4]` oder `int *(matrix[4])` ) etwas anderes als ein *Zeiger auf ein Feld* (`int (*matrix)[4]`), sodaß beim Aufruf der Funktion mit einem unpassenden Parameter der Compiler mindestens eine Warnung ausgeben sollte.

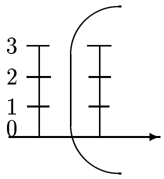
Die folgende Grafik soll die innere Struktur der entsprechenden Variablentypen verdeutlichen, die Typen in Klammern haben denselben Aufbau und lassen sich infolgedessen in die zugehörigen Typen konvertieren:

(`int **matrix`)
`int *matrix[4]`




`matrix[4][0-`$\infty$`]`

`int (*matrix)[4]`
(`int matrix[4][4]`)




`matrix[0-`$\infty$`][4]`

Welche Konversionen möglich sind, zeigt auch das folgende C-Fragment:

```
void    f(int *matrix[4] );     /*  array of pointers */
void    g(int (*matrix)[4]);    /*  pointer to array */


int **mptr;               /*  Mehrfacher Zeiger */
int m4x4[4][4];           /*  Mehrfaches Feld */

void    h()
{
        f(mptr);          /*  richtig */
        g(m4x4);          /*  richtig */

        f(m4x4);          /*  falsch */
        g(mptr);          /*  falsch */
}
```

Generell kann in einem mehrfachen Feld der erste Index unbestimmt bleiben, da die restliche Struktur dennoch eindeutig bestimmt ist. Es ist **nicht** möglich, *mehrere* Indices unbestimmt zu lassen, da so dem Compiler die Information fehlt, wie ein Element angesprochen werden soll:

```
void    clear(int matrix[][],int m,int n);
```

ist daher keine erlaubte Deklaration.

Um Felder mit variabler Länge in mehreren Dimensionen ansprechen zu können, ist eine explizite Berechnung der Position jedes Elementes nötig:

```
void    clear(int *matrix,int m,int n)
{
int     i,j;
        for(i=0;i<m;i++)
                for(j=0;j<n;j++)
                        matrix[i*n+j] = 0;      /* Indexberechnung */
}

int     m[4][4];

main()
{
        clear((int*)m,4,4);     /* Typenkonversion ist nötig ! */
        return 0;
}
```

Dies ist zwar keine besonders saubere Methode, da eine fehlerträchtige Typenkonversion nötig ist, aber die einzig mögliche, um dynamische Feldgrenzen in mehreren Dimensionen zu erhalten[7]. Diese Problematik kann erst unter C++ mithilfe der Datenabstraktion befriedigend gelöst werden.

Übungsbeispiel: Matrizenoperationen (Einlesen, Addieren, Multiplizieren, Ausgeben). Zur formatierten Ausgabe von Zahlenwerten sind die Zusatzangaben bei `printf()` (wie auch `fprintf()` und `sprintf()`) recht brauchbar:

| | |
|---|---|
| `printf("%5lg",val);` | gibt mindestens 5 Ziffern aus |
| `printf("%05lg",val);` | gibt mindestens 5 Ziffern mit führender 0 aus |
| `printf("%.5lg",val);` | gibt 5 Ziffern, ggf. gerundet, aus |

Vorschlag zu Funktionsweise des Programmes: Die Parameterzeile des Programms (`*argv[]`) könnte die Art und Weise der Matrizenoperationen auf Dateien festlegen, sodaß das fertige Programm selbst als universelles Werkzeug eingesetzt werden kann. Wenn das Programm beispielsweise „`matrix`“ genannt wird, soll mit der Anweisung `matrix c = a + b` am DOS-Prompt die Matrix aus der Datei „`a`“ mit derjenigen aus der Datei „`b`“ addiert werden und in die der Datei „`c`“ geschrieben werden. Ungültige Dateinamen wie „`.`“ können zur Eingabe via Tastatur/Ausgabe via Bildschirm verwendet werden (z.B.: `matrix c = .  + b` bedeutet, daß eine Matrix per Hand eingegeben werden muß). Welche Operationen (und Operatoren) realisiert werden, bleibt dem Programmierer überlassen.

[7]Unter GNU C können nicht nur Konstanten, sondern auch Variablen als Feldgrenzen bei der Definition eines Arrays verwendet werden und das Problem kann somit an den Compiler delegiert werden.

### 8.1.4 Zeiger auf Zeiger

Selten, aber doch sind Zeiger auf Zeiger nötig, ohne daß dies mit einem mehrdimensionalen Array etwas zu tun hätte. Dieser Fall tritt immer dann auf, wenn ein Zeiger *per reference* übergeben werden soll, beispielsweise in der Libraryfunktion `double strtod(const char*s,char**endptr)`, die via `endptr` die Möglichkeit einer bessere Fehlererkennung als `atof()` oder `scanf()` bietet.

Ansonsten werden Zeiger auf Zeiger eher selten verwendet, wenn nicht gerade irgendein geometrischer Effekt erwünscht ist, wie im folgenden Beispielprogramm (leider ist es ganz und gar nicht portabel).

Ein kleines Weihnachtsprogramm:

```
#include <stdio.h>

main()
{
char *p=NULL;
double X[4] =
              {1.70407621655723789e-306, 7.06362701922617297e-304,
              8.62464925270070124e-307,-5.53049347827401587e+303},
                                     *pp,
                                    **ppp,
                                   ***pppp,
                                  ****ppppp,
                                 *****pppppp,
                                ******ppppppp,
                               *******pppppppp,
                              ********ppppppppp;
                                      1;
                                     2+2;
                                    return
                                   ppp=&pp,
                                  pppp=&ppp,
                                 ppppp=&pppp,
                                pppppp=&ppppp,
                               ppppppp=&pppppp,
                              pppppppp=&ppppppp,
                             ppppppppp=&pppppppp,
                                 pp=X, printf(
                                      p,
                                    --*ppp,
                                   --**pppp,
                                  --***ppppp,
                                 ++****pppppp,
                                ++*****ppppppp,
                               ++******pppppppp,
                              ++*******ppppppppp,
                        (int(*)(int))p=(int(*)(int))pp);
}
```

## 8.2 Funktionen mit variablen Parametern

Funktionsparameter werden in C - wie in anderen Programmiersprachen auch - über den Stack übergeben. Anders als in Pascal etwa wird jedoch der **letzte** Parameter **zuerst** auf den Stack gelegt. Dies kann mit dem folgenden kleinen Programm leicht verdeutlicht werden:

```
#include <stdio.h>

int     f(void) {       return  puts("f()");    }
int     g(void) {       return  puts("g()");    }
int     h(void) {       return  puts("h()");    }

int     test(int i,int j,int k)
{
        return i+j+k;
}


int     main()
{
        test(f(),g(),h());

        return 0;
}
```

Viele, wenn auch nicht alle, Compiler werden hier zuerst die Funktion `h()`, dann `g()` und zuletzt `f()` aufrufen, da die Funktionswerte in dieser Reihenfolge auf den Stack gelegt werden müssen.

Durch diese „umgekehrte“ Reihenfolge liegt der erste Parameter immer an einer fixen Position relativ zum Stack der aufzurufenden Funktion – unabhängig von der Anzahl der Parameter kann dann in dieser Funktion der erste Parameter ausgewertet werden. Er kann dann Informationen über die folgenden Parameter enthalten, wie dies beispielsweise in der Funktion `printf()` geschieht.

In Abhängigkeit vom fixen Parameter kann sich die Funktion `printf()` langsam „nach unten“, zu weiteren Parametern durchhangeln. Dazu muß anfangs ein Zeiger auf den fixen Parameter gesetzt werden. Bei jeder Parameterauswertung (durch %-Sequenz im Formatstring bestimmt) muß dieser Zeiger um die Größe des zugehörigen Typs inkrementiert werden. Zum Glück muß dies der Programmierer nicht selbst machen, dazu gibt es die Macros `va_start`, `va_arg` und `va_end` aus `<stdarg.h>` (siehe auch Online-Hilfe).

Um anzuzeigen, daß nach einer Reihe fixer Parameter die folgenden beliebig sind, wird in der Funktionsdeklaration die sogenannte Ellipse (drei Punkte, „...“) verwendet:

```
int printf(const char *format, ...);
```

oder auch

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

Da die Interpretation der variablen Parameter in einer benutzerdefinierten Funktion geschieht, ist es verständlich, daß der Compiler nicht die Konsistenz mit dem Formatstring oder etwaiger anderer Kennungsmerkmale prüfen kann.

Weitere Standardfunktionen mit variablen Parametern sind die `execl???` – Funktionen, bei denen die Argumentliste mit einem `NULL`-Zeiger abgeschlossen werden muß.

Eher selten wird die Auswertung von Argumenten tatsächlich in einem Programm vorkommen, öfter jedoch soll eine Argumentliste über eine benutzerdefinierte Funktion an eine Standardfunktion durchgereicht werden, wie im kommenden Beispiel. Dazu können dann allerdings nicht die gewöhnlichen Funktionen wie `printf()` verwendet werden, da diese Funktion ja erwartet, daß die Argumentenliste auf einem fixen Platz relativ zum aktuellen Funktionsstack liegt. Zu diesem Zweck gibt es die Funktion `vprintf()`, die nicht eine Argumentenliste, sondern einen Zeiger auf eine Argumentenliste (Typ `va_list`, definiert in `<stdarg.h>`) erwartet:

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

void error(const char *fmt,...)
{
        va_list argptr;

        va_start(argptr, fmt);
        puts("Fehler:  ");
        vprintf(fmt, argptr);
        putchar('\n');
        va_end(argptr);
        exit(1);        /*  Programmabbruch mit Fehlercode 1 */
}

int     Funktiondieimmereinsliefert(void);

int     main()
{
int     i;

        i = Funktiondieimmereinsliefert();

        if (i!=1)
                error("Funktion lieferte den Wert %d !",i);

        return 0;
}
```

Hat eine Funktion **keine** fixen Parameter, so kann in C (aber nicht in C++ !) die Ellipse auch fortgelassen werden. Beim Aufruf findet dann keinerlei Überprüfung der Parameter statt; eine solche Funktion verhält sich dann so, als ob sie keinen Prototyp hätte. Dies ist daher der Defaulttyp für Funktionen, die ohne vorherige Deklaration aufgerufen werden.

# 9 Zeiger auf Funktionen

Zeiger auf Funktionen stellen gleichsam den Höhepunkt im Umgang mit Zeigern dar; während bei Datenzeigern eine falscher Adreßwert nur möglicherweise zu einem merkbaren Programmfehler führt, ist dies bei fehlerhaften Funktionszeigern mit Sicherheit der Fall. Doch wie in der Wirtschaft gilt auch hier: Je größer das Risiko, desto größer der Gewinn.

Die Definition eines Funktionszeigers erfolgt analog zu einer Funktionsdeklaration:

```
double        sin(double);
```

... deklariert eine Funktion mit einem Parameter vom Typ `double` und einem Rückgabewert gleichen Typs. Ein Zeiger auf eine derartige Funktion läßt sich wie folgt definieren:

```
double        (*f)(double);
```

`*f` ist der „Name“ der Funktion, `f` der Zeiger auf die Funktion. Dabei ist zu beachten, daß auch der Funktionsaufruf „`()`“ einen **Operator** darstellt und daher zur Unterscheidung von einer Funktionsdeklaration die Klammern um `*f` unverzichtbar sind. Ohne diese Klammerung würde eine Funktion namens `f()` deklariert werden, die einen Zeiger auf einen `double` liefert.

Um die Adresse einer Funktion zu erhalten, dürfen die Klammern zum Funktionsaufruf nicht angegeben werden, da „`&sin(1.0)`“ die Adresse des *Funktionswertes* bedeuten würde (was wenig sinnvoll ist, da der Funktionswert zumeist in einem Prozessorregister übergeben wird). Ohne Funktionsklammern ist auch der Adreßoperator überflüssig, da „`sin`“ keine andere Bedeutung haben kann als eben „`&sin`“:

```
f = sin;
```

ist die korrekte Zuweisung der *Adresse der Funktion* `sin()` an den Funktionszeiger `f`.

Soll die Funktion dann über den Zeiger aufgerufen werden, muß **zuerst** der Zeiger derefenziert werden und **dann** der „Inhalt“ des Zeigers aufgerufen werden (siehe Rangfolge der beteiligten Operatoren):

```
printf("Sinus(pi) = %lg\n", (*f)(3.141592) );
```

Ist ein Funktionszeiger erst einmal deklariert, kann er wie eine Funktion verwendet werden, sodaß auch die folgende alternative Syntax gültig ist:

```
printf("Sinus(pi) = %lg\n", f(3.141592) );
```

Da es kein „*array of functions*“ geben kann, ist mit Funktionszeigern keine Arithmetik möglich. Dies läßt sich auch dadurch erklären, daß der Operator „`sizeof`“ auf Funktionen nicht angewendet werden kann und daher die Formel aus 6.5 (Zeigerarithmetik) nicht ausführbar ist.

## 9.1 Funktionen als Funktionsparameter

Es hat natürlich wenig Sinn, Funktionszeiger zu verwenden, wenn man auch die Funktion selbst direkt aufrufen könnte (wie in der obigen `printf()`-Anweisung, die nur für `f==sin` ausgelegt ist). Etwas anderes ist es, wenn man eine Funktion schreiben möchte, die mit einer noch nicht bekannten Funktion operieren soll. Das folgende Beispiel demonstriert eine Funktion, die die numerische Ableitung einer mathematischen Funktion `double f(double)` an einer bestimmten Stelle `x` mit quadratischer Interpolation berechnet:

$$\left.\frac{d\,f}{d\,x}\right|_x = \lim_{\varepsilon\to 0} \frac{f(x+\varepsilon)-f(x-\varepsilon)}{2\varepsilon}$$

In C sieht dies folgendermaßen aus:

```
#define EPSILON         1E-10

double  df(double (*f)(double), double x)
{
        return ( (*f)(x+EPSILON) - (*f)(x-EPSILON) ) /(2*EPSILON);
}
```

Da diese Schreibweise doch etwas umständlich ist, kann sie abgekürzt werden, sodaß es erscheint, als würde eine Funktion selbst als Parameter übergeben werden. Dies ist natürlich unmöglich, der Parameter `f` im folgenden Beispiel ist wie oben ebenfalls ein Funktionszeiger:

```
double  df(double f(double), double x)
{
        return ( f(x+EPSILON) - f(x-EPSILON) ) /(2*EPSILON);
}
```

Der Aufruf der Funktion über den Funktionszeiger kann, wie bereits erwähnt, auf jeden Fall als Funktionsaufruf geschrieben werden.

Mit dem folgenden Hauptprogramm kann leicht überprüft werden, ob `df()` wirklich die Ableitung der übergebenen Funktion liefert und wenn ja, wie genau dies ist:

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

double  df(double f(double), double x);

int     main()
{
double  x;
        for(x = 0; x<M_PI_2; x += M_PI/8)
        {
                printf("x = %lg\t sin(x) = %lg\t dsin(x)/dx = %lg\n",
                        x        ,sin(x)        ,df(sin,x) );
                printf("x = %lg\t cos(x) = %lg\t dcos(x)/dx = %lg\n\n",
                        x        ,cos(x)        ,df(cos,x) );
        }

        return 0;
}
```

Anmerkung: Das obige Beispiel kann noch etwas optimiert werden, wenn bekannte Funktionen speziell behandelt werden:

```
#include <math.h>

double  df(double f(double), double x)
{
        if (f==sin)     return  cos(x);
```

```
        if (f==cos)     return -sin(x);

        return ( f(x+EPSILON) - f(x-EPSILON) ) /(2*EPSILON);
}
```

### 9.1.1 Übungsbeispiel: Verwendung von Quick-Sort

Dieser bekannte Algorithmus ist in der Libraryfunktion `qsort()` implementiert und braucht so in C nicht immer neu programmiert werden, wenn irgendein Array effizient sortiert werden soll. Die Funktionsdeklaration von `qsort()` in `<stdlib.h>` ist üblicherweise von der folgenden Form:

```
void qsort(void *data, int number_of_elements, int size_of_element,,
int (*compare)(const void *, const void *));
```

Das einzige, was dem Programmier zu tun bleibt, ist die Definition einer Funktion, die angibt, in welcher Beziehung zwei Elemente eines Feldes zueinander stehen. Diese Funktion wird an `qsort()` übergeben und von dort mit zwei Parametern aufgerufen, die auf zwei Objekte verweisen, die verglichen werden sollen. Da `qsort()` über keinerlei **Typ**information dieser Objekte verfügt, sind diese beiden Parameter `void*`-Zeiger. In der Vergleichsfunktion ist daher auf jeden Fall eine Typenkonversion notwendig[8].

Übungsaufgabe: Ein Feld beliebigen Typs (`char`, string, `int, double, ...`) soll mit Zufallswerten initialisiert und dann sortiert werden. Falls Strings sortiert werden sollen, ist zu beachten, daß `qsort()` nur Felder mit fixen Elementgrößen behandeln kann, also entweder nur ein Feld aus Zeigern oder ein Feld aus Strings mit fixen Längen sortieren kann (Vgl. Abschnitt 8).

Anmerkung: Wenn sehr viele Elemente sortiert werden müssen, wirkt sich der Funktionsaufruf bei jedem Vergleich bremsend aus. Ein neu programmierter Quick-Sort, der direkt auf den Elementen des Bildschirmspeichers operiert, ist merklich schneller. So fällt jedoch der Vorteil weg, bereits Gelöstes wiederverwenden zu können. Diese Problematik (Wiederverwendbarkeit versus Effizienz) kann erst in C++ mithilfe von `template`'s befriedigend gelöst werden.

## 9.2 Ein Feld aus Funktionszeigern

Oftmals treten in zeitkritischen Programmbereichen `if`-Anweisungen auf, die wegoptimiert werden können. Denn: Was ist schneller als eine `if`-Abfrage ? Antwort: **Keine** `if`-Abfrage! Anstatt eine Variable zu testen und je nachdem eine andere Funktion aufzurufen, kann diese direkt über einen Funktionszeiger erreicht werden. Ob eine Funktion statisch aufgerufen wird oder dynamisch über einen Zeiger, ist zeitlich irrelevant, denn auch im statischen Fall verwendet der Compiler einen internen Zeiger um die Funktion zu identifizieren.

Anstelle einer Reihe von `if`-Abfragen kann einfach ein *array of pointer to functions* indiziert und die entsprechende Funktion angesprungen werden. Jeder „Ast“ wird damit gleich schnell erreicht, bei `if`-Anweisungen sind weiter hinten gelegene Operationen mit mehr Abfragen verbunden und damit langsamer. Dieser Mechanismus ist ähnlich zur `switch`-Anweisung, allerdings dynamischer, da die Elemente des *array of pointer to functions* zur Laufzeit modifiziert werden können; als Nachteil (oder Vorteil, je nachdem, wie man dies sieht) ist kein „Durchfallen“ zum nächsten „`case`“ möglich.

Das folgende Anwendungsbeispiel stellt das Fragment eines Verfahrens dar, mit dem in der Computergraphik Texturen wie Holz, Marmor oder ähnliches simuliert werden können. Dazu wird ein gleichmäßiges Raster im $\mathbb{R}^3$ definiert, in der Art, daß jedem Punkt mit ganzzahligen

[8] Alternativ kann auch der Funktionszeiger beim Aufruf von `qsort()` typenkonvertiert werden, was allerdings den Funktionsaufruf kryptischer gestaltet.

Koordinaten ein fixer Zufallswert zwischen 0 und 1 zugeordnet wird. Diese Zufallswerte können, einmal berechnet, in einem statischen, konstanten Feld gespeichert sein, das mit den Koordinaten des Raumpunktes indiziert wird. Um für einen **beliebigen** Punkt einen eindeutigen Zahlenwert zu erhalten, muß der Wert zwischen den Rasterpunkten interpoliert werden. Auf diese Art ist im gesamten Raum für jeden Punkt ein Zahlenwert definiert, der lokal nur wenig variiert und dem nur noch ein Farbwert zugeordnet werden muß, damit eine Textur erscheint.

Hier soll nur das Prinzip der Interpolation im eindimensionalen Fall gezeigt werden; der Algorithmus ist leicht auf mehrere Dimensionen zu erweitern. Gegeben seien $f(0)$ und $f(1)$, gesucht ist $f(x)$ für $0 \leq x \leq 1$. Im einfachsten Fall kann dieser Wert linear interpoliert werden. Als Textur stellt sich jedoch heraus, daß so unrealistische Kanten an den Stützstellen entstehen. Um einen glatten Übergang zu erreichen, verwendet man eine kubische Interpolation, sodaß die Steigung des Interpolationspolynoms bei $f(0)$ und $f(1)$ jeweils 0 ist. Für manche Effekte ist aber dennoch das lineare Verfahren interessant; eine Umschaltung zwischen beiden Möglichkeiten ist daher wünschenswert. Da 1.) die Interpolation sehr schnell erfolgen soll und 2.) die Interpolationsfunktion mehrmals aufgerufen werden muß, bietet sich die Umschaltung mittels Funktionszeigern an.
Die Interpolationsfunktionen haben die folgende Gestalt:

```
static  double linear(double x)
{
        return x;               /*  linear(0) = 0, linear(1) = 1 */
}


static  double cubic(double x)
{
        return x*x*(3-2*x);     /*  cubic(0) = 0, cubic(1) = 1 */
}
```

Mit diesen beiden Funktionen kann nun leicht zwischen zwei Werten interpoliert werden:

```
/*  a ist der Wert für x=0, b für x=1. */
/*  Zurückgegeben werden soll der Wert an der Stelle 0<=x<=1  */
double  interpolate(double a,double b,double x,int cubic_interpolate)
{
static double (*ifunc[2])(double) = { linear, cubic };

        cubic_interpolate &= 1; /*  nur 0 oder 1 zulässig ! */

        return  a*ifunc[cubic_interpolate](x) +
                b*ifunc[cubic_interpolate](1-x);
}
```

Jeder Interpolationspunkt wird mit der Interpolationsfunktion gewichtet. In drei Dimensionen wird zwischen den 8 Punkten eines Rasterwürfels gemittelt, die Interpolationsfunktion muß genausooft aufgerufen werden.

Übungsaufgabe: Erweitere das Interpolationsverfahren auf zwei Dimensionen und stelle den Interpolationswert farblich als Funktion der Bildschirmkoordinate dar, z.B. (symbolische Schreibweise):

```
Pixel(x,y) = rgb(255,255,255) * Texture(x/30.0, y/30.0) );
```

Die Skalierung mit dem Zahlenwert `30.0` (z.B.) ist dabei nötig, da ja **zwischen** den Rasterpunkten interpoliert werden soll und die Bildschirmkoordinaten `(x,y)` selbst Rasterpunkte (weil ganzzahlig) darstellen.

## 9.3 Funktionszeiger als Rückgabewert

Es kommt vor, daß eine Funktion einen Funktionszeiger als Parameter enthält und beispielsweise einen statischen Funktionszeiger setzt, der auch von anderen Funktionen innerhalb eines Modules verwendet wird. Dann ist es sinnvoll, daß diese Funktion den alten Zeiger als Funktionswert liefert, damit der Aufrufer diesen speichern und ggf. wieder zurücksetzen kann.

Angenommen, es handle sich um einen Zeiger auf eine Funktion, die nichts (`void`) zurückliefert und einen `int` als Parameter hat. Der Funktionszeiger wird, wie bereits erwähnt, so definiert:

```
void (*funcptr)(int);
```

Die Variable vom Typ Funktionszeiger heißt hier `funcptr`. Nun soll aber eine Funktion definiert werden, die einen Funktionszeiger liefert. Der Einfachheit halber soll diese Funktion vorerst ein Argument vom Typ `double` (um Verwechslungen zu vermeiden) haben. Statt `funcptr` wird also einfach `func(double)` eingesetzt:

```
void (*func(double))(int);
```

`func` ist damit als eine Funktion deklariert, die einen `double` als Parameter besitzt und einen Zeiger auf eine Funktion liefert, die einen `int` als Parameter hat und nichts liefert.

Man könnte hiermit nun beide Funktionen „zugleich“ aufrufen, etwa:

```
func(3.0)(24);
```

aber das ist nicht der Sinn der Sache. Vielmehr soll die Funktion ja abermals einen Funktionszeiger als Parameter erhalten, der die neu zu setzende Funktion angibt:

```
void (*func(void (*f)(int)))(int);
```

Damit ist schlußendlich `func` eine Funktion, die als Parameter einen Zeiger auf eine Funktion besitzt, die einen `int` als Parameter hat und keinen Rückgabewert liefert, selbst aber einen Zeiger auf eine Funktion liefert, die einen `int` als Parameter hat und keinen Rückgabewert liefert.

Und bevor jetzt der Autor ob dieses kryptischen Konstruktes gelyncht wird, sollte man sich in der Online-Hilfe des jeweils verfügbaren Compiles mit der Hilfeseite zur Funktion „`signal()`“ vergewissern, daß derartige Konstrukte tatsächlich auch praktisch auftreten und gar nicht so selten gebraucht werden. Mithilfe der Funktion `signal()` können nämlich einige Interruptvektoren auf eigene Routinen verbogen werden. Tritt eine Ausnahmesituation ein, die auch explizit mit `raise()` erzeugt werden kann, wird die entsprechende Funktion angesprungen.

Unter DOS wird dieser Mechanismus nur halbherzig unterstützt, in Unix ist dies aber eine oft gebrauchte Möglichkeit um irgendwelche Prozesse zu irgendwelchen Aktionen zu veranlassen. Beispielsweise kann auch die segmentation violation (Signal `SIGSEGV`) abgefangen werden und so noch wichtige Daten auf die Festplatte geschrieben werden, bevor das Programm selbst endet. Als Ausnahmesituation gilt auch CTRL-C, das als Signal `SIGINT` auch unter DOS abgefangen werden kann.

## 9.4 Graphische Anwendungen

Wie sieht die Deklaration einer Funktion `f` aus, die folgendermaßen verwendet werden kann ?

```
f(0)(1)(2)(3)(4)(5)(6)(7)(8)(9);
```

Viel Spaß beim Rätseln !!

# Index