

Inhaltsverzeichnis

1	Aller Anfang ist schwer	3
1.1	Einleitung	3
1.2	Das erste Programm	3
1.3	Analyse des ersten Programmes	3
1.4	Kommentare und Lesbarkeit	5
2	Fundamentale Datentypen, Ein- Ausgabe	7
2.1	Fundamentale Datentypen	7
2.2	Ein- und Ausgabe	9
3	Operatoren	12
4	Der Präprozessor	16
5	bedingte Anweisungen	18
5.1	if-Anweisung	18
5.2	switch-Anweisung	20
6	Schleifen	22
6.1	while-Schleife	22
6.2	do-Schleife	22
6.3	for-Schleife	23
6.4	Sprunganweisungen	24
7	Zeiger, Arrays und Files	25
7.1	Zeiger	25
7.2	Arrays	25
7.2.1	Dynamische Speicherverwaltung	27
7.2.2	Strings	28
7.2.3	Mehrdimensionale Arrays	28
7.3	typedef	30
7.4	Files	30
8	Funktionen	34
8.1	Call by Value	36
8.2	Call by Reference	38

8.3	Bibliotheken, Header-Dateien	39
-----	------------------------------	----

A	Bedienung der Computer	43
A.1	Der login-Prozeß	43
A.2	Das Dateisystem	44
A.3	Der Editor	44
A.4	Programmieren in C	45

1 Aller Anfang ist schwer

1.1 Einleitung

Computer sind in der Lage bestimmte Operationen um ein vielfaches schneller auszuführen als der Mensch. Es obliegt jedoch dem Menschen zu entscheiden, was der Computer in welcher Reihenfolge tun soll. Ein mächtiges Hilfsmittel zur Bedienung von Computern sind die höheren Programmiersprachen; C ist eine davon. Wie jede normale Sprache hat C einen Grundwortschatz. Dieser kann durch eigene Befehle erweitert werden. Ein solches Konzept erlaubt es, die Sprache auf die individuellen Probleme anzupassen und einmal verfasste Programmteile in anderen Programmen wiederzuverwenden.

1.2 Das erste Programm

Betrachten Sie folgendes Programm :

```
#include <stdio.h>

main()
{ printf( "Hello World!\n" );
} p
```

Die obigen Zeilen sind der Quellcode oder Quelltext C-Programms. Wenn wir dieses Programm compilieren und anschließend starten, erzeugt es folgende Ausgabe auf dem Bildschirm:

```
Hello World!
```

Es ist nicht entscheidend zu verstehen, was während des Compilierens mit dem Quellcode passiert. Man sollte jedoch wissen, daß ein C-Quelltext kein ausführbares Programm darstellt, sondern erst mit Hilfe des Compilers zu einem ausführbaren Programm gemacht wird.

1.3 Analyse des ersten Programmes

Nun betrachten wir den Quelltext des Programms etwas genauer:

- `#include <stdio.h>`

Im Laufe des Programms wird mit Hilfe der Anweisung "printf" eine Ausgabe erzeugt. Dieser Befehl ist nicht Bestandteil des Grundwortschatzes, sondern muß dem Compiler gesondert erklärt werden. Dies geschieht durch obige Zeile.

Zusätzlich zum Grundwortschatz existiert eine große Menge von Befehlen, genauer Funktionen (vgl. Kap. 8), die mit Hilfe des Grundwortschatzes programmiert worden sind. Diese Befehle werden in separaten Dateien dem Compiler beigelegt. Hierzu gibt es einen Standard, der vorschreibt, welche Befehle vorhanden sein müssen und wie sie aufzufinden sind. Der Befehl "printf" fällt unter diesen Standard.

Natürlich kann sich jeder Programmierer selbst unabhängig vom Standard eigene Befehle erzeugen und diese bei Bedarf in das Programm einbinden. Wie dies funktioniert wird weiter unten erklärt.

- `main()`

Ein C-Programm enthält immer eine "Funktion" mit dem Namen "main". Ein Programm ohne main-Funktion ist nicht lauffähig.

- `{`

Die offene geschweifte Klammer zeigt hier den Beginn der Funktion main an.

- `printf("Hello World!\n");`

Der Befehl "printf" gibt einen String aus, d.h. eine Folge von Zeichen, der durch Hochkomma eingeschlossen wird. Der Text 'Hello World!' wird ausgegeben, während \n ein Sonderzeichen ist, welches für den Zeilenumbruch sorgt. Mit dem Semikolon wird das Ende eines Befehls angezeigt.

- }

Die geschlossene geschweifte Klammer zeigt hier das Ende der Funktion main an.

Beim Aufruf des Programms wird im wesentlichen die main-Funktion ausgeführt, d.h. die Befehle werden ausgeführt, die in dieser stehen. Außerhalb der geschweiften Klammern dürfen keine weiteren Befehle vorkommen.

1.4 Kommentare und Lesbarkeit

Das Hello World Programm hätte auch so präsentiert werden können:

```
#include<stdio.h>
main(){printf("Hello World\n");}
```

Zur besseren Lesbarkeit der Programme sollte man sich ein festes Layout überlegen. Hierzu gibt es keine festen Regeln, obwohl einige Regeln sinnvoll erscheinen:

- jeder Befehl bekommt eine eigene Zeile,
- von geschweiften Klammern eingefasste Blöcke einrücken.

Kommentare beginnen in C mit /* und enden mit */. Innerhalb dieser Zeichen dürfen beliebige Zeichenfolgen stehen. Zeilenumbrüche sind ebenfalls erlaubt.

Kommentare und Einrückungen dienen dazu, ein Programm leichter lesbar und verständlicher zu machen. Sie können aber auch bei zu häufigem Gebrauch die Lesbarkeit beeinträchtigen. Gute Kommentare zu schreiben, ist ebenso wie das Programmieren eine Fähigkeit, die geübt werden muß. Auch hierzu kann man einige Vorschläge machen.

- Zu Beginn der Datei steht ein Kommentar, der folgendes umfassen könnte:
 - Name der Datei, Versionsnummer
 - Kurzbeschreibung des Programms

- Betriebssystem und verwendeter Compiler
- Datum der Erstellung, Datum der letzten Änderung, Autor

- Ein Kommentar zu jeder nicht trivialen Funktion (vgl. Kap. 8), der die Aufgabe der Funktion und den verwendeten Algorithmus kurz beschreibt.
- Ein Kommentar zu jeder globalen Variablen (vgl. Kap. 2).
- Eine Anweisungsfolge, die ungewöhnlich oder schwierig ist, sollte man kurz erläutern
- und nicht viel mehr.

Beachtet man dies bei dem Hello World Programm, so wird daraus:

```
/* hello.c 1.0
   Das Programm gibt auf dem Bildschirm einen Text aus.
   Unix cc
   30.4.1777 Friedrich Gauss
*/
#include<stdio.h>

main()
{
    printf( "Hello World!\n" );
}
```

2 Fundamentale Datentypen, Ein- Ausgabe

2.1 Fundamentale Datentypen

Um innerhalb eines Programms Daten, z.B. das Ergebnis einer Rechnung speichern zu können, werden Variablen verwendet, denen vom Programmierer ein Name zugeteilt wird. Die Sprache C ist streng typisiert, d.h. jede Variable hat einen genau bestimmten Typ, der ebenfalls festgelegt werden muß.

Ein Name, auch Bezeichner genannt, kann aus einer beliebig langen Folge von Buchstaben und Ziffern bestehen. Das erste Zeichen muß ein Buchstabe sein; der Unterstrich `_` wird wie ein Buchstabe behandelt. Groß- und Kleinbuchstaben gelten als unterschiedlich.

Bezeichner werden nicht ausschließlich für Variablen verwendet. Andere Beispiele folgen noch.

C-Datentypen

Typ	Anwendungsbeispiele (Wertebereich)	Beispiel- Namen
int	Zähler für Schleifen, ganzzahlige Rechenoperationen ($-32768 \dots 32767$)	i,j,m,n
long int	Sehr große ganzzahlige Rechnungen ($-2147483648 \dots 2147483647$)	li,lj,lm,ln
float	Fließkomma Rechnungen ($-10^{38} \dots 10^{38}$)	x,y,z
double	Fließkomma Rechnungen ($-10^{308} \dots 10^{308}$)	x,y,z
char	Zeichen (z.B.: a...z, 0...9)	c,s

Die Datentypen `int`, `float`, `double` und `char` sind fundamentale Datentypen. Weitere Datentypen werden durch voranstellen der Wörter `short`, `unsigned`

und `long` erzeugt. Innerhalb eines Programms können nun Variablen definiert werden.

```
/* So werden Variablen deklariert, d.h.
   Ihre Namen werden eingefuehrt, aber sie
   haben noch keinen bestimmten Wert.
*/
int i;
float x, y;
char Zeichen;

/* Hier wird den obigen Variablen ein
   bestimmter Wert zugewiesen.
   (Initialisierung der Variablen)
*/
i = 2;
x = 3.14;
y = 100.001;
Zeichen = 'a';

/* Deklaration und Initialisierung koennen
   auch zusammen erfolgen.
   (Definition von Variablen)
*/
int k = 2;
int Index = 1;
double Volumen = 3.4;
```

Die Deklaration einer Variablen bedeutet, daß dem Programm mitgeteilt wird, wie ein Bezeichner zu interpretieren ist. Im obigen Beispiel etwa ist `i` als eine Variable des Types `integer` zu lesen.

Eine Deklaration besitzt außerdem einen Bezugsrahmen, in der sie gültig ist. Es gibt zwei Arten von Bezugsrahmen.

- Local: Der Name wird in einem Block deklariert. Ein Block beginnt in C mit `{` und endet mit `}`. Der Name darf dann nur in diesem Block und in allen darin eingeschlossenen Blöcken verwendet werden.

Diese Deklarationen müssen zu Beginn eines Blockes stehen und dürfen nicht mit anderen Befehlen vermischt werden.

- File: Ein Name wird außerhalb aller Blöcke definiert. Er kann dann im ganzen File, der Quelldatei, frei verwendet werden. Der Name wird in diesem Fall global genannt.

Wird in einem Block ein bereits außerhalb dieses Blockes deklarierter Name erneut in einer Deklaration verwendet, so überdeckt der neue den alten Namen.

```
int x;           /* globales x */

main()
{
    int x;       /* lokales x ueberdeckt globales x */
    x = 1;       /* Zuweisung an lokales x */
    {
        float x; /* ueberdeckt erstes lokales x */
        x = 2;   /* Zuweisung an zweites lokales x */
    }
    x = 3;       /* Zuweisung an erstes lokales x */
}
```

2.2 Ein- und Ausgabe

Mit der Einführung einer Variablen muß auch eine Möglichkeit bestehen, ihren Inhalt, z.B. auf dem Bildschirm, auszugeben, oder ihr mittels Tastatureingabe einen neuen Wert zuzuweisen. Dies geschieht mit den Befehlen "printf" und "scanf".

Den Befehl "printf" haben wir bereits bei der Besprechung des "Hello World" Programms kennengelernt. Dort haben wir einen Textstring ausgegeben. In einen solchen Textstring kann man nun bestimmte Symbole einfügen, sogenannte Formatanweisungen, die bei der Ausgabe durch den Wert einer gewünschten Variablen ersetzt werden. Für Variablen jeden Datentypes gibt es verschiedene Formatanweisungen. Uns reichen zunächst die Folgenden:

Formatanweisungen

Symbol	Datentyp und Ausgabeform
%d	int
%ld	long int
%f	float in Fließkommenschreibweise
%e	float in Exponentenschreibweise
%lf	double in Fließkommenschreibweise
%le	double in Exponentenschreibweise
%c	char
%s	für Strings

Zur Verdeutlichung folgendes Beispiel :

```
#include <stdio.h>

main()
{
    int i;
    float x, y;

    i = 4;
    x = 56.3;
    y = -23.4;
    printf( "Der Wert von i ist %d.\n", i );
    printf( "x = %e, y = %f\n", x, y );
}
```

Dieses Programm erzeugt die Ausgabe :

```
Der Wert von i ist 4.
x = 5.630000e+01, y = -23.400000
```

Nach dem gleichen Schema werden Variablen mit dem Befehl "scanf" über Tastatur eingelesen. Der jeweiligen Variablen muß zusätzlich der sogenannte Adreßoperator & vorangestellt werden.

```

#include <stdio.h>

main()
{
    int i;
    float x, y;

    scanf( "%d", &i );
    scanf( "%e %f", &x, &y );
}

```

Zu beachten ist jedoch, daß die Stringkonstante keinen Text enthält, da dieser nicht ausgegeben wird, d.h. die Ausgabe der folgenden Befehle ist gleich :

```

scanf( "%d", &i );
scanf( "Bitte geben sie einen Wert ein : %d", &i );

```

3 Operatoren

Operatoren dienen zur Verknüpfung von Operanden. Man benötigt sie, um aus bekannten Werten neue zu gewinnen. Sehr bekannte Beispiele für Operatoren sind der '+' und '-' Operator.

Die meisten Operatoren sind zweistellig, d.h. sie benötigen zwei Operanden, es gibt jedoch auch einstellige und einen dreistelligen Operator. Operanden sind die Objekte auf die die Operatoren angewendet werden. Operanden können in C Konstanten, Variablen, Funktionsaufrufe oder Ausdrücke sein. Ein Ausdruck ist die Verknüpfung mehrerer Operanden und Operatoren nach bestimmten Regeln. Im weiteren werden diese Regeln verdeutlicht, zunächst jedoch gibt es eine formale Beschreibung des syntaktischen Aufbaus von Ausdrücken :

1. Eine Konstante ist ein Ausdruck: **"Hallo", 100**
2. Eine Variable ist ein Ausdruck: **x, i**
3. Ein Funktionsaufruf ist ein Ausdruck, wenn die Funktion nicht vom Typ 'void' ist (vgl. Kapitel 8): **f(x)**
4. Ist A ein Ausdruck und ϕ ein einstelliger Präfixoperator, so ist auch ϕA ein Ausdruck: **++j**
5. Ist A ein Ausdruck und ϕ ein einstelliger Postfixoperator, so ist auch $A\phi$ ein Ausdruck: **j - -**
6. Sind A und B Ausdrücke und ϕ ein zweistelliger Operator, so ist auch $A\phi B$ ein Ausdruck: **a + b**
7. Ist A ein Ausdruck, so ist auch (A) ein Ausdruck: **(a + b)**

Jeder Ausdruck hat einen Rückgabewert. Dieser ist durch die verwendeten Operatoren und Operanden eindeutig festgelegt. Man erhält den Rückgabewert eines Ausdrucks durch seine Auswertung.

Ausdruck	Rückgabewert
4 + 6	10
3 * (-2)	-6
8 + 2 % 3	1

Der Rückgabewert ist der Hauptunterschied zwischen Ausdrücken und Anweisungen. Anweisungen haben keinen Rückgabewert und können daher nicht in anderen Ausdrücken verwendet werden. Ein wichtiger Punkt ist der Typ des Rückgabewertes, der ebenfalls eindeutig durch den Ausdruck bestimmt wird. Nicht jeder Operator ist für jeden Operanden geeignet oder sinnvoll. Immer dann, wenn die Operanden eines Ausdrucks unterschiedliche Typen haben, werden automatische Typkonvertierungen vorgenommen. Diese Konvertierungen werden auch implizite Typkonvertierungen genannt. Ein Ausdruck wird in C durch Anhängen eines Semikolons zu einer Anweisung, kann also im Prinzip für sich alleine stehen, ohne in einen Kontext eingebunden zu sein. So ist z.B.

```
main() {
    1 + 1;
}
```

ein korrektes (aber nutzloses) C Programm, da der Rückgabewert des Ausdrucks hier nicht weiter verwendet wird. Ein Beispiel für die Verwendung des Rückgabewertes wäre :

```
main() {
    int Rest;

    Rest = 7 % 3;
    printf ("Rest der Division 7 / 3 : %d", Rest);
}
```

Im folgenden eine Liste der wichtigsten Operatoren. Die Buchstaben A, B usw. bezeichnen im folgenden Ausdrücke.

Arithmetische Operatoren : + - * / stehen für Addition, Subtraktion, Multiplikation und Division. Der Ausdruck $A + B$ hat als Rückgabewert die Summe von A und B. z.B. $4 + 5$ hat den Rückgabewert 9. Es bleibt der Modulo-Operator %. Er ist nur für ganzzahlige Typen definiert. Der Ausdruck $A\%B$ hat den Rückgabewert $A \bmod B$, z.B. $41\%5$ den Wert 1.

Zuweisungsoperatoren : = ist der Zuweisungsoperator. Der Operator weist A den Wert von B zu. Der Rückgabewert ist B. Da die Zuweisung keine Anweisung sondern ein Ausdruck ist, kann man z.B. sehr einfach mehrfache Zuweisung realisieren. $A=B=C$ wird rechtsassoziativ ausgewertet, d.h. wie $A=(B=C)$, also wird zuerst B und dann A der Wert von C zugewiesen.

Arithmetische Zuweisungsoperatoren : Der Ausdruck $A+=B$ ist äquivalent zu $A=(A+B)$, d.h. zu A wird der Wert von B hinzuaddiert. Sein Rückgabewert ist $A+B$. Die anderen arithmetischen Operatoren funktionieren analog.

Inkrement- und Dekrementoperatoren : ++ und -- sind Post- und Präfixoperatoren. ++ erhöht A um 1, hat aber den Rückgabewert A. ++A erhöht ebenfalls A um 1, hat jedoch den Rückgabewert $A+1$. Der Operator -- funktioniert analog mit -1. Diese beiden Operatoren werden oft in Schleifen eingesetzt.

Relationale Operatoren : Diese dienen dazu, zwei Ausdrücke auf bestimmte Eigenschaften hin miteinander zu vergleichen. Besteht diese Eigenschaft, so nennt man den daraus resultierenden Ausdruck WAHR, sein Rückgabewert ist dann ungleich null. Sonst nennt man diesen Ausdruck FALSCH, sein Rückgabewert ist dann gleich null.

Es gibt folgende relationale Operatoren : ==, <, <=, >, >=, != mit der Bedeutung : gleich, kleiner, kleiner gleich, größer, größer gleich, ungleich.

Beispiele : $3==3$ ist WAHR, Rückgabewert ist ungleich null. $3<=2$ ist FALSCH, Rückgabewert ist gleich null. $3!=2$ ist WAHR usw.

Logische Operatoren : && und || sind die Umsetzung des mathematischen Und und Oder, d.h. $A\&\&B$ ist WAHR, wenn A und B WAHR ist. $A||B$ ist WAHR, wenn mindestens einer der Ausdrücke A und B WAHR ist. ! ist der Nicht-Operator, d.h. !A ist WAHR, falls A FALSCH ist.

Konditional-Operator : Er ist der einzige dreistellige Operator und hat die Form $A?B:C$. Es wird überprüft, ob der Ausdruck A WAHR oder FALSCH ist. Ist er WAHR erhält der Ausdruck $A?B:C$ den Wert B, ist er FALSCH den Wert C.

Cast-Operator : einstelliger Operator, der einer Variablen explizit einen neuen Datentyp zuweist, ohne ihren Wert zu ändern. Beispielsweise geben die Ausdrücke

```
(double) i
(int) x
```

den Variablen i und x die neuen Datentypen double bzw int.

Es gibt noch eine Reihe weiterer Operatoren, die bei Bedarf in späteren Kapiteln eingeführt werden.

4 Der Präprozessor

Beim Aufruf des Compilers wird der C-Quelltext zunächst automatisch durch ein anderes Programm, den sogenannten Präprozessor, bearbeitet. Der Präprozessor sucht nach Zeilen, die mit einem # beginnen und führt die dort stehenden Befehle aus. Diese Befehle sind nicht in C geschrieben, sondern haben eine eigene Sprache. Die Zeile mit dem Präprozessorbefehl

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

ist schon bekannt, und soll nun etwas genauer erläutert werden. Der Präprozessor wird hierdurch angewiesen, die betreffende Zeile durch den Inhalt der Datei stdio.h zu ersetzen. Es gibt einen Standardpfad zu dieser Datei, der dem Compiler bzw. Präprozessor bekannt ist. Der Inhalt von Dateien mit Endung .h wird in 8.3 beschrieben.

Ein weiterer Präprozessorbefehl ist #define.

```
#define NAME TEXT
```

Jedesmal wenn im Programmtext NAME auftaucht, wird dieser durch TEXT ersetzt. Das Programm

```
#include <stdio.h>
#define forever for(;;)

main () {
    forever printf ("Hallo\");
}
```

macht z.B. das gleiche, wie das Programm

```
#include <stdio.h>

main () {
    for(;;) printf ("Hallo\");
}
```

gibt nämlich ohne Ende Hallo auf dem Bildschirm aus.

An "define" können auch Parameter übergeben werden. Wurde in einem Programm die Zeile

```
#define MIN(a,b) ((a)<(b))?a:(b))
```

eingeführt, so expandiert

```
MIN(zahl1, zahl2);
```

zu dem Ausdruck

```
((zahl1)<( zahl2))?zahl1:( zahl2));
```

welcher das Minimum von zahl1 und zahl2 liefert. Eine solche Konstruktion mit Parametern nennt man ein Makro. Es fällt auf, daß bei der Definition eines Makros der Datentyp der Parameter nicht angegeben wird. Hierdurch kann der Compiler aber nicht überprüfen, ob das Makro korrekt erstellt wurde. Deshalb muß der Programmierer hier besonders vorsichtig sein, um schwer erkennbare Fehler zu vermeiden.

Makros können sich auch über mehrere Zeilen erstrecken. Wichtig ist hierbei, daß der Zeilenumbruch unmittelbar hinter dem Backslash erfolgt.

```
#define HALLO(anz) {
    int j;
    for (j = 0; j < anz; ++j) {
        printf ("Hallo\n");
    }
}
```

Dieses Makro würde in einem Programm bei Verwendung der Zeile

```
HALLO(100)
```

100 mal Hallo auf dem Bildschirm ausgeben. Siehe hierzu unter for-Schleife in Kapitel 6.

Bei der Verwendung von Makros ist zu beachten, daß zwischen dem Makronamen und der ersten Klammer " (" kein Leerzeichen sein darf.

5 bedingte Anweisungen

5.1 if-Anweisung

In den meisten Programmen ist es nötig, im Programmablauf in Abhängigkeit von zur Laufzeit ermittelten Werten zu verzweigen. Hierzu bietet C die if-Anweisung. Ihre Syntax lautet

```
if ( BEDINGUNG )
    ANWEISUNG1
[ else
    ANWEISUNG2 ]
```

Die if-Anweisung dient dazu, in Abhängigkeit vom Rückgabewert der Bedingung, die ein Ausdruck ist, entweder die Anweisung1 oder die Anweisung2 auszuführen.

Anweisung1 wird genau dann ausgeführt, wenn der Rückgabewert der Bedingung ungleich null, also WAHR ist. Ist der Rückgabewert gleich null, die Bedingung falsch, so wird Anweisung2 ausgeführt.

Auf den else-Teil der if-Anweisung kann verzichtet werden. In diesem Fall wird, falls die Bedingung FALSCH ist, die ganze if-Anweisung einfach übersprungen. Niemals werden beide Teile der if-Anweisung ausgeführt.

Zur Verdeutlichung zunächst zwei Beispiele :

```
#include <stdio.h>

main()
{
    int i, j;

    printf( "Geben sie zwei Zahlen ein : " );
    scanf( "%d %d", &i, &j );
    if (i == j) {
        printf( "Die Zahlen sind gleich.\n" );
    }
    else {
        printf( "Die Zahlen sind verschieden.\n" );
    }
}
```

Die Ausgabe des Programms hängt von den beiden Zahlen i und j ab. Es wird der Text

Die Zahlen sind gleich.

bei Gleichheit ausgegeben und

Die Zahlen sind verschieden.

bei Ungleichheit.

Das nächste Programm testet, ob das Polynom $ax^2 + bx + c$ (mit reellen Koeffizienten), komplexe oder reelle Nullstellen hat. Dazu werden zunächst die Koeffizienten eingelesen und dann die Diskriminante $D = \sqrt{b^2 - 4ac}$ ausgerechnet. Dabei muß man darauf achten, daß auch $D < 0$ möglich ist. An dieser Stelle wird mit Hilfe der `if`-Anweisung zu den entsprechenden Fällen verzweigt.

```
#include <stdio.h>
#include <math.h> /* fuer sqrt */
#define SQ(a) ((a)*(a))

main()
{
    double a, b, c, radikant;

    printf( "Koeffizienten zum Polynom\n ax^2+bx+c\n\n" );
    printf( "a = " ); scanf( "%lf", &a );
    printf( "b = " ); scanf( "%lf", &b );
    printf( "c = " ); scanf( "%lf", &c );

    radikant = SQ(b) - 4 * a * c;
    if (radikant >= 0) {
        printf( "Das Polynom hat reelle Loesungen.\n" );
        printf( "D=%lf\n", sqrt( radikant ) );
    } else {
        printf( "Das Polynom hat komplexe Loesungen.\n" );
        printf( "D=%lf\n", sqrt( -radikant ) );
    }
}
```

5.2 switch-Anweisung

Unter Umständen hat man zu mehreren "konstanten" Fälle zu verzweigen. Die bedeutet, zur Verzweigung zu diesen Fällen, ist ein Ausdruck mit einer Konstanten zu vergleichen. Hierzu ist nun die `switch`-Anweisung geeignet. Ihre Syntax lautet

```
switch ( AUSDRUCK ) {
case KONSTANTE1 :
    ANWEISUNG1
    break;
case KONSTANTE2 :
    ANWEISUNG2
    break;
...
[ default :
    DEFAULT-ANWEISUNG ]
}
```

Zur Verdeutlichung ein Beispiel :

```
#include <stdio.h>

main()
{
    int i, j;
    char c;

    printf( "Ausdruck : " );
    scanf( "%d %c %d", &i, &c, &j );
    printf( "Ergebnis : " );
    switch( c ) {
case '+' :
        printf( "%d", i + j );
        break;
case '-' :
        printf( "%d", i - j );
        break;
}
```

```

case '*' :
    printf( "%d", i * j );
    break;
case '/' :
    printf( "%d", i / j );
    break;
default :
    printf( "Unbekannter Operator." );
}
printf( "\n" );
}

```

Dieses Programm liest einen Ausdruck der Form $A\phi B$ ein. Um das Ergebnis ausgeben zu können, müßte man eigentlich schon vorher wissen, um welchen Operator es sich handelt. Hier kommt nun die switch-Anweisung zum Zuge. Wichtig ist hierbei das "break". Es führt dazu, daß die switch-Anweisung bei Übereinstimmung mit einer Konstanten nur diese Anweisungsfolge ausführt. Ansonsten würde noch mit allen weiteren Konstanten verglichen werden. Die default-Anweisung ist optional und wird hier nur ausgeführt, wenn vorher keine Übereinstimmung erzielt wurde. Bei den zum Vergleich aufgeführten Konstanten darf insbesondere keine doppelt vorkommen.

6 Schleifen

Oft müssen Anweisungen mehrmals durchlaufen werden. Hierzu stellt C die while-, die do- und die for-Schleife zur Verfügung.

6.1 while-Schleife

Die Syntax der while-Schleife lautet

```
while ( BEDINGUNG ) ANWEISUNG
```

und hat folgende Bedeutung.

1. Es wird geprüft, ob die Bedingung ungleich Null (WAHR) oder gleich Null (FALSCH) ist. Ist die Bedingung WAHR, wird bei 2. weitergemacht. Ist die Bedingung FALSCH, wird zum nächsten Befehl nach der while-Schleife gesprungen.
2. Es wird die Anweisung ausgeführt und dann zu 1. gegangen.

Ein Beispiel hierzu ist die Berechnung der Fakultät einer natürlichen Zahl n .

```

int i = 1;
long Fakultaet = 1;

while (++i <= n) {
    Fakultaet *= i;
}

```

6.2 do-Schleife

Die Syntax der do-Schleife:

```
do ANWEISUNG while ( BEDINGUNG );
```

In diesem Fall wird die Anweisung ausgeführt und dann die Bedingung überprüft. Ist die Bedingung WAHR, wird die Anweisung wiederum ausgeführt, andernfalls die Schleife verlassen.

Der Hauptunterschied zur while-Schleife liegt darin, daß die Anweisung auf jeden Fall einmal durchlaufen wird.

Wichtig ist, das Semikolon am Ende der do-Schleife nicht zu vergessen.

6.3 for-Schleife

Die for-Schleife hat folgende Syntax.

```
for (INITIALISIERUNG; BEDINGUNG; REINITIALISIERUNG)
    ANWEISUNG
```

Dies hat die Bedeutung:

1. Die Initialisierung wird durchgeführt.
2. Es wird überprüft, ob die Bedingung WAHR oder FALSCH ist. Ist die Bedingung FALSCH wird zum nächsten Befehl nach der for-Schleife gesprungen. Sonst wird Punkt 3 ausgeführt.
3. Die Anweisung wird ausgeführt. Anschließend erfolgt die Reinitialisierung, wonach wieder zu 2. gegangen wird.

Die Berechnung der Fakultät von n kann nun auch mit der for-Schleife durchgeführt werden.

```
int i;
long Fakultaet;

for (Fakultaet = 1, i = 1; i <= n; ++i) {
    Fakultaet *= i;
}
```

Zur Programmsteuerung kann immer sowohl eine for- als auch eine while-Schleife verwendet werden (siehe Bsp.). Welcher Typ ausgewählt wird, ist eine Frage des Geschmacks. Die Lesbarkeit des Programms ist bei dieser Wahl ein wichtiges Kriterium.

6.4 Sprunganweisungen

Es soll noch kurz auf die Anweisungen

```
break;
continue;
```

eingegangen werden.

Die break-Anweisung ist schon von der switch-Anweisung bekannt. Sie darf aber auch in Schleifen eingesetzt werden. Nach der Anweisung break führt das Programm die nächste Anweisung nach der Schleife aus.

Die continue-Anweisung springt ebenfalls an das Schleifenende, aber verläßt diese nicht. Die Schleife wird also weiterhin durchlaufen.

7 Zeiger, Arrays und Files

7.1 Zeiger

Wird in einem Programm ein Integer *i* definiert, so werden zur Speicherung des Wertes von *i* im Speicher des Computers mindestens zwei Bytes verwendet. Ein Byte ist eine Folge von 8 Bit, d.h. eine Folge von 8 Nullen oder Einsen, die der Computer als Speichereinheit verwendet. Als Adresse von *i* wird dann die Zahl bezeichnet, die angibt, ab wo die Bytes von *i* im Computer gespeichert sind. Für alle anderen Datentypen gilt Entsprechendes.

In C kann mittels sogenannter Zeiger, auch Pointer genannt, auf den Inhalt dieser Adressen direkt zugegriffen werden. Es können einige der bisher bekannten Operatoren verwendet werden. Man kann z.B. durch den Operator `=` einem Zeiger eine Adresse zuweisen oder durch den Operator `++` den Zeiger auf das nächste Datenelement im Speicher setzen.

Zusätzlich gibt es zwei neue einstellige Operatoren `&` und `*`. Der Ausdruck `&i` liefert die Adresse des Integers *i* und `*ip` ist der Inhalt der Adresse auf die ein Zeiger *ip* zeigt. Wie die Definition von Zeigervariablen zu erfolgen hat, wird nun vorgeführt.

```
double x; /* double-Variable */
double* xp; /* Zeiger auf double */

xp = &x; /* xp zeigt nun auf x */
*xp = 3.5; /* x wird der Wert 3.5 zugewiesen*/
```

7.2 Arrays

Die Konstruktion eines Arrays kann z.B. dazu dienen, Vektoren oder Matrizen zu speichern. Ein Array ist eine Menge mehrerer Elemente gleichen Typs, die unter einem Namen gespeichert werden. Beim Anlegen eines Arrays muß die Anzahl der Elemente (*Größe*) angegeben werden. Durch den Operator `[]` kann auf die Elemente des Arrays zugegriffen werden. Der Index-Bereich geht hierbei von 0 bis *Größe*-1.

```
int werte[80]; /* Array von 80 int's */
int Primzahlen[]={2, 3, 5, 7, 11} /* Array von 5 int's */
int Primzahlen[5]={2, 3, 5, 7, 11} /* oder auch so */
```

```
double* v[10]; /* Array aus 10 double-Pointern */

werte[0] = 77; /* das erste Element */
werte[79] = werte[0]; /* das letzte Element */
```

Der Array-Name kann auch als Zeiger auf das erste Element des Arrays angesehen werden, sodaß die beiden Anweisungen

```
werte[5] = 12;
```

und

```
*(werte + 5) = 12;
```

äquivalent sind.

Im Beispiel vorher wurde ein Array Primzahlen definiert. Bei solch einer Art der Definition ist zu beachten, daß der Zeiger konstant ist, d.h. er kann durch keine Operationen verändert werden.

Zeiger können zur Manipulation von Arrays verwendet werden. Im folgenden Beispiel wird ein Array initialisiert und dann auf zwei verschiedene Arten auf andere Arrays kopiert.

```
main()
{
  int u[5] = { 1, 4, 9, 16, 25};
  int v[5], w[5], i;

  for (i = 0; i < 5; ++i) { /* kopiert u auf v */
    v[i] = u[i]; /* mittels [] */
  }

  for (i = 0; i < 5; ++i) { /* kopiert u auf w */
    *(w + i) = *(u + i); /* mittels Zeigern */
  }
}
```

7.2.1 Dynamische Speicherverwaltung

Bisher haben wir Variablen durch Angabe ihres Namens und des gewünschten Datentyps eingeführt. Der hierzu im Computer benötigte Speicherplatz wurde automatisch bereitgestellt und nach Verlassen des Bezugsrahmens auch wieder freigegeben. Diese Vorgehensweise erfordert, daß zum Zeitpunkt der Programmerstellung die Größe der Variablen bekannt sein muß. Im Zusammenhang mit Arrays etwa kann dies zu Problemen führen, falls erst bei Ausführung eines Programms die benötigte Arraygröße bekannt ist. In diesen Fällen kann man wie folgt verfahren.

Es muß der zur Speicherung erforderliche Speicherplatz berechnet werden. Dies geschieht mit Hilfe des Operators `sizeof()`.

```
sizeof (typ)
```

liefert die Anzahl von Bytes, die `typ` zur Speicherung benötigt. Diese ist auch für die fundamentalen Datentypen nicht Standard, sondern hängt vom verwendeten Computer ab.

Die Funktion `malloc()` reserviert den erforderlichen Speicher. Der Aufruf

```
malloc (size);
```

liefert einen Zeiger auf einen Teil des Speichers für ein Objekt von *size* Bytes Größe. Das Argument ist `unsigned` und der Rückgabewert ist ein Zeiger auf `char`.

Schließlich wird durch die Funktion `free()` der reservierte Speicher wieder freigegeben. Geschieht dies nicht, so bleibt dieser Teil des Speichers auch nach Verlassen des Bezugsrahmens blockiert.

Im nächsten Beispiel wird ein Array von variabler Größe angelegt.

```
#include <stdlib.h> /* enthaelt malloc(), free() */

main ()
{
    int    i, N;
    double* vp;

    scanf ("%d", &N);    /* N noch nicht bekannt */
```

```
/* vp ist Zeiger auf double-Array von Laenge N */
/* wichtig!: typecast (double*) */
vp = (double*) malloc (N * sizeof (double));

/* Array wird auf Null gesetzt */
for (i = 0; i < N; i++) {
    *(vp + i) = 0;
}

free (vp);    /* Array loeschen */
}
```

7.2.2 Strings

Ein String, eine Folge von Zeichen, wird in C in einem `char`-Array gespeichert. Das Ende des Strings wird durch das Zeichen `\0` markiert.

```
#include <stdio.h>

main () {
    /* Definition zweier Strings */
    char Beispiel[] = "ein String";
    char myString[] = { 'A', 'B', 'C', '\0' };

    printf ("%s\n%s\n", Beispiel, myString);
}
```

Das Programm erzeugt die Ausgabe:

```
ein String
ABC
```

7.2.3 Mehrdimensionale Arrays

Mehrdimensionale Arrays sind Arrays von Arrays.

```

int a[100];          /* ein-dimensionales Array */
int b[3][5];        /* zwei-dimensionales Array */
int c[7][8][4];     /* drei-dimensionales Array */

b[1][3] = 4;
c[2][4][2] = b[1][3];

```

In zweidimensionalen Arrays werden typischerweise Matrizen gespeichert.

```

#include <stdio.h>

main () {
    int A[2][3] = { { 11, 12, 13},
                   { 21, 22, 23} };
    int i, j;

    for (i = 0; i < 2; i++) {
        for (j = 0; j < 3; j++) {
            printf ("A[%d][%d]=%d ", i, j, A[i][j]);
        }
        printf ("\n");
    }
}

```

Wir erhalten die Ausgabe:

```

A[0][0]=11  A[0][1]=12  A[0][2]=13
A[1][0]=21  A[1][1]=22  A[1][2]=23

```

Die dynamische Speicherung von Matrizen, siehe 7.2.1, ist etwas aufwendiger. Zur Speicherung einer Matrix A mit Einträgen double wird ein Array von Zeigern auf double benötigt.

```
double* A[];      /* Groesse variabel */
```

Diese Zeiger zeigen auf die Zeilen der Matrix, wobei die Zeilen als dynamisches Array realisiert werden.

```
A [i] = (double*) malloc (N * sizeof (double));
```

7.3 typedef

Es gibt die Möglichkeit, Definitionen übersichtlicher zu gestalten. Mittels einer typedef-Deklaration erhält ein Bezeichner die Bedeutung eines Datentypes. Dieser Bezeichner steht dann als Synonym für diesen Datentyp. Die typedef-Deklaration gilt innerhalb ihres Bezugsrahmens.

```

/* globale Deklaration */
typedef double real;
typedef real  matrix[3][3];
typedef int   index;

main ()
{
    matrix A;      /* 3x3-real-Matrix */
    index  i, j;
    real   x;

    x = 1.234;
    i = 1;
    j = 2;
    A[i][j] = x;
}

```

7.4 Files

Es besteht mit C die Möglichkeit auf Dateien (Files) zuzugreifen. So könnte man numerische Daten, die in einer Datei gespeichert sind, dort lesen und als Input für ein Programm verwenden. Ebenso könnte man den Output eines Programms in eine Datei schreiben. Bei großen Datenmengen ist dies eine übliche Methode.

Dazu existiert nun in C der Datentyp FILE, der in der Header-Datei stdio.h definiert wird. Der Zugriff auf eine Datei erfolgt dann durch einen Zeiger auf ein FILE-Element, dem vorher eine Datei zugewiesen worden ist. Bei der Zuweisung einer Datei zu einem FILE-Element muß angegeben werden, ob aus der Datei gelesen oder in sie geschrieben werden soll.

Der gesamte Vorgang wird unten am Beispiel eines Programms demonstriert,

welches die Quadrate der Zahlen Eins bis Zehn in eine Datei schreibt und diese danach wieder ausliest.

In eine Datei schreiben bedeutet, es wird dort ein Eintrag nach dem letzten Element eingefügt. Also entsteht eine Kette aus Einträgen. Das Lesen einer Datei beginnt mit dem ersten Element dieser Kette und wird beim darauf Folgenden fortgesetzt. Der Vorgang kann wiederholt werden, bis das Dateiende erreicht wird.

```
#include <stdio.h>

main ()
{
    FILE* source;          /* Pointer auf FILE */
    int i, q;

    /* Oeffnet bzw erzeugt Datei quadrat zum Schreiben. */
    source = fopen ("quadrat", "w+");

    for( i=1; i<=10; i++ ) {
        fprintf( source, "%d\n", i*i );
    }

    /* Datei schliessen. */
    fclose( source );

    /* Oeffne die Datei zum Lesen. */
    source = fopen ("quadrat", "r");

    for( i=1; i<=10; i++ ) {
        fscanf( source, "%d", &q );
        printf( "%d\n", q );
    }

    /* Datei schliessen. */
    fclose( source );
}
```

Zunächst wird ein Zeiger source auf ein FILE-Element deklariert. Diesem Zeiger wird durch den Befehl

```
source = fopen ("quadrat", "w+");
```

die Datei quadrat zugewiesen. Durch "w+" wird die Datei, falls sie noch nicht existiert, neu angelegt bzw. falls sie schon existiert überschrieben. Man sagt auch kurz, die Datei wird zum Schreiben geöffnet. Der Zugriff auf quadrat kann nun über den Namen source erfolgen. Durch

```
fprintf( source, "%d\n", i*i );
```

wird der Rückgabewert von $i * i$ in die Datei quadrat geschrieben. Analog zu printf kann man statt %d alle anderen Formatanweisungen verwenden. Um Daten aus der Datei quadrat lesen zu können, muß die Datei für den Lesezugriff geöffnet werden. Dies geschieht durch

```
source = fopen( "quadrat", "r" );
```

Mit Hilfe des Befehls fscanf kann nun, völlig analog zu scanf, aus quadrat gelesen werden. Es ist nützlich zu wissen, daß ein Ausdruck der Form

```
fscanf( source, "%d", i );
```

einen Rückgabewert besitzt. Wird versucht, aus einer Datei zu lesen, nachdem das letzte Element schon gelesen wurde, so ist der Rückgabewert gleich der Konstanten EOF (end of file). Auf diese Weise kann in einem Programm das Ende einer Datei erkannt werden.

Der Befehl fclose schließt die Datei, d.h. beendet den Zugriff über source. Es sei noch der Zeigerwert NULL erwähnt. Sollte nämlich das Öffnen einer Datei nicht funktionieren, z.B. weil die zu lesende Datei nicht existiert, so ist der Rückgabewert von fopen NULL, d.h. der Zeiger ist leer. Eine Abfrage, ob die Öffnung der Datei erfolgreich war, erhöht die Stabilität des Programms.

```

#include <stdio.h>

main()
{
    FILE* source;

    source = fopen( "quadrat", "r" );

    if (source==NULL) {
        printf( "Die Datei konnte nicht geoeffnet werden.\n" );
    } else {
        printf( "Die Datei wurde erfolgreich geoeffnet.\n" );
        fclose( source );
    }
}

```

8 Funktionen

Unsere bisherigen Programme bestanden nur aus einem Block, der Funktion main. Um einen bestimmten Programmteil mehrmals auszuführen, war es nötig, ihn in eine Schleifenkonstruktion einzubetten. In diesem Kapitel wird nun das Funktionskonzept in C erläutert, mit dessen Hilfe es möglich wird, eine vorher bestimmte Anweisungsfolge an einer beliebigen Stelle im Programm aufzurufen.

Bevor eine Funktion aufgerufen werden kann, muß diese natürlich deklariert worden sein. Die Definition der Funktion, damit ist die Anweisungsfolge gemeint, kann an einer beliebigen Stelle im Programm stehen. Deklaration und Definition einer Funktion haben beide global zu erfolgen.

Bei der Deklaration muß angegeben werden, von welchem Typ die Parameter sind, von denen die Funktion eventuell abhängt und welchen Typ ein möglicher Funktionswert hat. Durch einen neuen Datentyp void wird hier entweder bestimmt, eine Funktion ist unabhängig von Parametern oder eine Funktion besitzt keinen Funktionswert. Anhand der nächsten Beispiele sollte das Prinzip klar werden.

```

/* Deklaration einer Funktion f, die von
   einer double-Variablen abhaengt und
   deren Funktionswert vom Typ int ist.
*/
int f (double);

/* g haengt von zwei double und einem
   int ab und besitzt als Funktionswert
   einen char-Pointer.
*/
char* g (double, double, int);

/* Die Funktion HALLO erwartet einen
   int, hat aber keinen Funktionswert.
   (vgl. mit Makro HALLO)
*/
void HALLO (int);

```

```

/* Die Funktion Beep erwartet weder
   Parameter noch hat sie einen
   Funktionswert.
*/
void Beep (void);

```

In der Definition einer Funktion müssen die Parameter benannt sowie der Block der Funktionsanweisungen aufgeführt werden. Wir beginnen mit einem einfachen Beispiel einer parameterlosen Funktion.

```

#include <stdio.h>

/* Deklaration der Funktion bars */
/* (vor ihrer Verwendung)      */
void bars (void);

main ()
{
    bars();    /* Aufruf von bars */
    printf( "Hello, World !\n" );
    bars();    /* nochmal bars   */
}

/* Definition der Funktion bars */
/* (nach ihrer Verwendung)      */
void bars (void)
{ printf( "-----\n" );
} p

```

Das Programm erzeugt die Ausgabe :

```

-----
Hello, World!
-----

```

8.1 Call by Value

Im nächsten Beispiel wird eine Funktion berechnet, die von einem Parameter des Typs double abhängt. Wir betrachten eine lineare Funktion der Form $f(x) = ax + b$ mit reellen Werten a und b .

```

#include <stdio.h>

/* Die Funktion wird definiert und
   dadurch gleichzeitig auch deklariert.
*/
void berechne_f (double p)
{
    double a, b, c;

    a = 3.0;
    b = 2.0;
    c = a * p + b;

    printf ("Das Ergebnis ist %lf.\n", c);
}

main ()
{
    double x;

    printf ("Argument der Funktion eingeben : x = ");
    scanf ("%lf", &x);
    berechne_f (x);
}

```

Dieses Programm erzeugt bei der Eingabe entsprechender Werte die folgende Ausgabe :

```

Argument der Funktion eingeben : x = 1.0
Das Ergebnis ist 5.000000.

```

Damit das Ergebnis berechnet werden kann, kopiert das Programm den Wert von x in die Variable p . Am Ende der Funktion `berechne_f` wird die Variable

p wieder gelöscht, wie auch die Variablen a, b und c. Einen Funktionsaufruf, bei dem nur der Wert der Variablen verwendet wird, nennt man daher *Call by Value*.

Bleiben wir noch etwas bei dem letzten Beispiel. Sagen wir, man möchte den Funktionswert von *f* einer Variablen *y* zuweisen. Mit der obigen Konstruktion ist dies jedoch nicht möglich. Der Funktionswert von *f* war zwar in der Variablen *c* enthalten, aber man konnte nicht auf ihn zugreifen. Dies soll nun aber ermöglicht werden. Dazu modifizieren wir das Beispiel wie folgt :

```
#include <stdio.h>

double f (double p)
{
    double a, b, c;

    a = 3.0;
    b = 2.0;
    c = a * p + b;
    return c;          /* NEU */
}

main()
{
    double x, y;

    printf ("Argument der Funktion eingeben : x = ");
    scanf ("%lf", &x);
    y = f (x);
    printf ("Der Funktionswert ist %lf.\n", y);
}
```

Diese Version des Programms erzeugt ebenfalls die Ausgabe :

```
Argument der Funktion eingeben : x = 1.0
Der Funktionswert ist 5.000000.
```

Mit dieser Konstruktion könnte man nun, wie im Kapitel über Operatoren bereits angedeutet, Funktionswerte in Ausdrücken verwenden.

8.2 Call by Reference

Bisher wurde der Wert einer Variablen durch den Aufruf einer Funktion nicht geändert. Das lag an dem Übergabemechanismus *Call by Value*.

Damit die Funktion den Wert verändert, muß man einen Weg finden, die Variable selbst und nicht ihre Kopie an die Funktion zu übergeben. Dazu übergibt man der Funktion die Adresse einer solchen Variable, also den Zeiger auf sie.

Ein einfaches Beispiel ist die Vertauschung zweier Variablen durch eine Funktion *swap* :

```
#include <stdio.h>

void swap( double* xp, double* yp )
{
    double tmp;

    tmp = *xp;
    *xp = *yp;
    *yp = tmp;
}

main()
{
    double x = 1,
           y = 2;

    printf ("x=%lf y=%lf\n", x, y);
    swap(&x, &y);
    printf ("x=%lf y=%lf\n", x, y);
}
```

Die Funktion *swap* bekommt als Parameter die Adressen von *x* und *y*. Mit Hilfe der Adressen wird nun der Wert der Variablen *x* und *y* selbst geändert. Dementsprechend lautet die Ausgabe des Programms

```
x=1.000000 y=2.000000
x=2.000000 y=1.000000
```

Dieser Übergabemechanismus wird *Call by Reference* genannt. Man übergibt der Funktion nicht den Variablenwert, sondern die Referenz bzw. den Zeiger auf die Variable.

Zum Schluß noch ein paar Hinweise. Man kann Funktionen von Funktionen aus aufrufen und die Übergabemethoden Value und Reference mischen. Die Übergabe Call by Reference ist mit Vorsicht zu genießen. Die Manipulation eines Zeigers kann schnell zu einem schweren Programmfehler führen, wenn dieser auf einen nicht definierten Speicherteil zeigt. Es ist daher eine Empfehlung, falls immer möglich *Call by Value* zu benutzen.

8.3 Bibliotheken, Header-Dateien

Die Funktion `swap` ist unabhängig vom Rest des Programms. Man könnte sie in einem beliebigen anderen Programm verwenden. Dazu müßten wir eigentlich den gesamten Code nochmals in das neue Programm schreiben. Es besteht die Möglichkeit, solchen Mehraufwand zu umgehen.

Die Definition der Funktion `swap` wird in eine eigene Datei geschrieben. Ein vernünftiger Name für diese Datei wäre etwa `swap.c`. Diese Datei wird kompiliert, ohne daraus ein ausführbares Programm zu machen, sondern zu einer Datei `swap.o`.

Um nun die Funktion `swap` in einem Programm benutzen zu können, wird diese dort lediglich deklariert. Beim Compilieren dieser Datei wird dann nur noch die Datei `swap.o` angegeben, und der Compiler findet dort die zugehörige Definition.

Die genaue Vorgehensweise, welche Befehlszeilen in den Computer einzugeben sind, wird im Anhang erläutert.

An dieser Stelle soll etwas genauer die Arbeitsweise des Compilers geschildert werden. Wir haben bisher immer davon gesprochen eine Datei wird zu einem ausführbaren Programm kompiliert. Dieser Prozeß ist in mehrere Schritte unterteilt.

1. Der Präprozessor: Dieser erste Schritt wurde in Kapitel 3 geschildert.
2. Der Compiler: Er übersetzt den vom Präprozessor veränderten Quelltext in eine vom Computer direkt zu verstehende Sprache. Das Resultat wird in einer Datei mit Endung `.o`, einer sogenannten Objektdatei, gespeichert. Für die Quelldatei `BEISPIEL.c` wird etwa eine Datei `BEISPIEL.o` erzeugt, die vom Computer direkt gelesen werden kann.

3. Der Linker: Hier wird aus `BEISPIEL.o` ein ausführbares Programm gemacht. Genauer gesagt mit Hilfe von einer oder gegebenenfalls auch mehreren Objektdateien wird ein ausführbares Programm erzeugt. Wie im Beispiel oben mit der Datei `swap.o` ist es nämlich manchmal nötig, ebenfalls Funktionen aus anderen Dateien zu verwenden. Man sagt, diese Dateien werden zu einem ausführbaren Programm zusammengebunden. Zu beachten ist dabei, daß in allen Dateien zusammen die Funktion `main` genau einmal auftauchen darf.

Für unser obiges Beispiel muß also zunächst eine Datei mit der Funktionsdefinition erstellt werden.

```
/* Datei swap.c: */

void swap (double* xp, double* yp)
{
    double tmp;

    tmp = *xp;
    *xp = *yp;
    *yp = tmp;
}
```

Das folgende Programm macht dann genau das gleiche wie die Version von oben.

```
/* Hauptprogramm */

#include <stdio.h>

void swap (double*, double*);

main ()
{
    double x = 1,
           y = 2;

    printf ("x=%1f y=%1f\n", x, y);
}
```

```

    swap (&x, &y );
    printf ("x=%lf y=%lf\n", x, y);
}

```

In eine Datei wie swap.c könnten auch mehrere Funktionen auf einmal geschrieben werden. So lassen sich große Sammlungen mit selbst definierten Funktionen erstellen.

In einer weiteren Stufe ist es dann noch möglich mehrere Dateien mit Funktionsdefinitionen zu einer einzigen Datei, einer Bibliothek, zusammenzuschließen. Gewisse Bibliotheken sind standardmäßig jedem Compiler beigelegt.

Will man Funktionen einer Bibliothek verwenden, so ist es natürlich nach wie vor nötig diese im Programm zu deklarieren. Dies kann bei vielen Funktionen mühsam sein, zumal die genaue Abhängigkeit von möglichen Parametern bekannt sein muß. Daher ist es üblich sämtliche Deklarationen der Bibliothek in eine sogenannte Header-Datei zu schreiben, die gewöhnlich die Endung .h besitzt. Wir kennen schon die Header-Dateien stdio.h und math.h.

Unser kleines Beispiel könnte also so aussehen:

```

/* Datei swap.h:                */
/* Deklaration der Funktion swap */

void swap (double*, double*);

```

```

/* Datei swap.c:                */
/* Definition der Funktion swap */

void swap (double* xp, double* yp)
{
    double tmp;

    tmp = *xp;
    *xp = *yp;
    *yp = tmp;
}

```

```

/* Hauptprogramm */

```

```

#include <stdio.h>
#include "swap.h"

```

```

main ()
{
    double x = 1,
           y = 2;

    printf ("x=%lf y=%lf\n", x, y);
    swap (&x, &y);
    printf ("x=%lf y=%lf\n", x, y);
}

```

Der Präprozessor-Befehl #include ersetzt die jeweilige Zeile durch den Inhalt der angeführten Datei. Die Anführungszeichen weisen den Compiler darauf hin, daß swap.h im aktuellen Verzeichnis zu finden ist. Dateien in spitzen Klammern sind in einem Standard-Include-Verzeichnis enthalten.

A Bedienung der Computer

A.1 Der login-Prozeß

Anders als bei einem PC muß einem UNIX-System ein Benutzer bekannt sein. Jeder Benutzer, auch User genannt, besitzt dort einen eigenen Namen, über den er angesprochen wird. Zusätzlich hat er sich ein Paßwort zu merken, welches unbedingt geheim zu halten ist.

Auf einem Computer können unter UNIX mehrere User gleichzeitig arbeiten. Allerdings braucht jeder dazu einen eigenen Monitor und eine Tastatur, die mit dem Computer direkt oder über ein Datennetz verbunden sind. Jeder Computer in diesem Netz besitzt ebenfalls einen eigenen Namen.

Nachdem mit Hilfe des Computernamens die Verbindung hergestellt worden ist, erscheint auf dem Bildschirm der sogenannte login-Prompt:

```
login:
```

Hier ist der Benutzername einzugeben. Nach drücken der Eingabetaste muß das Paßwort eingegeben werden:

```
password:
```

Das Paßwort erscheint nicht bei seiner Eingabe.

War dies Alles erfolgreich, ist der UNIX-Prompt \$ zu sehen und hier können nun die weiteren Befehle eingegeben werden.

Mit dem Befehl

```
$ passwd
```

läßt sich das Paßwort ändern. Eine genaue Beschreibung einzelner Befehle erhält man über den Befehl man. Für eine Erklärung von passwd ist hierzu die Zeile

```
$ man passwd
```

einzugeben. Durch drücken der Leertaste wird der Hilfetext durchgeblättert und mit der Taste q die Hilfe beendet.

Das Verlassen eines Unix-Systems erfolgt durch das Kommando

```
$ exit
```

A.2 Das Dateisystem

Jeder Benutzer hat auf dem Computer ein eigenes Verzeichnis, welches HOME-Verzeichnis genannt wird. In diesem können andere Verzeichnisse oder Dateien erstellt werden. Hier befindet man sich auch zuerst nach dem login-Prozeß. Durch den Befehl

```
$ mkdir program
```

wird zum Beispiel das Verzeichnis program erstellt. Auf eine Datei example in diesem Verzeichnis könnte über den Begriff program/example zugegriffen werden. Mit dem Befehl

```
$ cd program
```

kann in dieses gewechselt werden, d.h. nun ist ein Zugriff alleine durch den Dateinamen example möglich. Das Verzeichnis in das jeweils gewechselt wird, wird aktuelles Verzeichnis genannt. Der Inhalt des aktuellen Verzeichnisses kann mit

```
$ ls
```

angezeigt werden und den Namen des aktuellen Verzeichnisses erhält man durch

```
$ pwd
```

Soll eine Datei einen neuen Namen erhalten, geschieht dies durch den Befehl:

```
$ mv altername neuername
```

Dateien und Verzeichnisse lassen sich mit **rm** bzw. **rmdir** wieder löschen. Dieser Befehl ist mit großer Vorsicht zu verwenden. Einmal gelöschte Daten können nicht wiederhergestellt werden.

A.3 Der Editor

Der Editor ist ein Programm, durch das Textdateien erstellt werden. Der C-Quelltext ist hierfür ein Beispiel.

A.4 Programmieren in C

Der Quelltext eines Programms wird in einer Textdatei mit der Endung `.c` gespeichert. Ein möglicher Name wäre etwa

```
beispiel.c
```

Der Quelltext dieser Datei muß in eine für den Computer lesbare Sprache übersetzt werden. Diese Arbeit übernehmen Compiler. Dies sind Computerprogramme, die von verschiedenen Herstellern erhältlich sind und dadurch auch unterschiedlich zu bedienen sind. Im Folgenden wird der frei verfügbare GNU-Compiler beschrieben.

Durch den Befehl

```
$ gcc beispiel.c
```

wird von dieser Datei ein ausführbares Programm erstellt, das den Namen `a.out` erhält. Diese kann unmittelbar aufgerufen werden und das Programm wird gestartet.

Damit nur die ersten beiden Schritte des in 8.3 beschriebenen Prozesses ausgeführt werden, verwendet man den Befehl

```
$ gcc beispiel.c -c
```

welcher eine Datei `beispiel.o` erzeugt.

Das Linken von zwei Objektdateien erfolgt beispielsweise durch:

```
$ gcc main.o beispiel.o
```

Mit der Datei `main.o` soll angedeutet werden, daß genau eine Datei die `main`-Funktion enthalten muß.

Falls in `beispiel.c` mathematische Funktionen wie z.B. Wurzel, Exponenten oder Sinus vorkommen, also die Bibliothek der mathematischen Funktionen verwendet werden soll, muß obiger Befehl ergänzt werden.

```
$ gcc beispiel.c -lm
```

Dem Linker wird so die Möglichkeit gegeben auf die mathematische Bibliothek zuzugreifen. Einige andere Bibliotheken, wie diejenige zu `stdio.h`, werden vom Compiler automatisch gefunden.

Möchte man statt des Namens `a.out`, einen anderen Namen für das ausführbare Programm erhalten, so ist die Option `-o` zu verwenden. Durch

```
$ gcc beispiel.c -o beisp
```

wird ein ausführbares Programm mit dem Namen `beisp` erzeugt.