1. Das erste Programm

1. 1. Analyse

Bevor Du mit dem Programmieren beginnst, solltest Du erst einmal verstanden haben, was das Programm überhaupt tun soll. Hat man den Sinn eines Programmes nicht verstanden, dann kann man es auch nicht programmieren.

Folgende Fragen solltest Du Dir bei der Analyse beantworten:

- **Welche Aufgabe hat das Programm zu erfüllen ?**
- **Welche mathematischen Formeln liegen der Aufgabe zugrunde ?**
- **Welche Variablen werden benötigt ?**
- **Welchen Datentyp sollen die Variablen haben ?**
- **Welche Variablen müssen eingegeben werden ?**
- **Welche Variablen müssen ausgegeben werden ?**

1. 2. Aufbau eines Programms

Der Aufbau eines Programms folgt in den meisten Fällen diesem Muster:

```
vordefinierte Bibliotheken
void main()
{
   Variablendefinitionen
   Eingabe
   Verarbeitung der Variablen
   Ausgabe
}
```

Dabei können die Teile `Eingabe`, `Verarbeitung der Variablen` und `Ausgabe` fließend ineinander übergehen. Allerdings solltest Du Dich zum Anfang auf diesen Programmaufbau beschränken. Es erleichtert das Verständnis. Zu beachten ist, daß nach jedem Befehl in den vier Teilen ein Semikolon stehen muß.

Die einzelnen Teile des Programmaufbaus werden auf den folgenden Seiten noch näher erläutert.

Man hat die Möglichkeit, Kommentare in den Programmtext einfügen. Diese werden beim Übersetzen ignoriert und dienen der Erläuterung des Programm - Codes. So werden Kommentare gekennzeichnet:

`/* ... */`	Kennzeichnet den Text zwischen den Sternen als Kommentar.
`//`	Alle Zeichen bis zum Ende der Zeile werden als Kommentar gekennzeichnet.

Beispiel:

```
x = 13;

//x = 14; diese Zeile wird vom Compiler nicht beachtet

y = x; /*Dieser Text wird vom Compiler nicht beachtet, der Befehl davor allerdings schon.*/
```

1. 3. Vordefinierte Bibliotheken

Eine vordefinierte Bibliothek enthält C - Code, der geschrieben wurde, um bestimmte Funktionen bereitzustellen. Diesen C - Code brauchst Du dann nicht mehr zu schreiben.

Hier zwei wichtige Bibliotheken:

`stdio.h` - Diese Bibliothek enthält Funktionen für die Ein- und Ausgabe. Wenn Du ein Programm schreibst, so mußt Du diese Bibliothek in fast allen Fällen einbinden.

`math.h` - Diese Bibliothek enthält mathematische Funktionen, wie z. B. die Sinusfunktion und die Quadratwurzel. Wenn Du allerdings nur in den Grundrechenarten programmierst (also Multiplikation, Addition, usw.), brauchst Du diese Bibliothek nicht einzubinden.

Eine vordefinierte Bibliothek wird **immer** am Anfang eines Programms nach dem folgenden Muster eingebunden:

```
#include <bibliotheksname>
```

Dabei muß für jede Bibliothek eine solche Zeile benutzt werden. Hast Du einmal vergessen, eine Bibliothek einzubinden, so kann es passieren, daß Du das Programm nicht übersetzen kannst, da der Compiler die Funktionen nicht kennt.

Beispiel:

```
#include <stdio.h>

#include <math.h>
```

1. 4. Hauptfunktion

Ein C - Programm benötigt **immer** eine Hauptfunktion. Dort wird das ganze Programm mit seinen Variablen, der Eingabe und der Ausgabe hineingeschrieben.

In den meisten Fällen wird die Hauptfunktion so definiert:

```
void main()
{
}
```

Zwischen den beiden geschweiften Klammern wird der C - Code des Programms geschrieben.

Tip: Damit Du nicht mit den geschweiften Klammern durcheinander kommst, ist es gut, wenn Du die Klammern paarweise eingibst. Denn zu jeder geöffneten Klammer gehört auch eine geschlossene. Hast Du irgendwo eine Klammer vergessen, so kannst Du das Programm nicht übersetzen.

1. 5. Variablendefinitionen

Um Variablen definieren zu können, muß man sich vorher überlegen, von welchem Datentyp diese Variablen sein sollen. In diesem Lehrgang werden drei Datentypen vorkommen:

`int`	Integer	ganze Zahlen (-2, -1, 0, 1, 2, 3, ...)
`float`		gebrochene Zahlen (-2.3, -1.21, 0.33, ...)
`char`	Character	Zeichen ('a', 'b', 'j', 'n')

Variablen werden nach dem folgenden Muster definiert:

```
datentyp variablenname_1, ..., variablenname_n;
```

Es ist zu beachten, daß nur Variablen mit dem gleichen Datentyp in einer Zeile definiert werden können.

Beispiel:

int zeile, spalte;

float laufvariable;

Ein Variablenname unterliegt einigen Regeln. Hier ein paar wichtige:

- **Variablennamen dürfen nicht mit einer Zahl beginnen. Innerhalb des Namens dürfen Zahlen vorkommen.**

- **Zeichen, wie Fragezeichen, Komma und Semikolon dürfen nicht im Namen vorkommen.**
- **Variablennamen, die genauso lauten wie reservierte Wörter (z. B. if, else und do) sind nicht erlaubt.**
- **Umlaute, wie ä und ü sind verboten.**
- **Zwischen Groß- und Kleinschreibung wird unterschieden.**

Wenn Du Variablennamen vergibst, so solltest Du darauf achten, daß der Name selbst erklärend ist. Es sollten nicht nur einzelne Buchstaben vergeben werden, denn dann findest Du Dich bald in Deinem eigenen C - Code nicht mehr zurecht.

1. 6. Eingabe

Für die Eingabe werden zwei Funktionen benötigt, die in der Bibliothek `stdio.h` schon definiert worden sind.

`printf()` - Diese Funktion wird benötigt, um dem Benutzer mitzuteilen, welchen Wert er jetzt eigentlich einzugeben hat. Du mußt immer davon ausgehen, daß derjenige, der vor dem Computer sitzt, keine Ahnung vom Ablauf des Programms hat. Stell Dir einfach vor, was Du gerne haben würdest, wenn Du derjenige bist.

So wird mit der Funktion `printf()` gearbeitet:

```
printf("ausgabetext");
```

Nach der Funktion `printf()` folgt die Funktion `scanf()`. Damit wird der Wert eingelesen, der vom Benutzer eingegeben wurde.

So sieht der Aufruf der Funktion aus:

```
scanf("formatstring", &variablenname);
```

Ein Formatstring teilt dem Computer mit, von welchem Datentyp der eingelesene Wert ist. Für die drei Datentypen gibt es unterschiedliche Formatstrings:

%i	Werte vom Typ Integer
%f	Werte vom Typ Float
%c	Werte vom Typ Character

Nach dem Komma, steht der Name der Variablen, die den eingelesenen Wert zugewiesen bekommt. Das & ist der sogenannte Adreßoperator. Das bedeutet, daß der eingelesene Wert an die Adresse der Variablen geht.

Wichtig: Der Datentyp der Variable und der Formatstring müssen zueinander passen ! Sonst kann es zu Fehlern kommen !

Beispiel:

```
int x;

printf("Bitte geben Sie eine ganze Zahl ein: ");

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

1. 7. Verarbeitung der Variablen

Bei der Verarbeitung ist die Planung sehr wichtig. Bei den Grundrechenarten gibt es fünf Operatoren:

+	Addition
-	Subtraktion
*	Multiplikation
/	Division (**Achtung - bei Division von ganzen Zahlen ist 5 / 2 nicht 2.5 sondern 2**)
%	Modulo - Operator (Rest aus einer ganzzahligen Division - Bsp.: 5 % 2 = 1)

Folgende Regeln gelten bei den Grundrechenarten:

- **Es gilt Punktrechnung vor Strichrechnung.**
- **Die Variable für das Ergebnis steht immer links vom Gleichheitszeichen.**
- **Die Klammern müssen immer richtig gesetzt werden, weil es sonst zu falschen Ergebnissen kommen kann.**

Achtung ! Wenn Du einer Variable vom Datentyp Integer den Wert einer Variablen vom Datentyp Float zuweisen möchtest, so gehen Dir die Stellen nach dem Komma verloren. Dies kann eventuell zu Fehlern in Deinem Programm führen. Einige Compiler weisen Dich auf diesen Sachverhalt mit einer Warnung hin.

Beispiel:

x = 2;

y = x + 3; /*Ergebnis ist 5*/

z = 5 + 3 * x; /*Ergebnis ist 11*/

y = (5 + 3) * x; /*Ergebnis ist 16*/

1. 8. Ausgabe

Bei der Ausgabe wird wieder die Funktion `printf()`benötigt. Für die Werte der Variablen, die ausgegeben werden, benötigt man wieder die Formatstrings. Immer da, wo der Wert einer Variablen hin soll, wird ein Formatstring eingesetzt.

```
printf("ausgabetext %formatstring1 ", variablenname);
```

Hast Du mehrere Variablenwerte in einer Anweisung auszugeben, so werden die Variablennamen durch ein Komma getrennt.

Bei den Formatstrings kannst Du außerdem festlegen, wie die Werte ausgegeben werden sollen:

`%2i`	Ganze Zahl mit 2 Stellen
`%2f`	Gebrochene Zahl mit 2 Stellen
`%6.2f`	Gebrochene Zahl mit insgesamt 6 Stellen, davon 2 hinter dem Komma
`%.2f`	Gebrochene Zahl mit 2 Stellen nach dem Komma

Folgende Dinge kannst Du noch in eine `printf()` - Anweisung einbauen:

`\n`	Nach dieser Zeichenkette wird eine neue Zeile eingefügt.
`\t`	Nach dieser Zeichenkette wird ein Tabulator (festgelegte Anzahl von Leerzeichen) eingefügt.

Das sieht bei verschiedenen Aufgaben (z. B. Ausgabe einer Tabelle) sehr gut aus.

Beispiel:

printf("Der Wert von x beträgt %i", x);

/*Für x = 5 sieht dies auf dem Bildschirm dann so aus: "Der Wert von x beträgt 5"*/

printf("Das Haus hat eine Höhe von %4.2f m.", hoehe);

/*Für hoehe = 2.202 sieht dies auf dem Bildschirm dann so aus: "Das Haus hat eine Höhe von 2.20 m."*/

1. 9. Aufgaben

1. Welche, der folgenden Variablennamen sind nicht zulässig ?

```
mehrwertsteuer
3rad
auto?
vierrad
int
länge
```

Lösung:

lsg_1_1.txt **227 Byte** **22. 07. 2001**

2. Schreib ein Programm, welches die Länge, Breite und Höhe eines Quaders einliest und das Volumen ausgibt !

Beispielausgabe:

Eingabe der Laenge [cm]: 2

Eingabe der Breite [cm]: 5

Eingabe der Hoehe [cm]: 4

Das Volumen des Quaders betraegt 40.00 ccm.

Lösungsvorschlag:

lsg_1_2.c **577 Byte** **22. 07. 2001**

3. Schreib ein Programm, welches die Anzahl der gefahrenen Kilometer, die Menge des verbrauchten Benzins und den Preis je Liter Benzin einliest ! Ausgegeben werden soll die Menge des verbrauchten Benzins je 100 km und die Kosten je 100 km.

Beispielausgabe:

Eingabe der gefahrenen Kilometer: 450.3

Eingabe der Menge des verbrauchten Benzins [l]: 30.34

Eingabe des Preises pro Liter Benzin [DM]: 2.02

Sie haben auf 100 km 6.7 l Benzin verbraucht. Dies kostete Sie bei einem Preis von 2.02 DM pro Liter auf 100 km 13.61 DM.

Lösungsvorschlag:

lsg_1_3.c	**780 Byte**	**22. 07. 2001**

2. Verzweigungen

2. 1. Bedingungen

2. 1. 1. Aufbau

Damit wir in Programmen auch unterschiedliche Wege einschlagen können, benötigen wir Verzweigungen. Diese Verzweigungen sind wiederum abhängig von Bedingungen, welche überprüft werden müssen. Bedingungen sind Anweisungen, bei denen zwei Werte verglichen werden. Diese Werte sollten vom gleichen Datentyp sein. Eine Bedingung kann erfüllt werden, oder nicht.

Eine Bedingung ist folgendermaßen aufgebaut:

```
(wert_1 vergleichsoperator wert_2)
```

Die beiden Werte *wert_1* und *wert_2* können sowohl Zahlen und Buchstaben, als auch Variablen sein.

Es gibt sechs wichtige Vergleichsoperatoren:

==	ist gleich
>	ist größer
<	ist kleiner
>=	ist größer gleich
<=	ist kleiner gleich
!=	ist ungleich

Beispiel:

(x < 5) /*Diese Bedingung ist erfüllt, wenn der Wert von x kleiner als 5 ist*/

(x == y) /*Diese Bedingung wird erfüllt, wenn die Werte von x und y übereinstimmen*/

(x != y) /*Diese Bedingung wird erfüllt, wenn die Werte von x und y nicht übereinstimmen*/

(x == 'j') /*Diese Bedingung wird erfüllt, wenn der Wert von x dem Zeichen 'j' entspricht*/

2. 1. 2. Verknüpfungen von Bedingungen

Es kann vorkommen, daß mehrere Bedingungen gleichzeitig erfüllt sein müssen, um einen Befehl auszuführen. Dazu müssen die Bedingungen verknüpft werden.

```
((Bedingung_1) verknüpfungsoperator (Bedingung_2))
```

Es gibt zwei wichtige Verknüpfungsoperatoren:

&&	und
||	oder

Eine verknüpfte Bedingung kann demnach so aussehen:

```
((Bedingung_1) && (Bedingung_2))
```

Zum Lesen:

Wenn *Bedingung_1* **und** *Bedingung_2* **erfüllt sind, dann ist die gesamte Bedingung erfüllt.**

Beispiel:

((x == y) && (x < 5)) /*Diese Bedingung ist erfüllt, wenn die Werte x und y übereinstimmen und darüber hinaus der Wert von x kleiner als 5 ist.*/

(x != y) || (x == 6)) /*Diese Bedingung ist erfüllt, wenn die Werte x und y nicht übereinstimmen oder wenn der Wert von x gleich 6 ist.*/

((x == y) || (x != y)) /*Achtung ! Diese Bedingung ist sinnlos, da sie immer erfüllt sein wird.*/

((x == y) && (x != y)) /*Achtung ! Diese Bedingung wird niemals erfüllt sein.*/

2. 2. if und else

2. 2. 1. Die if – Anweisung

Eine if - Anweisung ist dazu da, um Entscheidungen zu treffen. Je nach Bedingung werden dann die Befehle ausgeführt, oder auch nicht.

Eine if - Anweisung ist wie folgt aufgebaut:

```
if (Bedingung)
   Befehl;
```

Zum Lesen:

Falls die *Bedingung* erfüllt ist, dann führe den *Befehl* aus.

Es ist auch möglich, daß bei einer Bedingung mehrere Befehle ausgeführt werden müssen. Dann müssen diese in geschweifte Klammern gesetzt werden.

```
if (Bedingung)
{
   Befehl_1;
   ...
   Befehl_n;
}
```

Zum Lesen:

Falls die *Bedingung* erfüllt ist, dann führe die *Befehle* aus.

Beispiel:

```
if ((x % 2) == 0)
    printf("%i", x);
/*Diese Anweisung würde den Wert von x nur ausgeben, wenn er eine gerade Zahl ist.*/
if ((x == 3) || (x == 5))
{
    y = x;
    printf("%i", y);
}
/*Diese Anweisungen werden nur ausgeführt, wenn der Wert von x gleich 3 oder 5 ist.*/
```

2. 2. 2. Die else – Anweisung

Will man aber für den Fall, daß die Bedingung nicht erfüllt ist, auch Befehle ausführen, so benötigt man die else - Anweisung.

```
if (Bedingung)
   Befehl_1;
else
   Befehl_2;
```

Zum Lesen:

Falls die *Bedingung* **erfüllt ist, dann führe** *Befehl_1* **aus.**
Ansonsten führe *Befehl_2* **aus.**

Zu beachten ist wiederum, daß mehrere Befehle in geschweifte Klammern gesetzt werden müssen.

Beispiel:

```
if (zahl >= 0)
   printf("Die Zahl %i ist eine positive Zahl.", zahl);
else
   printf("Die Zahl %i ist eine negative Zahl.", zahl);
/*Diese Anweisung aus if und else, gibt aus, ob die Zahl positiv oder negativ ist.*/
```

2. 2. 3. Kombination aus if - und else – Anweisungen

Sofern man mehrere Bedingungen zu untersuchen hat, empfiehlt es sich, eine Kombination aus if - und else - Anweisungen zu benutzen.

```
if (Bedingung_1)
   Befehl_1;
else if (Bedingung_2)
   Befehl_2;
else
   Befehl_3;
```

Zum Lesen:

Falls *Bedingung_1* **erfüllt ist, führe** *Befehl_1* **aus.**
Ansonsten, falls *Bedingung_2* **erfüllt ist, führe** *Befehl_2* **aus.**
Ansonsten führe *Befehl_3* **aus.**

Beispiel:

```
if (geburtstag < jahrestag)

    printf("Sie hatten bereits Geburtstag.");

else if (geburtstag > jahrestag)

    printf("Sie haben in diesem Jahr noch Geburtstag.");

else

    printf("Herzlichen Glückwunsch zum Geburtstag.");

/*Der Wert von geburtstag wird berechnet, indem der Tag im Jahr genommen wird.
Genau so wird auch der Wert von jahrestag, dem aktuellen Datum, berechnet. Ist
der geburtstag kleiner als der jahrestag, so hatte die Person schon Geburtstag. Hatte
sie noch nicht, so ist folglich, der Geburtstag größer als der Jahrestag. Treffen beide
Bedingungen nicht zu, so gibt es den entsprechenden Glückwunsch.*/
```

2. 3. switch, case und brake

2. 3. 1. Die switch - case – Anweisung

Wenn eine Variable mehrere verschiedene Werte annehmen kann und man für jeden einzelnen Wert eine eigene Verarbeitung programmieren muß, können if - else - Anweisungen leicht unübersichtlich werden.

Um dem Abhilfe zu verschaffen, verwendet man die switch - case - Anweisung.

```
switch (variable)
{
   case wert_1 : anweisung_1_1; ... anweisung_1_m;
   break;

   case wert_2 : anweisung_2_1; ... anweisung_2_m;
   break;

   ...
```

```
    case wert_n : anweisung_n_1; ... anweisung_n_m;
}
```

Zum Lesen:

Für den Fall, daß die *variable* **den** *wert_1* **hat, führe die Anweisungen** *anweisung_1_1* **bis** *anweisung_1_m* **aus.**
Für den Fall, daß die *variable* **den** *wert_2* **hat, führe die Anweisungen** *anweisung_2_1* **bis** *anweisung_2_m* **aus. ...**

Zu beachten ist, daß man die Anweisungen, im Gegensatz zur if - else - Anweisung, nicht in die geschweiften Klammern setzen muß.

Achtung ! switch - case - Anweisungen eignen sich nicht für Aufgaben, bei denen Wertebereiche überprüft werden müssen. Sie eignen sich nur für Aufgaben, bei denen jeder Wert eine eigene Verarbeitung benötigt.

Beispiel:

```
if (auto == 1)
    printf("Sie fahren einen Mercedes.");
else if (auto == 2)
    printf("Sie fahren einen BMW.");
else if (auto == 3)
    printf("Sie fahren einen Jaguar.");
else if (auto == 4)
    printf("Sie fahren einen Trabant.");
else if (auto == 5)
    printf("Sie fahren einen Audi TT.");

/*Dies sieht noch einfach aus. Versuch das aber mal mit allen Automarken, die es gibt. Eleganter ist die folgende Variante.*/
```

```
    switch (auto)
    {
        case 1: printf("Sie fahren einen Mercedes."); break;
        case 2: printf("Sie fahren einen BMW."); break;
        case 3: printf("Sie fahren einen Jaguar."); break;
        case 4: printf("Sie fahren einen Trabant."); break;
        case 5: printf("Sie fahren einen Audi TT.");
    }
```

/*Das folgende Beispiel ist ein Fall, für den eine switch - case - Anweisung sich nicht eignet. Zuerst die if - else - Anweisung:*/

```
if (x < 5)
    printf("Die Zahl x ist kleiner als 5.");
else
    printf("Die Zahl x ist groesser als 4.");
```

/*Setzen wir dann voraus, daß x nur Werte von 1 bis 5 annehmen kann, so müßte eine switch - case - Anweisung dann so aussehen:*/

```
switch (x)
{
    case 1: printf("Die Zahl x ist kleiner als 5."); break;
    case 2: printf("Die Zahl x ist kleiner als 5."); break;
    case 3: printf("Die Zahl x ist kleiner als 5."); break;
    case 4: printf("Die Zahl x ist kleiner als 5."); break;
    case 5: printf("Die Zahl x ist groesser als 4.");
}
```

/*Man sieht, daß dies sehr umständlich ist. Könnte x auch negative Zahlen und Zahlen, die größer sind als 5, annehmen, so würde das Programmieren mit einer switch - case - Anweisung unmöglich werden.*/

2. 3. 2. Die break – Anweisung

Hatte die *variable* den *wert_1* und hat der Computer die Anweisungen ausgeführt, so würde er die Anweisungen, die danach kommen, auch noch ausführen. Deswegen wird nach den Anweisungen ein *break* eingeführt. Dies bedeutet, daß die switch - case - Anweisung abgebrochen wird. Es wird nicht mehr weiter verglichen und das Programm wird fortgesetzt.

2. 3. 3. Die default – Anweisung

Wir hatten bei der if - else - Anweisung schon festgestellt, daß man beim letzten *else* keine Bedingung einfügen muß. Hierfür gibt es eine vergleichbare Anweisung, die default - Anweisung.

```
switch (variable)
{
   case wert_1 : anweisung_1_1; ... anweisung_1_m;
   break;

   case wert_2 : anweisung_2_1; ... anweisung_2_m;
   break;

   ...

   case wert_n : anweisung_n_1; ... anweisung_n_m;
   break;

   default : anweisung_1; ... anweisung_n;
}
```

Zum Lesen:

Für den Fall, daß die *variable* den *wert_1* hat, führe die Anweisungen *anweisung_1_1* bis *anweisung_1_m* aus.
Für den Fall, daß die *variable* den *wert_2* hat, führe die Anweisungen *anweisung_2_1* bis *anweisung_2_m* aus. ...
Standardmäßig führe die Anweisungen *anweisung_1* bis *anweisung_n* aus.

Zu beachten ist, daß man nach den Anweisungen hinter *default* kein *break* setzt. Dies wäre sinnlos, da ja sowieso die Vergleiche damit abgeschlossen sind.

Beispiel:

```
/*Dieses Beispiel baut auf dem Beispiel aus Kapitel 2. 3. 1. auf*/
switch (auto)
{
    case 1: printf("Sie fahren einen Mercedes."); break;
    ...
    default: printf("Sie haben keine gültige Automarke eingegeben.");
}
```

2. 4. Aufgaben

1. Schreib ein Programm, welches bei einer eingegebenen Anzahl von Punkten, die entsprechende Note ausgibt. Dabei gelten folgende Regeln:

- **Mehr als 30 Punkte ergeben die Note 1**
- **21 bis 30 Punkte ergeben die Note 2**
- **11 bis 20 Punkte ergeben die Note 3**
- **Weniger als 11 Punkte ergeben die Note 4**

Beispielausgabe:

```
Eingabe der Punktzahl: 25
Fuer 25 Punkte gibt es eine 2.
```

Lösungsvorschlag:

lsg_2_1.c **984 Byte** **25. 07. 2001**

2. Schreib ein Programm, welches ein Menü mit 5 Punkten anzeigt ! (Erzeugen, Ausdrucken, Anfuegen, Loeschen, Aufhoeren) Bei Eingabe einer Zahl soll der gewählte Menüpunkt ausgegeben werden. Das Programm soll mit if - else - Anweisungen programmiert werden.

Beispielausgabe:

1: Erzeugen

2: Ausdrucken

3: Anfuegen

4: Loeschen

5: Aufhoeren

Bitte waehlen Sie einen Menuepunkt: 3

Sie haben Anfuegen gewaehlt.

Lösungsvorschlag:

lsg_2_2.c **1121 Byte** **25. 07. 2001**

3. Schreib das Menü - Programm mit Hilfe einer switch - case - Anweisung !

Lösungsvorschlag:

lsg_2_3.c **1106 Byte** **25. 07. 2001**

3. Schleifen

3. 1. Allgemeines

Schleifen werden in der Programmierung benutzt, um Programmiercode, welcher mehrfach hintereinander ausgeführt werden muß, einzusparen.

In der C - Programmierung existieren 3 Arten von Schleifen:

- **while - Schleife (kopfgesteuerte Schleife)**
- **do - while - Schleife (fußgesteuerte Schleife)**
- **for - Schleife (kopfgesteuerte Schleife)**

3. 2. Die while – Schleife

While - Schleifen werden in der Regel für Programmteile benutzt, bei denen eine Bedingung überprüft werden muß, **bevor** der Programmcode innerhalb der Schleife ausgeführt wird. Dabei kann es passieren, daß dieser gar nicht ausgeführt wird. Daher eignen sich while - Schleifen für Programmteile, welche mindestens einmal ausgeführt werden müssen (z. B. Variableneingabe am Anfang eines Programms), weniger.

Eine while - Schleife ist folgendermaßen aufgebaut:

```
while (Bedingung)
{
   Befehl_1;
   ...
   Befehl_n;
}
```

Zum Lesen:

Solange die *Bedingung* erfüllt ist, führe die *Befehle* aus.

Bevor die Befehle in der Schleife ausgeführt werden, wird die Bedingung überprüft. Nur wenn diese erfüllt ist, tritt das Programm in die Schleife ein und die Befehle werden ausgeführt. Da die Bedingung zuerst überprüft wird, wird die while - Schleife auch **kopfgesteuerte Schleife** genannt.

Wie eine Bedingung aufgebaut ist, wurde im **Kapitel 2. 1.** behandelt.

Wenn Du in der Schleife nur einen Befehl ausführen möchtest, so benötigst Du die geschweiften Klammern nicht.

Achtung !

Die Bedingung darf nicht immer erfüllt sein. Ansonsten kann die Schleife und damit auch das Programm nicht beendet werden.

Beispiel:

```
int x = 0;
while (x < 5)
{
    printf("x = %i", x);
    x = x + 1;
}

/*Die Befehle in dieser Schleife werden nur ausgeführt, wenn die Bedingung erfüllt
ist, also wenn x kleiner als 5 ist. Die Variable wird hier gleich mit dem Wert 0
initialisiert, da sie kurz darauf in der Bedingung verglichen wird.*/

int x = 0;
while (x < 5)
{
    printf("x = %i", x);
    x = x - 1;
}

/*Achtung ! Dieser Programmtext ist gefährlich. Er sieht auf den ersten Blick
genau so aus, wie der vorherige. Allerdings wird hier x immer kleiner. Die
Bedingung wird also immer erfüllt sein. Die Schleife wird niemals enden und das
Programm kann sich nicht selbst beenden.*/
```

3. 3. Die do - while – Schleife

Do - while - Schleifen werden in der Regel für Programmteile benutzt, bei denen die Bedingung überprüft wird, **nachdem** der Programmcode innerhalb der Schleife ausgeführt wurde. Der Programmcode wird also mindestens einmal ausgeführt. Sie werden daher häufig in der Variableneingabe am Anfang eines Programms eingesetzt. Ist allerdings die Überprüfung der Bedingung vor der Ausführung des Programmcodes erforderlich, so eignet sich eine do - while - Schleife nicht.

Eine do - while - Schleife ist folgendermaßen aufgebaut:

```
do
{
   Befehl_1;
   ...
   Befehl_n;
}
while (Bedingung);
```

Zum Lesen:

Führe die *Befehle* **aus, solange die** *Bedingung* **erfüllt ist.**

Bei der do - while - Schleife ist zu beachten, daß zuerst die Befehle einmal ausgeführt werden und dann überprüft wird, ob die Bedingung erfüllt ist. Eine do - while - Schleife wird daher auch **fußgesteuert** genannt.

Auch hier kann man die geschweiften Klammern einsparen, wenn man in der Schleife nur einen Befehl ausführt. Und es ist wiederum darauf zu achten, daß die Bedingung nicht immer erfüllt sein darf.

Beispiel:

```
int wert;
int anzahl = 0;
do
{
   printf("Bitte Wert eingeben [0 bricht ab]: ");
   scanf("%i", &wert);
```

```
    anzahl = anzahl + 1;
}
while (wert != 0);
printf("\n\nSie haben %i Zahlen eingegeben.", anzahl - 1);

/*Dieser Programmteil demonstriert eine solche sich wiederholende Eingabe. Erst
nachdem der Programmcode einmal durchlaufen wurde, wird überprüft, ob die
Bedingung erfüllt ist. Nach dem Ende der Schleife wird die Anzahl der
eingegebenen Zahlen ausgegeben. Dabei wird anzahl - 1 ausgegeben, da die zuletzt
eingegebene 0 nicht mit gerechnet werden soll. Die Variable anzahl wird hier mit 0
initialisiert, da in der Schleife der Wert von anzahl um 1 erhöht wird.*/
```

3. 4. Die for – Schleife

Während wir bei while - Schleifen und do - while - Schleifen nur eine Bedingung überprüft haben, bevor oder nachdem der Programmcode innerhalb der Schleife ausgeführt wurde, arbeiten wir bei den for - Schleifen mit einer Laufvariable, deren Wert ausschlaggebend ist, für die Ausführung des Programmcodes innerhalb der Schleife.

Eine for - Schleife ist folgendermaßen aufgebaut:

```
for (laufvariable = startwert; Bedingung mit
Laufvariable; Änderung der Laufvariable)
{
   Befehl_1;
   ...
   Befehl_n;
}
```

Zum Lesen:

Solange die *Bedingung* **erfüllt ist, führe die** *Befehle* **aus.**

Wie man sieht, ist das Lesen der for - Schleife genauso, wie das Lesen der while - Schleife. Sie ist also ebenfalls **kopfgesteuert**.

Der Kopf der for - Schleife beinhaltet 3 wichtige Anweisungen. Die erste Anweisung setzt die Laufvariable auf einen Startwert. Sie wird also initialisiert. Die zweite Anweisung stellt die

Bedingung dar, welche erfüllt sein muß, damit der Programmcode ausgeführt wird. Diese Bedingung sollte den Wert der Laufvariable überprüfen. Tust Du das nicht, so ist die Benutzung der for - Schleife sinnlos. Die dritte Anweisung besagt, wie sich der Wert der Laufvariablen nach einem Schleifendurchlauf verändert. Du mußt Dich dort entscheiden, ob der Wert sich erhöht oder ob er kleiner wird. Wie Du eine Wertänderung in der C - Programmierung abkürzen kannst, findest Du auf der nächsten Seite.

Auch hier kann man die geschweiften Klammern einsparen, wenn man in der Schleife nur einen Befehl ausführt.

Beispiel:

```
int x;

for (x = 0; x < 6; x = x + 1)

{

    printf("%x ", x);

}


/*Dies ist eine sehr einfache Schleife. Die Variable x wird zu Beginn der for - Schleife mit 0 initialisiert. Danach wird überprüft, ob die Bedingung erfüllt ist. Ist dies der Fall, so wird der Wert von x ausgegeben. Danach wird der Wert von x um 1 erhöht. Es erfolgt wieder die Überprüfung der Bedingung. Diese Schleife wird genau 6 mal durchlaufen.*/


int anzahl, gesamt, wert;

int summe = 0;

printf("Wieviele Werte wollen Sie eingeben ? ");

scanf("%i", &gesamt);

for (anzahl = 0; anzahl < gesamt; anzahl = anzahl + 1)

{

    printf("Eingabe des %i. Wertes: ", anzahl + 1);

    scanf("%i", &wert);

    summe = summe + wert;

}

printf("Die Summe der eingegebenen Werte ist %i.", summe);

/*Dieses Beispiel zeigt den Einsatz der for - Schleife im Eingabeteil eines Programms. Die Anzahl der einzugebenden Werte wird festgelegt. Danach wird die
```

for - Schleife gestartet. Die Laufvariable anzahl wird mit 0 initialisiert. Die Bedingung besagt, daß anzahl kleiner als die Anzahl der einzugebenden Werte sein muß. Dann wird der erste Wert eingegeben. Die Summe, welche zu Beginn des Programms mit 0 initialisiert wurde, wird um den eingegebenen Wert erhöht. Nach dem Schleifendurchlauf wird die Laufvariable um 1 erhöht. Nach dem Ende der Schleife wird die Summe ausgegeben.*/

3. 5. Arithmetische Abkürzungen in C

In der C - Programmierung hast Du die Möglichkeit Befehle, welche die Werte von Variablen verändern, abzukürzen. Die folgende Tabelle soll das verdeutlichen.

Befehl	**Beschreibung**
`variable++;`	erhöht den Wert der Variable um 1
`variable--;`	erniedrigt den Wert der Variable um 1
`variable += 2;`	erhöht den Wert der Variable um 2
`variable -= 3;`	erniedrigt den Wert der Variable um 3
`variable *= 5;`	multipliziert den Wert der Variablen mit 5
`variable /= 3;`	dividiert den Wert der Variablen durch 3

Jeder dieser Befehle verändert den Wert der Variablen. Du kannst dadurch also auf die längere Schreibweise (Beispiel: variable = variable + 2;) verzichten.

3. 6. Aufgaben

1. Schreib ein Programm, welches eine Anfangszahl und eine Endzahl einliest. Achte darauf, daß die Anfangszahl kleiner als die Endzahl ist ! Ausgegeben werden soll jede dritte Zahl, die sich **zwischen** der Anfangszahl und der Endzahl befindet.

Schreib den Teil der Variablenverarbeitung mit:

- **einer while - Schleife**
- **einer do - while - Schleife**
- **einer for - Schleife**

Beispielausgabe:

Eingabe der Anfangszahl: 1

Eingabe der Endzahl: 15

Zahlen zwischen den beiden Zahlen: 2 5 8 11 14

Lösungsvorschlag:

lsg_3_1_a.c	**1144 Byte**	**26. 08. 2001**
lsg_3_1_b.c	**1190 Byte**	**26. 08. 2001**
lsg_3_1_c.c	**1104 Byte**	**26. 08. 2001**

2. Schreib ein Programm, welches vom Nutzer beliebig viele Zahlen eingeben läßt. Nach Eingabe einer 0, soll die Eingabe beendet werden. Ausgegeben werden sollen die Anzahl der eingegebenen Zahlen, die Summe der eingegebenen Zahlen und der Durchschnitt aller eingegebenen Zahlen.

Achtung ! Die eingegebene 0 soll nicht mit in die Berechnung einfließen !

Beispielausgabe:

Eingabe einer Zahl [0 bricht ab]: 1

Eingabe einer Zahl [0 bricht ab]: 2

Eingabe einer Zahl [0 bricht ab]: 3

Eingabe einer Zahl [0 bricht ab]: 0

Eingegebene Zahlen: 3

Summe: 6.0

Durchschnitt: 2.0

Lösungsvorschlag:

lsg_3_2.c	**1281 Byte**	**26. 08. 2001**

3. Schreib ein Programm, welches vom Nutzer beliebig viele Zahlen eingeben läßt. Nach Eingabe einer negativen Zahl soll die Eingabe beendet werden. Nach jeder eingegebenen Zahl soll neben der eingegebenen Zahl auch der Mittelwert aller eingegebenen Zahlen neu berechnet und ausgegeben werden. Wenn die eingegebene Zahl mehr als 10 % vom aktuellen Mittelwert abweicht soll zusätzlich hinter dem Mittelwert noch ein **!** ausgegeben werden.

Beispielausgabe:

Eingabe einer Zahl [negative Zahl bricht ab]: 3

eingegeben: 3 Mittelwert: 3.0

Eingabe einer Zahl [negative Zahl bricht ab]: 2

eingegeben: 2 Mittelwert: 2.5 !

Eingabe einer Zahl [negative Zahl bricht ab]: 0

eingegeben: 0 Mittelwert: 1.7 !

Eingabe einer Zahl [negative Zahl bricht ab]: -1

Programmende !

Lösungsvorschlag:

lsg_3_3.c **1254 Byte** **26. 08. 2001**

4. Arrays

4. 1. Allgemeines

Arrays (dt. Felder) werden in der C - Programmierung verwendet, um mehrere Werte, die ein und den selben Sachverhalt beschreiben, zusammenzufassen.

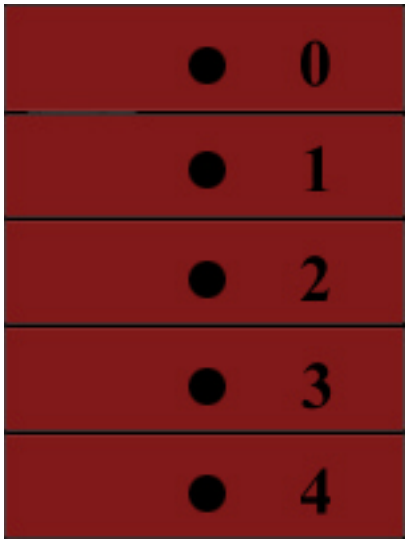
Eine Wetterstation nimmt beispielweise verschiedene Werte auf. Sie misst die Temperatur, den Luftdruck und die Luftfeuchtigkeit. Für jeden dieser Werte würde man eine Variable in einem C - Programm benötigen. Da aber Wetterstationen die Temperatur über einen bestimmten Zeitraum messen müssen, werden für die Temperatur, die Luftfeuchtigkeit und den Druck mehrere Werte benötigt. Dafür sind die Arrays in der C - Programmierung zuständig. Mit Hilfe eines solchen Feldes kann man beispielsweise die Temperatur der Wetterstation im Zeitraum von einer Stunde speichern.

In der C - Programmierung unterscheidet man die Arrays anhand der Dimensionen. Wir behandeln in diesem Lehrgang nur ein- und zweidimensionale Arrays.

4. 2. Eindimensionale Arrays

4. 2. 1. Erläuterung

Ein Array ist ein sehr abstraktes Gebilde. Um es besser zu verdeutlichen, könntest Du Dir beispielsweise eine Komode vorstellen, die eine bestimmte Anzahl von Schubladen hat.



In dieser Komode (im Beispiel 5 Schubfächer) werden die einzelnen Werte untergebracht. Diese Werte sind alle vom gleichen Datentyp.

4. 2. 2. Definieren des eindimensionalen Arrays

Ein eindimensionales Array wird folgendermaßen definiert:

```
datentyp feldname[feldgröße];
```

Beispiel:

int komode[5];

/*Das Array komode (um das Beispiel aufzugreifen) kann insgesamt 5 Werte vom Datentyp integer aufnehmen.*/

4. 2. 3. Zugriff auf die Elemente des eindimensionalen Arrays

Um auf die einzelnen Elemente des Arrays zuzugreifen (bei der Komode werden die Fächer geöffnet) wird folgende Schreibweise verwendet:

```
feldname[feldelement]
```

Folgendes kann man mit einem Feldelement machen:

- dem Feldelement einen Wert zuweisen
- den Wert des Feldelementes auslesen
- den Wert des Feldelementes manipulieren (z. B. Addition oder andere arithmetische Operationen)

Besonders wichtig ist zu wissen, dass bei Arrays nicht begonnen wird mit 1 zu zählen, sondern mit 0. Will man also das 3. Element des Arrays komode ansprechen, dann muss man folgende Schreibweise verwenden:

```
komode[2]
```

Die Numerierung von Feldnamen geht also von 0 bis (*feldgröße* - 1).

Warnung !

Wenn Du fälschlicherweise versuchst, ein Element außerhalb des Feldes anzusprechen, stürzt Dir das Programm ab !

Beispiel:

int komode[5];

komode[1] = 30; //ist zulässig

komode[5] = 10; /*ist nicht zulässig, da das Element außerhalb des Feldes liegen würde*/

komode[3] = 2.1; /*ist nicht zulässig, da der Wert vom falschen Datentyp ist*/

4. 3. Zweidimensionale Arrays

4. 3. 1. Erläuterung

Wenn man eine Wetterstation hat, die verschiedene Werte messen muss, eignet sich ein eindimensionales Array. Messen aber mehrere Wetterstationen gleichzeitig um die Wettervorhersage genauer zu gestalten, so bräuchte man mehrere Felder. Für solche Fälle sind in der C - Programmierung die mehrdimensionalen (in unserem Fall zweidimensionalen) Felder zuständig. Das sind sozusagen Arrays aus Arrays.

Ein zweidimensionales Array kannst Du Dir in Form einer Tabelle vorstellen:

wert[0][0]	wert[1][0]	wert[2][0]
wert[0][1]	wert[1][1]	wert[2][1]
wert[0][2]	wert[1][2]	wert[2][2]
wert[0][3]	wert[1][3]	wert[2][3]
wert[0][4]	wert[1][4]	wert[2][4]

Wir legen hiermit fest, daß in diesem Lehrgang die erste Zahl die Spalte und die zweite Zahl die Zeile darstellen soll.

4. 3. 2. Definieren des zweidimensionalen Arrays

Ein zweidimensionales Array wird folgendermaßen definiert:

```
datentyp feldname[spaltenanzahl][zeilenanzahl];
```

Beispiel:

int tabelle[3][5];

/*Das zweidimensionale Array tabelle hat also 3 Spalten und 5 Zeilen, welche Werte vom Datentyp integer aufnehmen.*/

4. 3. 3. Zugriff auf die Elemente des zweidimensionalen Arrays

Die einzelnen Elemente des Arrays (also der Tabelle) werden folgendermaßen angesprochen:

```
feldname[spalte][zeile]
```

Auch hier kann man wieder dem Element einen Wert zuweisen, den Wert durch Arithmetik manipulieren oder den Wert des Elementes auslesen.

Zu beachten ist wiederum, daß die Elemente von 0 an gezählt werden. Der Wert `spalte` kann also von 0 bis (`spaltenanzahl` - 1) gehen. Analog dazu geht der Wert `zeile` von 0 bis (`zeilenanzahl` - 1).

Beispiel:

int tabelle[3][5];

tabelle[1][3] = 3; //ist zulässig

tabelle[3][3] = 5; /*ist nicht zulässig, da der Spaltenwert außerhalb des Feldes liegt*/

tabelle[2][6] = 10; /*ist nicht zulässig, da der Zeilenwert außerhalb des Feldes liegt*/

4. 4. Arbeiten mit Arrays: for – Schleifen

Wenn Du mit Arrays arbeitest, wirst Du um die Benutzung von for - Schleifen kaum herumkommen.

Bei eindimensionalen Arrays wirst du häufig eine einfache for - Schleife benutzen. Dabei benötigst Du eine Laufvariable, die die Nummer des Feldelementes darstellt. Mußt Du beispielsweise alle Elemente eines Arrays ausgeben, so kannst Du dies wie folgt realisieren:

```
for (element = 0; element < 5; element++)
   printf("%i ", array[element]);
```

Dabei muß natürlich folgendes vorher definiert worden sein:

int array[5];

int element;

Die Elemente des Arrays mußten vorher mit Integer - Werten gefüllt worden sein.

Bei zweidimensionalen Arrays benutzt Du in den meisten Fällen zwei ineinander verschachtelte for - Schleifen. Du benötigst dabei zwei Laufvariablen, eine für die Spalte und eine für die Zeile. Um beispielsweise eine Tabelle auszugeben, die in einem zweidimensionalen Array gespeichert ist, kannst Du folgende Schreibweise verwenden:

```
for (zeile = 0; zeile < 5; zeile++)
{
   for (spalte = 0; spalte < 3; spalte++)
      printf("%i ", tabelle[spalte][zeile]);
   printf("\n");
}
```

Hierbei muß wieder vorher definiert worden sein:

- **int tabelle[3][5]**
- **int zeile**
- **int spalte**

Auch hier mußten die Elemente des Arrays mit Integer - Werten gefüllt worden sein.

Welche Art von Array bzw. welche Art von Operationen Du benutzt hängt natürlich ganz von der Aufgabenstellung ab. Das Programmierungselement for - Schleife ist jedoch das am häufigsten benutzte bei Arrays.

Beispiel:

int luftdruck[3600];

int sekunde;

for (sekunde = 0; sekunde < 3600; sekunde++)

scanf("%i", &luftdruck[sekunde]);

/*Diese Anweisung liest für jede Sekunde einer Stunde (60 [Sekunden] * 60 [Minuten]) den Luftruck ein. Pro Sekunde wird einem Feldelement ein Wert zugewiesen.

```
Für 10 Wetterstationen würde der Teil des Programms folgendermaßen aussehen*/
int luftdruck[10][3600];

int sekunde, wetterstation;

for (wetterstation = 0; wetterstation < 10; wetterstation++)

for (sekunde = 0; sekunde < 3600; sekunde++)

scanf("%i", &luftdruck[wetterstation][sekunde]);

/*Pro Wetterstation werden 3600 Werte (für jede Sekunde einer) eingelesen.*/
```

4. 5. Aufgaben

1. Schreib ein Programm, welches 10 ganze Zahlen einliest. Diese Zahlen sollen dann vom Programm rückwärts ausgegeben werden.

Beispielausgabe:

```
Eingabe der 1. Zahl: 1

Eingabe der 2. Zahl: 3

Eingabe der 3. Zahl: 5

Eingabe der 4. Zahl: 6

Eingabe der 5. Zahl: 7

Eingabe der 6. Zahl: 8

Eingabe der 7. Zahl: 9

Eingabe der 8. Zahl: 10

Eingabe der 9. Zahl: 12

Eingabe der 10. Zahl: 14


Ausgabe in umgekehrter Reihenfolge: 14, 12, 10, 9, 8, 7, 6, 5, 3, 1
```

Lösungsvorschlag:

lsg_4_1.c **1208 Byte** **17. 03. 2002**

2. Schreib ein Programm, welches 10 ganze Zahlen einliest. Diese Zahlen sollen dann vom Programm der Größe nach sortiert und ausgegeben werden. Beginne mit der kleinsten Zahl.

Hinweis: Du benötigst einige Hilfsvariablen.

Beispielausgabe:

Eingabe der 1. Zahl: 5

Eingabe der 2. Zahl: 9

Eingabe der 3. Zahl: 17

Eingabe der 4. Zahl: 3

Eingabe der 5. Zahl: 77

Eingabe der 6. Zahl: 11

Eingabe der 7. Zahl: 8

Eingabe der 8. Zahl: 2

Eingabe der 9. Zahl: 13

Eingabe der 10. Zahl: 6

Ausgabe in sortierter Reihenfolge: 2, 3, 5, 6, 8, 9, 11, 13, 17, 77

Lösungsvorschlag:

lsg_4_2.c **1806 Byte** **17. 03. 2002**

3. Schreib ein Programm, welches die Noten von 3 Schülern in 7 Fächern einliest und diese als Tabelle ausgibt. Dabei soll zu jedem Schüler die Durchschnittsnote am Ende der Zeile ausgegeben werden.

Achte auf einen guten Tabellenkopf !

Beispielausgabe nach erfolgter Eingabe:

Schueler	*N 1*	*N 2*	*N 3*	*N 4*	*N 5*	*N 6*	*N 7*	*D*
S 1	*1*	*3*	*1*	*2*	*1*	*4*	*2*	*2.00*
S 2	*6*	*4*	*5*	*2*	*1*	*1*	*2*	*3.00*
S 3	*4*	*5*	*3*	*3*	*3*	*2*	*1*	*3.00*

Lösungsvorschlag:

lsg_4_3.c	**1716 Byte**	**17. 03. 2002**

5. Funktionen

5. 1. Allgemeines

Funktionen werden in der C - Programmierung verwendet, um Programmtext, der sich mehrfach wiederholt, einzusparen.

Hat man beispielsweise die Aufgabe, die Werte einer Meßstation auszugeben und muß man das noch einmal machen, nachdem die Werte sich geändert haben, so ist hier eine Funktion angebracht.

Funktionen müßtest Du schon aus der Mathematik kennen. Eine der bekanntesten Funktionen ist wohl `y = f (x) = mx + n`, welche eine Gerade beschreibt. Im Verlauf dieses Kapitels werde ich diese Funktion zur Hilfe nehmen, um den Sachverhalt zu verdeutlichen.

5. 2. Aufbau

Eine Funktion der C - Programmierung hat den folgenden Aufbau:

```
rückgabedatentyp funktionsname (datentyp1 parameter1,
..., datentyp_n parameter_n)
{
   Funktionsrumpf
}
```

Eine Funktion wird vor der Funktion `main()` definiert.

5. 3. Funktionsname

Der Funktionsname ist wichtig für die Identifizierung einer Funktion. Die Benennung einer Funktion unterliegt wieder mehreren Regeln:

- **Funktionsnamen dürfen nicht mit einer Zahl beginnen. Innerhalb des Namens dürfen Zahlen vorkommen.**
- **Zeichen, wie Fragezeichen, Komma und Semikolon dürfen nicht im Namen vorkommen.**
- **Funktionsnamen, die genauso lauten wie reservierte Wörter (z. B. if, else und do) sind nicht erlaubt.**
- **Umlaute, wie ä und ü sind verboten.**
- **Zwischen Groß- und Kleinschreibung wird unterschieden.**

Die Regeln sind genau die gleichen, wie bei der Benennung von Variablen.

Bei unserer Beispielfunktion `f(x)` ist der Funktionsname also `f`.

5. 4. Parameter

Eine Funktion arbeitet mit zwei verschiedenen Arten von Variablen. Variablen, die sie von außen übergeben bekommt und Variablen, die in ihr definiert sind.

Die inneren Variablen werden wie Variablen in einem normalen Programm definiert, also mit Datentyp und Variablenname.

Die Variablen, die an die Funktion übergeben werden, werden **`Parameter`** genannt. Sie werden bei der Definition der Funktion mit Datentyp und Parametername angegeben. Im Funktionsrumpf kann man dann die Parameter wie eine normale Variable behandeln.

Warnung !

Es ist nicht möglich, die Parameter nach Datentyp zu trennen, wie es bei der normalen Variablendefinition möglich ist ! Jeder Parameter muß extra mit Datentyp definiert werden !

Falls eine Funktion keine Parameter benötigt, so wird nach dem Funktionsnamen nur eine leere Klammer gesetzt.

Bei unserer Beispielfunktion haben wir nur einen Parameter. Er hat den Namen *`x`*. Der Datentyp dieses Parameters könnte beispielsweise *`float`* sein.

5. 5. Funktionsrumpf

Im Funktionsrumpf werden die Variablen definiert und die jeweiligen Operationen durchgeführt. Welcher Programmtext dort stehen soll, hängt von der Aufgabe der Funktion ab. Man kann dort alle Grundrechenarten ausführen und Funktionen, wie *`printf()`* und *`scanf()`*, aufrufen.

Bei der Definition der Variablen brauchst Du übrigens nicht darauf zu achten, ob diese schon in der Funktion *`main()`* vorkommen. Das ist für die Funktion vollkommen uninteressant. Die Variablennamen dürfen nur nicht mit den Parameternamen übereinstimmen.

Hinweis:

Wenn Du die Werte der Parameter änderst, so werden diese nicht unbedingt auch im Programm geändert. Das mußt Du bei der Programmierung berücksichtigen.

5. 6. Rückgabewert

Wenn man nun die Funktion ausgeführt hat, so kann es sein, daß die Funktion einen Wert zurück liefern muß. Im Beispiel wäre dies der Wert für *mx + n*.

Um einen Wert zurückzugeben, benutzt man die Funktion *return()*. Sie wird folgendermaßen eingesetzt:

```
return (rückgabewert);
```

Der Datentyp des Rückgabewertes wurde bereits bei der Definition der Funktion als *Rückgabedatentyp* festgelegt. Beim Aufruf der Funktion kann dieser Rückgabewert einer Variable zugewiesen werden.

Soll eine Funktion keinen Wert zurück liefern, so wird im Funktionsrumpf statt des Datentyps das Wort **void** eingetragen. Dies besagt also, daß kein Wert zurück gegeben wird. Wir benutzen void schon lange bei der Funktion *main()*.

5. 7. Aufruf der Funktion

Eine Funktion wird folgendermaßen aufgerufen:

```
funktionsname (parameter1, ..., parameter_n);
```

Sofern die Funktion einen Wert zurück liefert, kann vor dem Funktionsaufruf auch noch eine Variable stehen:

```
variable = funktionsname (parameter1, ..., parameter_n);
```

Der Aufruf der Funktion erfolgt in der Funktion *main()*.

5. 8. Das komplette Beispiel

Am Anfang des Kapitels hatten wir die mathematische Funktion *y = f(x) = mx + n*. In C programmiert, könnte sie beispielsweise so aussehen:

```
float f (float x)
{
   float m, n, y;
   printf("\nEingabe von m: ");
   scanf("%f", &m);
   printf("\nEingabe von n: ");
   scanf("%f", &n);
   y = m * x + n;
   return(y);
}
```

Die Funktion hat also den Namen *f* und den Parameter *x*. Als Rückgabedatentyp haben wir *float*. In der Funktion sind die 3 Variablen *m*, *n* und *y* definiert, wovon zwei durch die Funktion *scanf()* noch eingelesen werden müssen. Die Variable *y* wird berechnet und durch die Funktion *return()* zurückgegeben.

Das Programm, welches diese Funktion aufruft kann beispielsweise so aussehen:

```
void main()
{
   float x, y;
   ...
   y = f(x);
   ...
}
```

Die Funktion *f(x)* wird also vom Programm aufgerufen und ihr Rückgabewert der Variable *y* zugewiesen.

5. 9. Rekursion

Bei manchen Aufgabenstellung kann es vorkommen, daß eine Funktion sich selbst aufrufen muß. Solch eine Funktion nennt man rekursive Funktion. Du findest eine solche Aufgabenstellung bei den Übungsaufgaben.

Du mußt darauf achten, daß die Funktion sich zwar selbst aufruft, allerdings mit anderen Parametern. Tust Du das nicht, so kommt Dein Programm nie zum Ende.

5. 10. Einige vordefinierte Funktionen

Hier findest Du einige Funktionen, die in der C - Programmierung schon festgelegt sind und die Du benutzen kannst. Einige davon kennst Du bereits.

`clrscr`
Parameter: keine
Rückgabewert: keiner
Aufruf: `clrscr();`
Aufgabe: löscht den Bildschirm
Bibliothek: `conio.h`

`gotoxy`
Parameter: int x, int y
Rückgabewert: keiner
Aufruf: `gotoxy(2,3);`
Aufgabe: bewegt den Cursor zu dieser Stelle auf dem Bildschirm
Bibliothek: `conio.h`

`printf`
Parameter: Zeichenkette, Variablen, die in dieser Zeichenkette vorkommen
Rückgabewert: keiner
Aufruf: `printf("Dies ist ein Test");`
Aufgabe: Druckt die Zeile auf dem Bildschirm aus.
Bibliothek: `stdio.h`

`scanf`
Parameter: Zeichenkette, Adreßoperator `&` mit der Variable, die den Wert bekommen soll
Rückgabewert: keiner
Aufruf: `scanf("%f", &x);`
Aufgabe: Schreibt den eingelesenen Wert in die Variable.
Bibliothek: `stdio.h`

`sqrt`
Parameter: Zahl
Rückgabewert: Zahl vom Datentyp float
Aufruf: `wurzel = sqrt(3);`
Aufgabe: Berechnet die Quadratwurzel.

Bibliothek: *math.h*

getch
Parameter: keine
Rückgabewert: Zeichen, vom Typ char
Aufruf: *zeichen = getch();*
Aufgabe: liest ein Zeichen, welches über die Tastatur eingegeben wurde ein und schreibt es in die Variable.
Bibliothek: *conio.h*

5. 11. Aufgaben

1. Schreib eine Funktion **inhalt**, welche die Punkte eines Rechtecks eingeben läßt und dessen Flächeninhalt berechnet. Ausgegeben werden soll der Flächeninhalt im dazugehörenden Testprogramm. Wir gehen davon aus, daß die Kanten des Rechtecks parallel zu den Achsen des Koordinatensystems verlaufen.

Hinweis: Die Punkte kannst Du über eindimensionale Arrays der Größe 2 eingeben lassen.

Lösungsvorschlag:

lsg_5_1.c	**933 Byte**	**04. 06. 2001**

2. Schreib eine Funktion **ggt**, welche den größten gemeinsamen Teiler zweier ganzer Zahlen berechnet. Die Zahlen sollen vom Programm übergeben werden und der größte gemeinsame Teiler zurückgegeben werden.

Hinweis: Die Funktion muß rekursiv sein.

Lösungsvorschlag:

lsg_5_2.c	**535 Byte**	**04. 06. 2001**

3. Schreib ein Programm, welches den Sinus einer eingegebenen Zahl berechnet. Benutze dabei die Funktion *sin(x)*, welche Dir noch unbekannt ist. Informiere Dich über diese Funktion selbständig. (Die Sinusfunktion ist auf "Radiant" eingestellt, falls Du nachrechnest.)

Wichtig: Du mußt die Bibliothek *math.h* einbinden !

Lösungsvorschlag:

lsg_5_3.c	**375 Byte**	**04. 06. 2001**

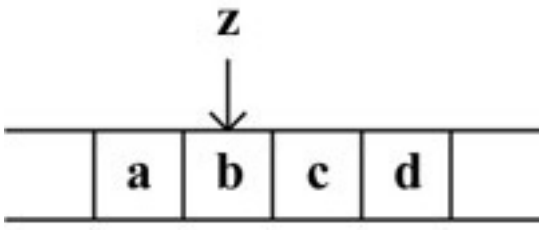
6. Zeiger

6. 1. Allgemeines

Zeiger spielen in der Programmierung eine wichtige Rolle. Sie sind wichtig, bei der Arbeit mit Arrays und Funktionen.

Ein Zeiger ist eine Variable, deren Inhalt die Adresse einer anderen Variable im Hauptspeicher des Computers ist. Mit dieser Adresse kann man auf die Variable zugreifen.

Die folgende Abbildung soll dies noch einmal verdeutlichen:



Der Zeiger *z* zeigt hierbei im Hauptspeicher auf die Variable *b*.

Bei der Arbeit mit Zeigern benötigen wir 2 Operatoren:

&	Adreßoperator, wird vor die Variable geschrieben und liefert die Adresse der Variable zurück (siehe Funktion scanf !)
*****	Verweisoperator, wird vor den Zeiger geschrieben und liefert den Wert der Variable zurück, auf die der Zeiger gerichtet ist.

6. 2. Definition eines Zeigers

Ein Zeiger wird folgendermaßen definiert:

```
datentyp *zeigername;
```

Zu beachten ist dabei, daß der Zeiger immer nur auf Variablen vom angegebenen Datentyp zugreifen kann. Hat der Zeiger beispielsweise den Datentyp *int*, so kann er nicht auf Variablen vom Datentyp *float* zeigen.

Es besteht wiederum die Möglichkeit, daß mehrere Zeiger vom gleichen Datentyp hintereinander definiert werden. Das kennst Du aber schon von der Definition von einfachen Variablen.

6. 3. Zugriff auf Zeiger

6. 3. 1. Adresse zuweisen

Auf einen Zeiger kann auf 3 Arten zugegriffen werden. Man kann dem Zeiger die Adresse einer Variablen zuweisen, man kann die Adresse, auf die der Zeiger zeigt, auslesen und den Wert der Variablen, auf die der Zeiger zeigt, auslesen.

Man weist einem Zeiger folgendermaßen eine Adresse zu:

```
zeiger = &variablen;
```

Dem Zeiger wird also die Adresse der Variablen zugewiesen.

6. 3. 2. Adresse auslesen

Um die Adresse, auf die der Zeiger zeigt, auszulesen, gibt man einfach nur den Zeigernamen an. Um die Adresse auszugeben, benötigst Du in der Funktion `printf()` den Formatstring `%p` (p = pointer, engl. Zeiger).

Die Adresse wird in hexadezimaler Form ausgegeben. Hexadezimale Zahlen werden folgendermaßen in das duale Zahlensystem umgerechnet:

0 = 0
1 = 1
2 = 2
3 = 3
4 = 4
5 = 5
6 = 6
7 = 7
8 = 8
9 = 9
A = 10
B = 11
C = 12
D = 13
E = 14
F = 15

Die Zahlen werden von rechts nach links betrachtet. Die erste Zahl wird mit 1 multipliziert, die zweite mit 16, die dritte mit 16*16 usw. Die Produkte werden miteinander addiert und damit hat man die hexadezimale Zahl in eine Zahl zur Basis 2 umgerechnet.

Bei einer Zahl FF32D wird also folgendermaßen gerechnet:

15*65536 + 15*4096 + 3*256 + 2*16 + 13*1 = 1045293

Wenn der Zeiger also die Adresse FF32D hat, so zeigt er auf die Adresse 1045293.

6. 3. 3. Wert der Adresse auslesen

Wenn Du den Wert der Variablen, auf deren Adresse der Zeiger gerichtet ist, auslesen möchtest, so mußt Du den Verweisoperator vor den Zeigernamen setzen:

```
*zeigername
```

Diese Zeichenkette kannst Du wiederum in eine `printf()` - Anweisung einbinden. Du hast außerdem die Möglichkeit, den Wert der Variablen zu ändern.

Wenn beispielsweise der Zeiger `z` auf die Variable `v` zeigt, so kann man den Wert der Variablen auch folgendermaßen ändern:

```
*z = neuerWert;
```

Der Variable `v` wurde also ein neuer Wert zugewiesen.

6. 4. Zeiger in Funktionen

Wie schon in Kapitel 5.5 erwähnt, werden die Werte von Parametern in Funktionen nicht geändert, auch wenn man es scheinbar tut. Hat man dies aber vor, so kann man statt der Variablen, Zeiger auf diese Variablen übergeben. Dann ist es möglich, die Werte der Parameter zu ändern.

Bei der Definition der Funktion muß der Zeiger in der Parameterliste eingetragen werden:

```
rückgabewert funktionsname(datentyp *zeigername)
```

In der Funktion kann man auf den Wert der Variable mittels Verweisoperator * zugreifen.

Im Hauptprogramm sieht der Aufruf der Funktion dann folgendermaßen aus:

```
funktionsname(zeigername);
```

Du hast auch die Möglichkeit, gleich die Adresse der Variablen zu übergeben. Du benötigst dann nicht extra einen Zeiger.Der Aufruf sieht dann folgendermaßen aus:

```
funktionsname(&variablenname);
```

Es wird also die Adresse übergeben, was ja einem Zeiger entspricht.

6. 5. Zeiger auf Arrays

In Kapitel 4 wurden für die einzelnen Feldelemente von Arrays Indizes benutzt. Alternativ dazu kann man auch Zeiger benutzen.

Dazu wird zuerst ein Feld definiert:

```
datentyp feldname[groesse];
```

Zusätzlich wird ein Zeiger definiert, der auf den gleichen Datentyp zeigen kann:

```
datentyp *zeigername;
```

Dieser Zeiger wird nun auf das Array gesetzt:

```
zeigername = feldname;
```

Diese Zeile bewirkt, daß der Zeiger auf das 1. Feldelement zeigt, also `feldname[0]`.

Will man nun auf eines der Feldelemente zugreifen, so mußt man einfach den Index zum Zeiger hinzu addieren.

```
*(zeigername + 3) = 10;
```

würde dann also bewirken, daß `feldname[3]` den Wert `10` zugewiesen bekommen würde.

6. 6. Anwendung von Zeigern: dynamische Speicherplatzzuweisung

Bei Feldern hatten wir bisher nur die Möglichkeit, die Größe am Beginn des Programms festzulegen. Dies bringt Nachteile mit sich, wenn man nicht so viele Feldelemente benötigt, oder noch schlimmer, wenn man mehr benötigt, als man vorher festgelegt hat.

Dazu benutzt man die Funktion `malloc()`. Diese Funktion legt die Größe während des Programmablaufs fest. Wenn man beispielsweise eine Variable vom Typ `Integer` definiert, so wird im Hauptspeicher Platz in der Größe von 2 Byte reserviert. Genau dies macht auch die Funktion `malloc()`.

Als erstes wird ein Zeiger definiert:

```
datentyp *zeigername;
```

Die Funktion `malloc()` wird dann wie folgt aufgerufen:

```
zeigername = (datentyp *) malloc (groesse *
sizeof(datentyp));
```

Die Funktion `sizeof()` liefert den benötigten Speicherplatz für den jeweiligen Datentyp zurück. Im Fall Integer wäre dies also 2.

Den Zeiger `zeigername` kannst Du nun wie einen normalen Zeiger auf ein Array benutzen. Das hast Du ja schon in Kapitel 6.5 gelernt.

Wenn Du mit der Verarbeitung fertig bist, so mußt Du den Speicher wieder freigeben. Dies tust Du mit der Funktion `free()`. Sie wird folgendermaßen aufgerufen:

```
free(zeigername);
```

Wichtig!

Um die Funktion `malloc()` benutzen zu können, mußt Du die vordefinierte Bibliothek `malloc.h` mit `#include` einbinden.

6. 7. Aufgaben

1. Schreib ein Programm, das eine beliebige Anzahl von Noten einliest. Die Anzahl soll bei Programmstart vom Nutzer abgefragt werden. Das Programm soll folgende Aufgaben erfüllen:

- Einlesen der Noten
- Bestimmen von Minimum und Maximum
- Berechnung des Durchschnittswertes
- Anzeigen einer Liste aller Noten mit Durchschnitt, Minimum und Maximum

Realisiere das ganze Programm mit einem Zeiger und dynamischer Speicherplatzzuweisung.

Lösungsvorschlag:

lsg_6_1.c **1330 Byte** **04. 06. 2001**

2. Schreib eine Funktion **`tausch`**, welche zwei übergebene ganze Zahlen vertauscht. Schreib ein dazugehöriges Hauptprogramm zum testen.

Hinweis: Übergib statt der beiden ganzen Zahlen die Adressen der beiden Zahlen und arbeite dann mit den Zeigern.

Lösungsvorschlag:

lsg_6_2.c **479 Byte** **04. 06. 2001**

3. Gegeben sind 2 Integervariablen:

Name: zahl1 Wert: 5 Adresse: FD00	Name: zahl2 Wert: 10 Adresse: FD02

Schreib nach jedem Schritt auf, welche Werte sich ändern:

- a) zeiger1 = &zahl1;
- b) zeiger2 = &zahl2;
- c) *zeiger1+=20;
- d) *zeiger2-=3;
- e) *zeiger1 = *zeiger2 + 5;
- f) *zeiger2 = *zeiger1 - zahl2;

Lösung:

lsg_6_3.txt **182 Byte** **04. 06. 2001**

7. Zeichenketten

7. 1. Allgemeines

In Kapitel 1 hast Du bereits den Datentyp *char* für einzelne Zeichen kennengelernt. In diesem Kapitel lernst Du, wie man ganze Worte in Variablen speichert. Dies sind Felder vom Datentyp *char*.

Zeichenketten werden in der Programmierung auch **Strings** genannt.

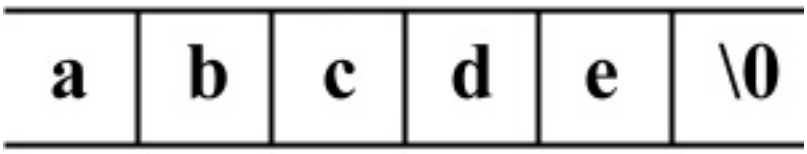
7. 2. Definition

Ein String wird folgendermaßen definiert:

```
char stringname[stringlaenge];
```

Es wird also ein Feld vom Datentyp *char* angelegt, welches die Größe *stringlaenge* hat.

Der eingelesene String ist folgendermaßen aufgebaut:



Dabei kennzeichnet das Zeichen `'\0'` das Ende der Zeichenkette. Wenn Du also einen String definierst, so mußt Du bedenken, daß bei ihm ein Zeichen für das Ende freigehalten werden muß. In Felder der Größe 20 passen demnach nur Zeichenketten der Länge 19.

Dieses Zeichen für das Ende hat jedoch auch seine Vorteile. Wenn Du beispielsweise nach dem Ende einer Zeichenkette suchst, so brauchst Du nur nach diesem Zeichen zu suchen.

7. 3. Wichtige Funktionen für Zeichenketten

Es gibt eine Fülle von Funktionen für Zeichenketten. Hier werde ich nur 5 vorstellen, die jedoch wichtig sind.

1. **gets(***zeichenkette***);**

Funktioniert prinzipiell wie die Funktion *scanf()*. Du rufst die Funktion auf und sie schreibt Dir in die Zeichenkette *zeichenkette* den eingegebenen String. Wichtig ist natürlich, daß die Zeichenkette vorher definiert wurde.

2. **puts(***zeichenkette);*

Funktioniert wie die Funktion *printf()*. Bei Aufruf gibt die Funktion die angegebenene Zeichenkette aus. Natürlich muß auch hier vorher die Zeichenkette definiert worden sein.

3. **strlen(***zeichenkette***);**

Diese Funktion gibt Dir die Länge einer angegebenen Zeichenkette zurück. Wenn Du also eine Integer-Variable *laenge* hast, so wird beim Aufruf

```
laenge = strlen(zeichenkette);
```

die Länge in die Variable geschrieben. Diese Funktion gibt nur die Anzahl der tatsächlich eingegebenen Zeichen zurück.

4. **strcmp(***zeichenkette1, zeichenkette2***);**

Diese Funktion vergleicht zwei Zeichenketten. Sie liefert einen Wert kleiner 0 zurück, wenn die *zeichenkette1* kleiner als die *zeichenkette2* ist. Ist das Umgekehrte der Fall, so liefert sie einen Wert größer 0 zurück. Bei gleichen Zeichenketten ist der Rückgabewert 0.

5. **strcpy(***ziel, quelle);*

Kopiert den Inhalt der Zeichenkette *quelle* in die Zeichenkette *ziel*.

7. 4. Umwandlung von Zeichenketten in Zahlen

Für die Umwandlung von Zeichenketten in Zahlen benötigen wir zwei Funktionen:

1. **integerzahl = atoi(***zeichenkette***);**

Wandelt eine Zeichenkette in eine Zahl vom Datentyp *int* um.

2. **floatzahl = atof(***zeichenkette***);**

Wandelt eine Zeichenkette in eine Zahl vom Datentyp *float* um.

Wenn eine Zeichenkette nicht in eine Zahl umgewandelt werden kann (z.B. "abc"), dann wird eine Null zurückgegeben.

Achtung !
Um diese Funktionen nutzen zu können, mußt Du die Bibliotheken **stdlib.h** und **math.h** mit **#include** einbinden !

7. 5. Zeiger auf Strings

Zeiger auf Strings werden genauso benutzt, wie Zeiger auf Arrays anderer Datentypen. Da ein String nur ein Array vom Datentyp *char* ist, macht das keinen Unterschied.

7. 6. Aufgaben

1. Schreib ein Programm, welches den Benutzer fragt, ob er weitermachen will. Es werden als Antworten nur die Zeichenketten "Ja" und "Nein" akzeptiert. Zum Start des Programms soll die Ausgabe "1. Versuch" erscheinen. Bei jedem weiteren "Ja" erscheint der nächste Versuch ("2. Versuch", "3. Versuch", ...).

Lösungsvorschlag:

lsg_7_1.c **692 Byte** **04. 06. 2001**

2. Schreib ein Programm, welches alle r und R eines eingegebenen Satzes (max. 99 Zeichen) in l und L umwandelt. Dies ergibt dann einen Satz, welcher ausgesprochen sich anhört, wie Deutsch mit chinesischem Akzent.

Lösungsvorschlag:

lsg_7_2.c **564 Byte** **04. 06. 2001**

3. Schreib ein Programm, welches einen elektronischen Psychologen darstellt. Die erste Frage lautet: "Über welches Problem möchtest Du mit mir reden ?". Nachdem der Benutzer eine Antwort gegeben hat, fragt er: "Erzähl mir mehr darüber." Wenn der Benutzer die Zeichenkette "Bye" eingibt, wird das Programm beendet. Denk Dir noch drei weitere Reaktionen auf Eingaben und implementiere das Ganze durch eine while - Schleife.

Lösungsvorschlag:

lsg_7_3.c **831 Byte** **04. 06. 2001**

8. Arbeiten mit Dateien

8. 1. Allgemeines

Bisher haben wir Daten, die wir in unseren Programmen erzeugt und verarbeitet haben, nur über den Bildschirm ausgeben können. Nach Beendigung eines Programms gingen diese Daten unwiderruflich verloren. Mit Hilfe von Dateien läßt sich dies ändern.

Die Arbeit mit einer Datei folgt immer dem gleichen Schema:

- **Datei öffnen**
- **Daten auslesen, verändern oder anhängen**
- **Datei schließen**

Wie diese Vorgänge im einzelnen gehandhabt werden, wird in diesem Kapitel beschrieben.

8. 2. Öffnen einer Datei

Bevor man eine Datei öffnen kann, benötigt man einen Zeiger auf die Struktur **`FILE`**. Eine Struktur ist eine Ansammlung von verschiedenen Variablen, die unter einem Namen zusammengefaßt werden. Strukturen werden in Kapitel 9 behandelt werden.

Der Zeiger auf die Struktur wird wie jeder andere Zeiger auch definiert:

```
FILE *zeigername;
```

Es ist vorteilhaft, wenn man als Zeigername die Aufgabe, die er zu erfüllen hat, benutzt. Wenn man eine Datei lesen möchte, so kann man diesem Zeiger beispielsweise den Namen *lesen* geben. Natürlich ist es Dir freigestellt auch andere Namen zu verwenden.

Als nächstes folgt die Öffnung der Datei. Dies geschieht mit der Funktion *`fopen()`*. Sie wird folgendermaßen aufgerufen:

```
zeigername = fopen (dateiname, modus);
```

Die Funktion schreibt also verschiedene Werte in die Variablen der Struktur **`FILE`** und weist die Adresse dieser Struktur dem Zeiger *zeigername* zu. Der Zeiger wird dabei an den Anfang der Datei gesetzt.

Als Dateiname wird ein Array vom Datentyp **`char`** verwendet.

Beim Modus gibt es verschiedene Dinge zu beachten. Die Zeichenkette *modus* beschreibt, wie eine Datei geöffnet werden soll. Hier sind die möglichen Modi:

w	write, schreibender Zugriff
r	read, lesender Zugriff
a	append, Daten an die Datei anhängen
t	Öffnen im Textmodus
b	Öffnen im Binärmodus

Wenn Du also in eine Datei schreiben möchtest, so mußt Du als Modus **"w"** angeben. Möchtest Du allerding mehrere Modi gleichzeitig nutzen, z. B. lesen und schreiben, so kannst Du dies nur folgendermaßen machen:

- "w+"
- "r+"
- "a+"

Mit diesen Zeichenfolgen kannst Du sowohl lesen, als auch schreiben.

Zusätzlich kannst Du noch angeben, ob die Datei binär oder im Textmodus geöffnet werden soll. Binärdateien sind unter anderem ausführbare Dateien mit der Endung .exe, während Textdateien auch HTML-Dateien für das Internet sind.

Wenn Du also möchtest, daß Du eine Datei lesen und in sie schreiben möchtest und diese zusätzlich noch im Binärmodus geöffnet werden soll, so benötigst Du die Zeichenkette "r+b". Gibst Du nicht explizit an, in welchem Modus die Datei geöffnet werden soll, so wird sie standardmäßig im Textmodus geöffnet.

Abschließend ein Beispielaufruf:

```
schreiben = fopen ("datei1.txt", "w+");
```

In die Datei *datei1.txt* kann also geschrieben werden. Sie kann aber auch gelesen werden.

Achtung ! Wenn Du eine Datei mit "w" öffnest, so wird der gesamte Inhalt der Datei gelöscht.

8. 3. Fehler beim Öffnen abfangen

Nun kann es passieren, daß Fehler beim Öffnen der Datei auftreten. Ursachen können sein:

- **Die Datei, aus der gelesen werden soll, existiert nicht.**
- **Die Datei, in die geschrieben werden soll, kann nicht angelegt werden (z. B. kein Speicherplatz mehr frei, Schreibschutz im Diskettenlaufwerk, versehentlich CD-ROM-Laufwerk gewählt)**

Dies ist sehr leicht mit dem zuvor definierten Zeiger abzufangen. Tritt ein Fehler beim Öffnen oder Schreiben der Datei auf, so wird an den Zeiger nicht eine Adresse gegeben sondern NULL. Dies bedeutet, daß ein Zeiger keine Adresse hat.

Wenn Du also überprüfen willst, ob die Datei ordnungsgemäß geöffnet oder angelegt wurde, so mußt Du nur überprüfen, ob der Zeiger auf diese Datei nicht NULL ist. Andernfalls mußt Du eine entsprechende Fehlermeldung ausgeben und darauf reagieren.

8. 4. Schreiben in eine Datei

Um in eine Datei schreiben zu können, muß in der Funktion *fopen()* der Modus *schreibend* angegeben worden sein.

Zum schreiben in Dateien gibt es 3 wichtige Funktionen: *fprintf()*, *fputs()* und *fputc()*.

fprintf() ist eng verwandt mit der Funktion *printf()*. Sie wird folgendermaßen aufgerufen:

```
fprintf(zeigername, "text mit formatstrings",
variablenname);
```

Der Unterschied zu *printf()* ist offensichtlich. Vor dem Ausgabetext wird der Zeiger auf die Struktur **FILE** übergeben. Der Text mit den Formatstrings wird also in die Datei geschrieben.

fputs() ist wiederum eng verwandt mit der aus Kapitel 7 bekannten Funktion *puts()*. Sie wird folgendermaßen aufgerufen:

```
fputs(zeichenkette, zeigername);
```

Die Zeichenkette wird also in die Datei geschrieben. Im Unterschied zu *fprintf()* wird der Zeigername als letzter Parameter an die Funktion *fputs()* übergeben.

fputc() ist so ähnlich wie die Funktion *fputs()*. Allerdings wird hier nicht eine Zeichenkette, sondern nur ein Zeichen in die Datei geschrieben. Zeichen sind bekannterweise Daten vom Typ **char**. Die Funktion wird folgendermaßen aufgerufen:

```
fputc(zeichen, zeigername);
```

Achtung ! Wenn Du eine Datei mit "w" (schreiben) öffnest, so wird der gesamte Inhalt überschrieben. Du mußt, wenn Du nicht alles überschreiben möchtest, die Datei mit "r+" oder "a" öffnen.

8. 5. Lesen aus einer Datei

Beim Schreiben haben wir wiederum drei Funktionen, die wir schon zu kennen scheinen: *fscanf()*, *fgets()* und *fgetc()*.

Allerdings müssen wir hier einen wichtigen Punkt berücksichtigen. Wir dürfen nur bis zum Ende der Datei lesen. Gehen wir darüber hinaus, so stürzt das Programm im schlimmsten Fall ab. Das Dateiende wird durch **EOF** (End Of File) gekennzeichnet. Wenn man also eine Datei ausliest, so muß man immer wieder abfragen, ob das eingelesene Zeichen **EOF** entspricht. Dies läßt sich immer gut mit einer *while*-Schleife realisieren.

Nun jedoch zu den Funktionen.

fscanf() ist, wie soll es anders sein, verwandt mit der Funktion *scanf()*. Sie wird folgendermaßen aufgerufen:

```
fscanf(zeigername, "formatstring", &variablenname);
```

Der Wert wird also aus der Datei in die Variable geschrieben.

Wenn der Wert gelesen wurde, so wird der Zeiger auf das Ende des Wertes gesetzt, damit der nächste Wert eingelesen werden kann.

fgets() liest einen String aus der Datei aus und schreibt ihn in ein Array vom Datentyp **char**. Sie wird folgendermaßen aufgerufen

```
fgets(variablenname, laenge, zeigername);
```

Dir wird es bestimmt aufgefallen sein, daß hier noch eine Länge angegeben werden muß. Da wir für die Variable *variablenname* nicht unendlich viel Speicherplatz reserviert haben, daß Array also nur eine begrenzte Größe hat, muß die Größe des Arrays angegeben werden. Es werden dann bis zu *laenge - 1* Zeichen eingelesen, da in das letzte Arrayelement noch das Zeichen **\0** geschrieben wird. Die Funktion beendet das Einlesen außerdem, wenn das Ende einer Zeile erreicht wurde. Es wurde also das Zeichen **\n** eingelesen.

fgetc() liest ein Zeichen aus einer Datei aus. Sie wird folgendermaßen aufgerufen:

```
variable = fgetc(zeigername);
```

Der Wert, der aus der Datei eingelesen wurde, wird in die Variable geschrieben. Dabei ist zu beachten, daß die Variable selbstverständlich vom Datentyp **char** sein muß.

8. 6. Anhängen an eine Datei

Wenn Du eine Datenbank hast, so kann es sein, daß Du in sie neue Datensätze eintragen willst. Wenn Du an eine Datei mehrere Daten anhängen willst, so mußt Du die Datei im Modus **"a"** öffnen. Der Zeiger wird dabei auf das Ende der Datei gesetzt. Von dort aus kannst Du die Daten in die Datei eintragen.

Wenn Du Daten voneinander trennen willst, so ist es vorteilhaft, wenn Du zwischen den Datensätzen das Zeichen **\n** schreibst. Dies erkennen die Funktionen zum lesen und reagieren darauf.

8. 7. Schließen der Datei

Wenn Du alle Arbeiten an der Datei abgeschlossen hast, so muß die Datei wieder geschlossen werden. Du erinnerst Dich sicher aus Kapitel 6, daß Speicherbereich, welcher Zeigern mit der Funktion *malloc()* zugewiesen worden wurde, durch die Funktion *free()* wieder freigegeben werden mußte. So ähnlich geschieht das mit den Dateien.

Um eine Datei zu schließen, mußt Du die Funktion *fclose()* benutzen. Sie wird folgendermaßen aufgerufen:

```
fclose(zeigername);
```

Damit wird die Datei geschlossen und kann an anderer Stelle wieder geöffnet werden.

8. 8. Aufgaben

1. Schreib ein Programm, welches sämtliche Umlaute einer Textdatei (ä, ü, ..) in die dazugehörigen Konsonanten umwandelt (ae, ue). Die Daten sollen aus einer Datei eingelesen werden und in eine neue geschrieben werden.

Lösungsvorschlag:

lsg_8_1.c **1244 Byte** **04. 06. 2001**

2. Schreib ein Programm, welches eine Textdatei verschlüsselt. Dabei soll eine einstellige Zahl eingegeben werden. Jeder Buchstabe soll durch den Nachfolger ersetzt werden, welcher durch die Zahl festgelegt wurde. Die Buchstabenfolge **"abcd"** würde bei einer eingegeben Zahl **3** dann **"defg"** lauten. Dabei ist zu beachten, daß bei einer eingegeben Zahl **1** der Nachfolger von **z** der Buchstabe **a** ist. Die eingelesenen Buchstaben sollen in eine neue Datei geschrieben werden, damit das Original erhalten bleibt.

Hinweise:

Zeichen sind in der Programmierung nichts anderes als Zahlen. Du kannst auf ein Zeichen also eine Zahl addieren und bekommst daraufhin dessen Nachfolger.

Das Alphabet wird in der Programmierung intern durch Zahlen vertreten. Die Großbuchstaben gehen dabei von 65 bis 90 und die Kleinbuchstaben von 97 bis 122.

Lösungsvorschlag:

lsg_8_2.c **1709 Byte** **04. 06. 2001**

3. Schreib ein Programm, welches die in der vorherigen Aufgabe verschlüsselte Datei wieder entschlüsselt. Dabei muß wiederum die einstellige Ziffer eingegeben werden.

Lösungvorschlag:

lsg_8_3.c **1708 Byte** **04. 06. 2001**

9. Strukturen

9. 1. Allgemeines

Wenn wir bisher einfache Aufgaben zu lösen hatten, so benötigten wir einfache Variablen. Hatten wir aber komplexe Sachverhalte, z. B. Daten von Personen, die registriert werden mußten, so benötigten wir verschiedene Variablen. Beispielsweise benötigt eine Adresse allein mindestens 4 Variablen um den Namen, die Straße mit Hausnummer, die Stadt und die Postleitzahl zu speichern. Bei noch komplexeren Sachverhalten würde der Programmtext schnell unübersichtlich werden.

Eine Struktur beschafft uns dabei Abhilfe. Sie kann verschiedene Variablen zu einer logischen Einheit zusammenfassen. Das Programm wird dabei übersichtlicher und verständlicher.

9. 2. Definition einer Struktur

Eine Struktur wird folgendermaßen festgelegt:

```
struct strukturname {
   datentyp1 variable1;
   datentyp2 variable2;
   ...
   datentyp_n variable_n;
};
```

Die Struktur *strukturname* enthält also die Variablen *variable1* bis *variable_n*. Die Definition der Struktur kann sowohl vor als auch in der Funktion *main()* erfolgen.

Achtung !
Das Semikolon nach der geschlossenen geschweiften Klammer nicht vergessen ! Der Compiler würde dies als Fehler ansehen !

Die Struktur, die nun erstellt wurde, kann wie ein normaler Datentyp behandelt werden. Du kannst also Variablen vom Datentyp der Struktur anlegen. Dies kann auf 2 verschiedenen Wegen passieren.

1. Definiere eine Variable, wie Du das bisher gemacht hast. Also wie folgt:

```
struct strukturname variable;
```

Die Variable ist dann vom Datentyp der Struktur.

2. Definiere die Variable bei der Definition der Struktur.

```
struct strukturname {
   ...
} variable1, variable2, ..., variable_n;
```

Die Variable *variable1* bis *variable_n* sind dann also vom Datentyp der Struktur.

Die beiden Wege unterscheiden sich folgendermaßen:

Beim ersten Weg hast Du die Variable nur in der Funktion *main()* definiert. Die Variable ist also nur in dieser Funktion gültig. Wenn Du die Variable in anderen Funktionen verwenden willst, mußt Du sie als Parameter übergeben.

Beim zweiten Weg hast Du die Variable für das gesamte Programm definiert. Sie ist in allen Funktionen gültig. Du mußt also nicht extra die Variable als Parameter übergeben.

9. 3. Zugriff auf die Elemente der Struktur

Der Zugriff auf die Elemente der Struktur ist sehr einfach. Er erfolgt folgendermaßen:

```
variable.strukturvariable
```

Mit dem Punkt nach dem Variablennamen wird auf die Struktur zugegriffen. Nach dem Punkt wird der Variablenname in der Struktur angegeben. Danach kannst Du mit der Variable arbeiten, wie Du es mit anderen normalen Variablen kennst. Du kannst ihr Werte zuweisen, Werte auslesen und Werte manipulieren.

9. 4. Variable Strukturen

In der Programmierpraxis kann es Dir passieren, daß Du Strukturen benötigst, welche sich den Aufgaben anpassen. Hast Du beispielsweise die Aufgabe, Studenten einer Universität und deren Leistungen zu speichern, so wird Dir eine einfache Struktur nicht helfen, da Studenten verschiedener Studiengänge, unterschiedliche Fächer belegen, in denen sie Leistungen erbringen müssen. Dafür sind variable Strukturen zuständig. Am folgenden Beispiel soll dieses Programmierelement demonstriert werden.

Ein Fachbereich hat zwei Studiengänge, Informatik und Information Management. Für die beiden Studiengänge sollen die Leistungen aller Studenten für das 1. Fachsemester in einer Struktur gespeichert werden. Die Informatiker haben dabei die Fächer Mathematik, Programmierung, Datenbanken und Theoretische Informatik. Die Information Manager haben die Fächer Mathematik, Wirtschaft, Datenbanken und Theoretische Informatik. Die Studiengänge unterscheiden sich im 1. Fachsemester nur in den Fächern Programmierung und Wirtschaft (ist in der Wirklichkeit natürlich ganz anders).

Um nun eine variable Struktur zu erstellen, benutzen wir das Schlüsselwort *union*. Dieses Wort definiert ebenfalls wie das Wort *struct* eine Struktur. Allerdings werden dabei nicht alle Variablen der Struktur benutzt, sondern nur eine Variable. Wenn also eine Struktur 3 Variablen hat, so werden *variable1* und *variable2* und *variable3* benutzt. Bei *union* wird *variable1* oder *variable2* oder *variable3* benutzt. Damit lassen sich dann ganz leicht variable Strukturen erstellen.

Im Beispiel könnte die Struktur student folgendermaßen aussehen:

```
struct student {
   char name[20];
   char vorname[20];
   int matrikelnummer;
   char studiengang[20];
   int mathematik;
   union {
      int programmierung;
      int wirtschaft;
   } zweites_fach;
   int datenbanken;
   int theo_inf;
} student1;
```

Nun muß man im Programm abfragen, welcher Studiengang eingegeben wurde. Ist der Studiengang Informatik, so muß eine Note für die Programmierung eingegeben werden:

```
student1.zweites_fach.programmierung = 1;
```

Ist der Student vom Studiengang Information Management, so muß eine Note für Wirtschaft eingegeben werden:

```
student1.zweites_fach.wirtschaft = 3;
```

Wie Du siehst, ist die Handhabung einer solchen variablen Struktur ziemlich einfach. Da *union* im Prinzip auch eine Struktur ist, ist zu erkennen, daß innerhalb einer Struktur eine Unterstruktur erzeugt werden kann. Das kannst Du auch mit *struct* machen.

9. 5. Felder von Strukturen

Da man Stukturen wie normale Datentypen behandeln kann, ist es auch möglich, Felder von Strukturen zu erstellen. Du mußt dazu nur nach dem Variablennamen die Größe des Arrays angeben:

```
struct strukturname variablenname[feldgröße];
```

Auf die einzelnen Elemente der Struktur kannst Du dann wie folgt zugreifen:

```
variablenname[index].strukturvariable
```

9. 6. Zeiger auf Strukturen

Zeiger auf Strukturen benötigen wir in der Hauptsache bei dynamischen Arrays. Wie ein Zeiger auf eine Variable angelegt wird, ist Dir bereits bekannt. Genauso verfährst Du mit Strukturen:

```
struct strukturname *zeigername;
```

Nun kannst Du ein dynamisches Array erstellen:

```
zeigername = (strukturname *) malloc (feldgröße * (sizeof
(strukturname));
```

Um nun auf die einzelnen Feldelemente zugreifen zu können, lernen wir einen neuen Operator kennen, den Pfeiloperator.

Der Zeiger ist auf das erste Element des Arrays gerichtet. Nun wollen wir in dem Feldelement einen Wert ändern. Bisher haben wir folgendes geschrieben um auf eine Variable innerhalb der Struktur zuzugreifen:

```
(*zeigername).strukturvariable
```

Übersichtlicher ist jedoch diese Schreibweise:

```
zeigername -> strukturvariable
```

Beides ist gleichbedeutend.

9. 7. Strukturen als Funktionsargumente

Auch hier gilt: Du kannst Strukturen wie normale Datentypen behandeln. Das heißt, daß Du Variablen der Struktur als Parameter an eine Funktion übergeben kannst. Darüber hinaus kannst Du Zeiger, auf Variablen der Struktur übergeben. Welchen Weg Du wählst, hängt von der Aufgabenstellung ab.

9. 8. Aufgaben

1. Schreib ein Programm, in welchem eine Struktur definiert ist, welche den Namen, Vornamen, das Geburtsdatum und 5 Noten eines Schülers speichern kann. Darüber hinaus soll ein Array definiert werden, welches 15 Schüler eine Schulklasse speichern kann. Die Daten und Noten eines jeden Schülers sollen eingegeben werden. Danach kann der Benutzer entscheiden, ob er alle Daten oder alle Noten der Schüler als Tabelle ausgegeben haben möchte.

Lösungsvorschlag:

lsg_9_1.c **1679 Byte** **04. 06. 2001**

2. Schreib ein Programm, welches eine Personenkartei einer Hochschule erstellt. Darin wird zwischen Lehrkörper und Studenten unterschieden. Beide gemeinsam haben die Daten Name, Vorname, Geburtsdatum. Die Lehrenden haben eine Personalnummer. Die Studenten haben eine Immatrikulationsnummer. Die Daten der Personen sollen eingegeben werden. Dann kann der Benutzer wieder entscheiden, ob er die Daten der Studenten oder der Lehrer als Tabelle ausgegeben haben möchte. Das Programm soll 10 Personen eingeben lassen.

Lösungsvorschlag:

lsg_9_2.c **2012 Byte** **04. 06. 2001**

3. Schreib ein Programm, welches eine Bibliotheksdatenbank erstellt. Eingegeben werden sollen die ISBN, der Buchtitel, der Autor, der Verlag und das Erscheinungsjahr. Die Datensätze sollen in einer Datei gespeichert werden. Es soll die Möglichkeit bestehen, anhand der ISBN die Daten des Buchtitels herauszusuchen. Doppelte ISBN dürfen nicht vorkommen. Entwirf dazu ein Menü, welches den Benutzer entscheiden läßt, ob er ein Buch suchen oder ein Buch eingeben möchte.

Lösungvorschlag:

lsg_9_3.c **2480 Byte** **04. 06. 2001**

10. Nützliches zum Schluß

10. 1. Allgemeines

Das letzte Kapitel behandelt zum Schluß des Lehrgangs verschiedene Themen, die Dir in Deiner zukünftigen Programmierpraxis helfen können. Es befaßt sich mit Datentypen, die wir noch nicht behandelt haben. Du erfährst etwas über sogenannte Präprozessoranweisungen und Du lernst, wie man an ein Programm Werte übergeben kann. Und Du lernst Suchalgorithmen kennen, die Dir bei der Bewältigung von großen Datenmengen behilflich sein können.

10. 2. Datentypen

10. 2. 1. Ganze Zahlen

In diesem Lehrgang hatten wir bisher für die Darstellung von ganzen Zahlen den Datentyp **`Integer`** verwendet. Darüber hinaus gibt es noch zwei andere Datentypen, die ebenfalls für diese Darstellung geeignet sind.

Der Datentyp **`short`** belegt für eine ganze Zahl einen Speicherbereich von 16 Bit. Dies entspricht einem Wertebereich von -2^{15} bis $2^{15} - 1$.

Der Wertebereich des Datentyps **`Integer`** variiert je nach Rechnersystem. Er belegt einen Speicherbereich von 16 oder 32 Bit. Letzeres würde dann einem Wertebereich von -2^{31} bis $2^{31} - 1$ entsprechen.

Gleiches passiert mit dem Datentyp **`long`**. Entweder belegt er 32 Bit oder 64 Bit.

Um herauszufinden, wie groß der jeweilige Datentyp auf Deinem Rechner ist, kannst Du Dir ein einfaches Programm schreiben, welches Dir die Größe des Datentyps ausgibt. Dafür mußt Du nur die Funktion `sizeof()` benutzen, die Du schon im Kapitel 6.6. kennengelernt hast. Sie gibt Dir die Größe des Datentyps in Byte zurück.

Wenn Du vor dem jeweiligen Datentyp das Schlüsselwort **`unsigned`** setzt, bedeutet dies, daß Du keine negativen Zahlen benutzt. Dies hat den Vorteil, daß Du damit den positiven Wertebereich erweiterst.

Um noch einmal die gesamten ganzzahligen Datentypen und deren Benutzung zu verdeutlichen, habe ich für Dich die folgende Tabelle zusammengestellt:

Datentyp	Benutzung	Ausdruck	Wertebereich
short	short variablenname;	printf("%i", variable);	-2^{15} - (2^{15}-1)
Integer	int variablenname;	printf("%i", variable);	-2^{15} - (2^{15}-1) - 16 Bit -2^{31} - (2^{31}-1) - 32 Bit
long	long variablenname;	printf("%i", variable);	-2^{31} - (2^{31}-1) - 32 Bit -2^{63} - (2^{63}-1) - 64 Bit
unsigned short	unsigned short var;	printf("%u", var);	0 - (2^{16}-1)
unsigned int	unsigned int var;	printf("%u", var);	0 - (2^{16}-1) - 16 Bit 0 - (2^{32}-1) - 32 Bit
unsigned long	unsigned long var;	printf("%u", var);	0 - (2^{32}-1) - 32 Bit 0 - (2^{64}-1) - 64 Bit

10. 2. Datentypen

10. 2. 2. Reelle Zahlen

Bei den reellen Zahlen haben wir bisher den Datentyp **`float`** kennengelernt. Darüber hinaus gibt es noch 2 andere Datentypen.

Der Datentyp **`float`** belegt für eine reelle Zahl einen Speicherbereich von 32 Bit. Damit kann man eine Zahl mit 7 Stellen nach dem Komma speichern.

Der Datentyp **`double`** belegt einen Speicherbereich von 64 Bit. Dies entspricht einer Zahl von maximal 15 Stellen nach dem Komma. Du erreichst also eine höhere Genauigkeit.

Größer ist noch der Datentyp **`long double`**. Er belegt 80 Bit und erreicht damit eine Genauigkeit von 19 Stellen.

Beachte bitte, daß die Anzahl der Stellen nach dem Komma und die Größe des belegten Speicherbereichs wieder vom jeweiligen Rechnersystem abhängen.

Um das ganze noch einmal zusammenzufassen, hier wieder eine Tabelle:

Datentyp	Ausdruck	Wertebereich	Genauigkeit
float	printf("%f", var);	+/-3.4*10^{-38} ... +/-3.4*10^{38}	7
double	printf("%lf", var);	+/-1.7*10^{-308} ... +/-1.7*10^{308}	15
long double	printf("%lf", var);	+/-3.4*10^{-4932} ... +/-3.4*10^{4932}	19

10. 3. Präprozessoranweisungen, die das Leben erleichtern

10. 3. 1. Allgemeines

Bevor der Compiler den Quellcode übersetzt, überprüft ein so genannter Präprozessor den Quellcode nach an Ihn gerichtete Anweisungen. Der Präprozessor ist ein Teil des Compilers und steuert den Übersetzungsvorgang.

Präprozessoranweisungen werden folgendermaßen definiert:

```
#präprozessoranweisung
```

Achtung ! Die Präprozessoranweisungen enden nicht mit einem Semikolon. Es gibt ansonsten eine Fehlermeldung !

10. 3. 2. #include

Die erste Präprozessoranweisung, die Dir von Anfang an bekannt ist, ist die **`#include`** - Anweisung. **`#include`** weist den Präprozessor an, den Quellcode der angegebenen Datei mit zu berücksichtigen.

Es gibt 2 Arten von #include - Anweisungen:

```
#include <dateiname>
```

Diese Art der Anweisung ist für sogenannte Standard - Headerdateien. Sie enden mit dem Suffix ".h". Das sind vordefinierte Dateien, von denen Du einige bereits kennen solltest. Für jedes Programm hatten wir beispielsweise die Datei **`stdio.h`** eingebunden, um die Funktionen `printf()` und `scanf()` zu benutzen.

```
#include "dateiname"
```

Diese Art der Anweisung ist für eigenen Quellcode. Du kannst eigene Headerdateien einbinden, welche Du mit dem Suffix ".h" speicherst. Außerdem kannst Du jede weitere Datei einbinden, die C - Code enthält. Dies ist von Vorteil, wenn Du C - Code hast, welcher in mehreren Programmen benutzt werden kann. Dann solltest Du die Algorithmen in einer eigenen Datei speichern, anstatt ihn immer wieder neu zu schreiben.

Der Unterschied zwischen den beiden Anweisungen ist, daß die erste Anweisung im sogenannten Include - Verzeichnis sucht. Das ist ein Verzeichnis, welches sich im Programm Deines Compilers befindet. Die zweite Anweisung sucht im Verzeichnis Deines Programms. Willst Du auch Quellcode einbinden, welcher sich außerhalb Deines Programmverzeichnisses befindet, so solltest Du die Pfade relativ zur Datei angeben.

10. 3. 3. #define

#define ist eine Präprozessoranweisung, mit der Du so genannte Makros definieren kannst. Ein Makro ist eine Zeichenkette, die anstelle einer anderen Zeichenkette eingesetzt wird.

Ein Beispiel:

Die Zahl **PI** ist für viele schwer zu merken. Nicht alle haben sie sofort im Kopf. (Ich auch nicht.) Wenn Du die Zahl aber häufig in Deinem Programmcode benutzen mußt, so wäre es doch von Vorteil, wenn Du dabei nur **PI** hinschreiben müßtest und sich die Sache damit erledigt hätte. Da hilft Dir die folgende Makrodefinition:

```
#define PI 3.14159265358979323846264338327 95
```

Wenn Du nun im Programm eine Variable mit **PI** vergleichen möchtest, so mußt Du nur schreiben:

```
if (variable == PI)
```

Der Compiler versteht aufgrund der anfänglichen Definition, was mit der Zeichenkette **PI** gemeint ist, nämlich das **PI** der Zahl 3.1415.... entspricht. Das Ganze kannst Du natürlich auch auf Zeichenketten ausweiten.

10. 3. 4. #ifdef, #ifndef und #else

Die Anweisungen **#ifdef** und **#ifndef** dienen in der Hauptsache dazu, um Quellcode einzublenden oder auszublenden.

Wenn Du beispielsweise einen Fehler in Deinem Programm vermutest, so kann es Dir bei der Suche helfen, den Inhalt von bestimmten Variablen auszugeben. Wenn Du den Fehler gefunden hast, so brauchst Du die Werte nicht mehr auszugeben. Dies würde bei einem großen Programm aber bedeuten, daß Du eventuell sehr viele Anweisungen wieder löschen müßtest. Da würde es helfen, wenn Du den Quellcode schnell ein- und ausblenden könntest.

Um so etwas zu können, mußt Du vorher eine sogenannte Marke definieren. Dies geht mit der Präprozessoranweisung #define. Sie könnte beispielsweise so aussehen:

```
#define Fehler
```

Nun brauchst Du an den bestimmten Stellen nur abfragen, ob diese Marke vorhanden ist:

```
#ifdef Fehler
   printf("%i", var);
#endif
```

Wenn also die Marke **Fehler** definiert ist, so wird die `printf` - Anweisung eingebunden. Ansonsten, wird der Quellcode bis zum **#endif** übergangen. Wie Du Dir sicher schon denken konntest, wird die `#ifdef` - Anweisung mit `#endif` beendet.

Wenn Du also nun den Fehler gefunden hast, so kannst Du die Zeile **`#define Fehler`** auskommentieren. Die `printf` - Anweisung wird dann nicht mehr berücksichtigt.

Neben `#ifdef` existieren noch 2 weitere Präprozessoranweisungen:

`#ifndef` wird benutzt, wenn Du Quellcode einfügen willst, wenn eine Marke nicht definiert ist.

`#else` kannst Du wie eine `else` - Anweisung benutzen. Um darin noch einmal eine `#ifdef` - Anweisung unterzubringen, mußt Du eine neue Zeile beginnen, da Präprozessoranweisungen nicht mit einem Semikolon enden.

Achtung ! Du solltest solche Präprozessoranweisungen sparsam benutzen, da Du Dich ansonsten bald nicht mehr in Deinem eigenen Quellcode zurechtfindest !

10. 4. Werte an ein Programm übergeben

Bisher hatten wir das Programm gestartet um dann einige Werte vom Programm abfragen zu lassen. Du lernst in diesem Teil die Möglichkeit kennen, beim Programmstart schon Werte an das Programm zu übergeben.

Die Funktion *`main()`* hatten wir vorher immer so definiert:

```
void main()
```

Willst Du an ein Programm gleich Werte übergeben, so definierst Du die Funktion folgendermaßen:

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

Um nun das ganze zu verdeutlichen gebe ich Dir folgendes Beispiel:

Du schreibst ein Programm `prog`. Um an das Programm Werte zu übergeben, rufst Du es folgendermaßen auf:

```
prog f1 2001 ms wc
```

Um in der Funktion *`main()`* auf die 4 Parameter zugreifen zu können, brauchst Du nur auf den 2. Parameter **`argv[]`** zugreifen. Er beinhaltet:

```
argv[0] = "prog"
argv[1] = "f1"
argv[2] = "2001"
argv[3] = "ms"
argv[4] = "wc"
argv[5] = "" (Stringende)
```

Der 1. Parameter **`argc`** gibt an, wie viele Parameter übergeben worden sind. In diesem Fall ist die Anzahl 5, da der Programmname ebenfalls gezählt wird. Um den 1. Parameter auszuwerten, mußt Du also auf das 2. Feldelement zugreifen - in diesem Fall `argv[1]`.

Was Du mit diesen übergebenen Parametern anstellst, ist Deine Sache. In vielen Fällen brauchst Du dann aber nicht andauernd den Benutzer um eine Eingabe bitten.

Achtung ! Wie Du sicher gesehen hast, werden übergebene Zahlen auch als Buchstaben angesehen. Du mußt sie also eventuell in Zahlen umwandeln.

10. 5. Suchalgorithmen

10. 5. 1. Allgemeines

Wenn Du große Datensätze sortieren mußt, so ist es sinnvoll, einen schnellen und effektiven Suchalgorithmus zu verwenden. Die beiden Suchalgorithmen, die ich Dir in diesem Kapitel vorstelle, sind nicht die schnellsten, dafür aber sehr einfach zu verstehen.

10. 5. 2. Selection Sort

Dies ist wohl der einfachste Suchalgorithmus überhaupt. Er sucht immer den niedrigsten (oder höchsten) Wert aus der Folge und vertauscht diesen dann mit dem jeweiligen Element.

Hier die genaue Abfolge des Algorithmus für aufsteigende Folge:

1. Setze den Anfangszeiger auf das 1. Element.
2. Durchsuche die Folge vom Anfangszeiger bis zum Ende nach dem kleinsten Element und merke Dir die Position.
3. Vertausche das Element vom Anfangszeiger mit dem gefundenen Element.
4. Setze den Anfangszeiger auf die nächste Position.
5. Wiederhole das ganze solange, bis der Anfangszeiger am vorletzten Element der Folge angekommen ist. Die Folge ist dann sortiert.

Zur Verdeutlichung, hier noch eine Grafik:

Unsortierte Folge:

30	99	53	27	45

Nach 1. Durchlauf:

27	99	53	30	45

Nach 2. Durchlauf:

27	30	53	99	45

Sortiert:

27	30	45	53	99

10. 5. 3. Bubble Sort

Dieser Algorithmus tauscht ebenfalls die einzelnen Elemente. Allerdings wird nicht jedes Element mit der gesamten Folge verglichen, sondern nur benachbarte Elemente miteinander .
Hier die genaue Abfolge des Algorithmus für aufsteigende Folge:

1. Setze den Anfangszeiger auf das 1. Element und die Tauschkennung auf 0.
2. Vergleiche das 1. Element und das 2. Element der Folge.
3. Falls das 2. Element kleiner als das 1. ist, vertausche die beiden Elemente und setze die Tauschkennung auf 1.
4. Wiederhole die Schritte 2. und 3. für alle weiteren Elemente der Folge.
5. Wiederhole die Schritte 1. bis 4. bis die Tauschkennung 0 bleibt oder (Anzahl der Elemente der Folge) - 1 Durchläufe erreicht sind.

Unsortierte Folge:

30	99	53	27	45

1. Durchlauf:

30	99	53	27	45
30	53	99	27	45
30	53	27	99	45
30	53	27	45	99

Nach 2. Durchlauf:

30	27	45	53	99

Sortiert:

27	30	45	53	99

Wie Du erkennen kannst, schiebt sich die größte Zahl in jedem Durchlauf nach hinten. Nach jedem Durchlauf kannst Du dann also einen Schritt einsparen, da die größte Zahl bereits ganz hinten ist.