Grundlagen

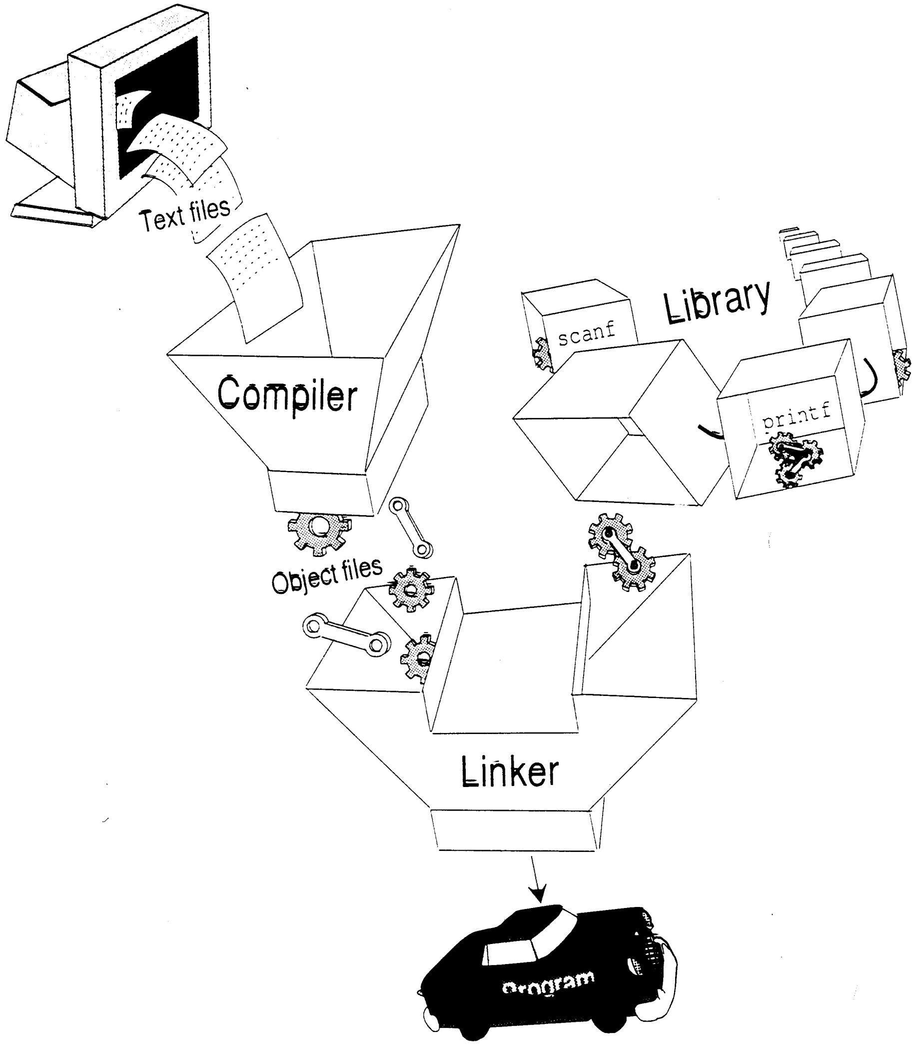
eine ausreichend hohe Ebene der Technologie ist von Zauberei nicht zu unterscheiden
–*Arthur C. Clarke* – (geklaut aus Practical C Programming)

Aufbau eines C-Programms

Ein C-Programm besteht aus einer *Ansammelung von Funktionen*, die *unabhängig voneinander* sind und *nicht verschachtelt* sein können.

```
/************************************************/
/** Kommentare: was macht es, wer schrieb es  *******/
/************************************************/
Deklarationsteil
int main(void)
{
Befehlsteil
return 0;
}
```

```
/***************************************************/
/** ein erstes kleines C Programm   **************/
/***************************************************/
#include<stdio.h>
void calc (void)
{
        printf("Hey jetzt bin ich in der calc Funktion\n");
}
int main(void)
{
        printf("Hey jetzt bin ich in der main Funktion\n");
        calc();
        printf("Hey jetzt bin ich in der main Funktion\n");
        return 0;
}
```

Metamorphose

Den gesamten Prozeß der Compilierung und Linkens kann mit Hilfe des Unix Befehl `make` gesteuert werden. Vorläufig reicht es aus `make otto` auszuführen, um ein Quellprogramm mit Namen `otto.c` in ein ausführbares Programm mit Namen `otto` zu übersetzen.

Variablen und Konstanten

C unterscheidet zwischen verschiedenen Arten der Repräsentation von Daten - somit gibt es mehrere Datentypen.

Datentypen:

```
int                     ganze Zahlen
float           Gleitkommazahlen
char            Zeichen
```

Variablen werden alle am Anfang eines *Blocks* nach der öffnenden Klammer vor den Befehlen deklariert.
Beispiel eines Progammblocks zum iterativen Berechnen der Zahl *pi*:

```
{       float pi;
        int   interationen = 100 ;
        char    antwort;
  ⋮
        pi                      = 3.1415;
        antwort = 'n' ;
  ⋮
}
```

Felder

Felder werden in C mit Hilfe der Klammeroperatoren [und] definiert. Die Deklaration eines Feldes von 10 ganzen Zahlen würde wie folgt ausssehen:
`int feld[10]`
Diese Deklaration definiert 10 integer Variablen
`feld[0], feld[1], ..., feld[9]`.
Felddeklarationen für die anderen Datentypen sind analog.

Ein- und Ausgabe

C besitzt keine *eingebauten* Konstrukte zur Ein- Ausgabe. Es gibt aber hierfür genormte Standardfunktionen. Diese sind in den Standardbibliotheken zusammengefaßt.
Wenn man diese Bibliotheksfunktionen nutzen will, muß man die entsprechenden *Headerfiles* in sein eigenes Programm einbinden.
Zwei mächtige Funktionen zur Ein- und Ausgabe sind `printf()` und `scanf()`. Um sie zu benutzen muß man das *Headerfile* `stdio.h` einbinden (includen).
Die `printf()` Funktion ist dazu vorgesehen die *Werte* von Variablen auszugeben. Sie hat folgendes Aussehen:
`printf("`*format string*`",`*wert1*,*wert2*,...);

Eine mögliche Ausgabeanweisung könnte wie folgt aussehen:

```
#include<stdio.h>
printf("%d Iterationen, Wert %f ",iter,pi);
```

Die Umwandlungszeichen für die vorgestellten Datentypen sind:

%d ganze Zahl (int)

%f Gleitkommazahl (float)

%c Zeichen (char)

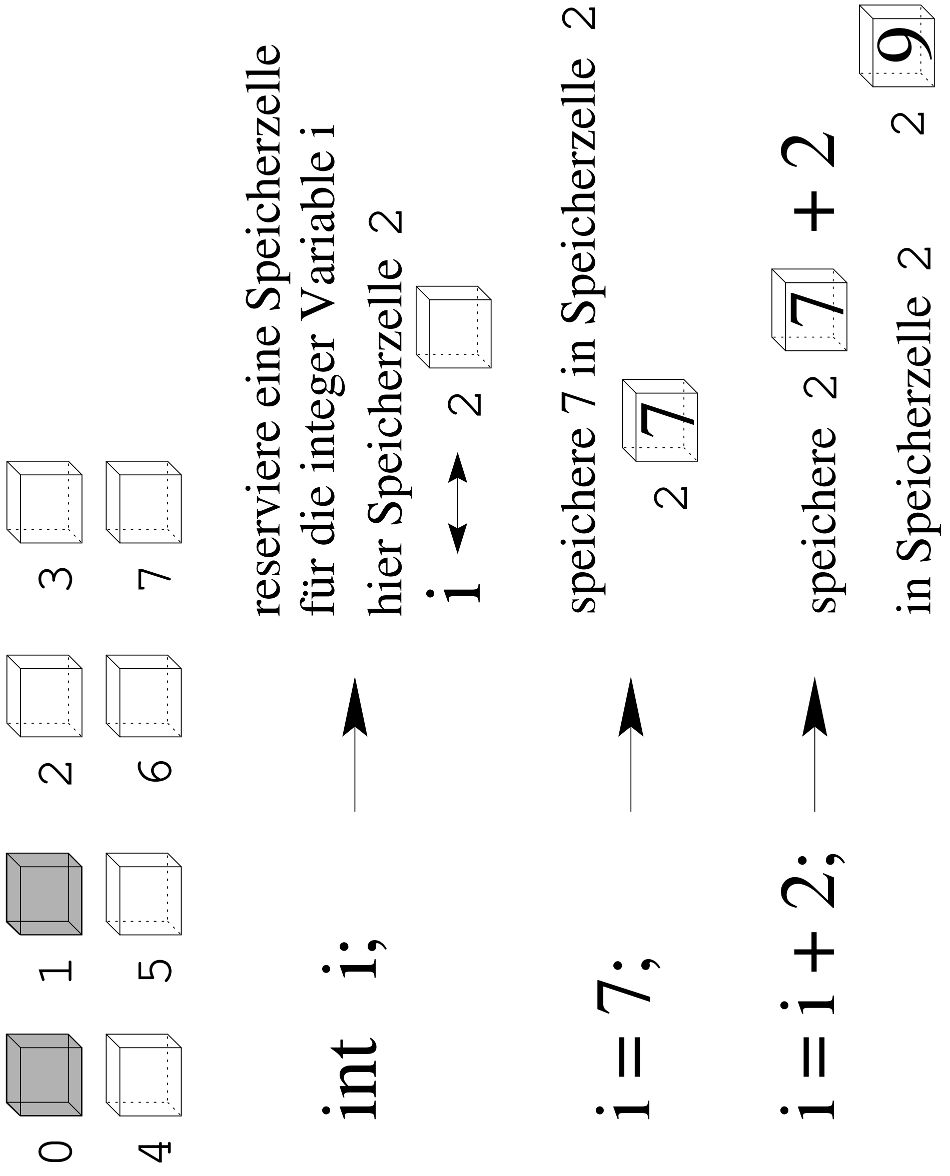
Die `scanf()` Funktion ist dazu vorgesehen *Werte* einzulesen und sie den angegebenen Variablen zuzuweisen. Sie hat folgendes Aussehen:

`scanf("`*format string*`",&`*var1*,`&`*var2*,...);

Eine mögliche Eingabeanweisung könnte wie folgt aussehen:

```
#include<stdio.h>
scanf("%d%f",&iter,&pi);
```

Das & Zeichen muß vor den Variablennamen gestellt werden.



Kontrollstrukturen, erste Schritte

while

Eine der drei Schleifenarten in C ist die `while`–Schleife. Sie besteht aus dem Schlüsselwort `while`, gefolgt von einem zu testenden Ausdruck und einem auszuführenden Befehl, der solange ausgeführt wird, wie der zu testende Ausdruck wahr ist.

if

Die `if`–Anweisung besteht aus dem Schlüsselwort `if`, einem zu testenden Ausdruck und einem Befehl oder Block von Befehlen, der dann ausgeführt wird, wenn der zu testende Ausdruck wahr ist.

```
int i = 0, grenzwert=0;
printf("bitte geben Sie den Grenzwert ein");
scanf("%d", &grenzwert);
printf("\n");
while( i <= grenzwert  )
  {
    if ( primzahl(i) ) printf("%d ist eine Primzahl\n",i);
    i = i+1;
  }
```

Grundlegendes zur Programmstruktur

- Whitespace = Zwischenräume zwischen lexikalischen Einheiten (Tokens): ␣ (Space), `\t` (Tabulator), `\n` (Newline). Whitespace hat außer dem Trenneffekt keine Relevanz (Ausnahme in Stringkonstanten).
- Schlüsselworte/Keywords = Reservierte Zeichenkette des Sprachkerns: `if, else, while, do, for....` Solche sind als Bezeichner nicht erlaubt.
- Bezeichner/Identifier = Zeichenketten der folgenden Konstruktionsregel: `[a-zA-Z_][a-zA-Z_0-9]*`
- Kommentare = beliebiger in `/*` und `*/` geklammerter Text. Kommentare werden wie Whitespace vom Compiler ignoriert.
- Beispiele
  1. `""` leerer String
  2. `" a = b + c; "` keine Anweisung

3. `" /* kein Kommentar */ "`
4. `" dies \" sind Gänsefüßchen im String "`
5. `/* "kein String" */`
6. ```
   "kein
   string"
   ```
7. ```
   "doch \
   ein string"
   ```

Der (Persönliche) Stil

Einrücken von neuen Programmblöcken

```
while (!found) {
        printf("Noch nichts gefunden\n");
        suche_weiter();
}
```

oder

```
while (!found)
{
        printf("Noch nichts gefunden\n");
        suche_weiter();
}
```

Zerlegen der Aufgabenstellung in seperate Funktionen

Kommentare – wo nötig

Sinnvolle Variablennamen

Keine riesig langen Funktionen

Keine riesig langen Ausdrücke

Zusammenfassung

- Eine Funktion ist die Basiseinheit eines C Programms. Die Programmausführung beginnt immer mit der `main()`–Funktion.
- Variablen werden am Anfang des Funktionsblocks deklariert. Die Deklaration besteht aus dem Datentyp, dem Namen der Variablen und einem Strichpunkt.
- Ein Programm wird in einem Texteditor als *Quellcode* geschrieben, durch den Compiler in *Objectcode* übersetzt und durch den Linker mit Standardbibiliotheksfunktionen zusammengebunden.
- Mit den Funktionen `scanf()` und `printf()` werden grundlegende Ein– und Ausgabeoperationen durchgeführt.
- Die `while`–Schleife wird dazu benutzt Befehlsblöcke mehrfach hintereinander auszuführen.
- Die `if`–Anweisung dient dazu, Befehle in Abhängigkeit des Wahrheitswertes von Ausdrücken auszuführen.
- Zuweisungen werden durch den Variablennamen, den Zuweisungsoperator = und einen neuen Wert realisiert. (z.B. $i = 4$; oder $i = i * 2$;)

Funktionen

Funktionen sind die „Bausteine“, aus denen ein C-Programm zusammengesetzt ist. Funktionen haben i.a. Argumente, führen damit bestimmte Aufgabe aus und geben einen Rückgabewert zurück.
Soll die Funktion keinen Rückgabewert zurückliefern, deklariert man *den Rückgabewert* als `void`. Dann macht es auch keinen Sinn das `return` Konstrukt in der Funktion zu benutzen. Soll die Funktion keine Argumente erhalten, schreibt man `void` in die Argumentenliste.

Wie sieht eine Funktion aus!

```
returntyp funktionsname ( argumenttyp argumentname ,... )
{
    Deklaration weiterer lokaler Variablen
    Befehlsfolge terminiert mit   return Rückgabewert
}
```

Wie wird eine Funktion aufgerufen!

Beim Aufruf einer Funktion liefert diese einen Ausdruck/Wert zurück, der dem Typ entspricht, der bei der Funktionsdefinition zugewiesen wurde. Angenommen eine Funktion `f` sei so deklariert, daß sie den Typ `int` zurück liefert und als Argumente zwei `float` Werte erhält, d.h.

```
int f ( float wert1, float wert2 )
{
     ⋮
}
```

Außerdem sei in der aufrufenden Funktion eine `int` Variable `berechnung` deklariert worden. Dann kann man der Varablen `berechnung` den Wert der Berechnung der Funktion `f(3,4)` durch `berechnung=f(3,4)` zuweisen.
Wird eine Funktion aufgerufen bevor sie deklariert wurde, muß vorher im Quelltext ein Funktionsprototyp der Funktion zu finden sein. Dieser beinhaltet den Rückgabewert, den Name der Funktion und die Typen der Argumente der Funktion. Man kann auch dort auch Argumentnamen mit angeben. Somit kann man als Funktionsprototyp einfach den eigentlichen Funktionskopf nehmen.

```
returntyp funktionsname ( argumenttyp argumentname ,... )
```


Beispiele: Funktionsdeklarationen

```
/****************************************************************
*  Hier beginnt die Funktion maximum.                           *
*  Sie bekommt 2 int-Werte und gibt das Maximum davon zurueck. *
****************************************************************/
int maximum(int a, int b)
{
  if (a<b) return b;
  return a;   /* Diese Zeile wird nur ausgefuehrt, falls a>=b */
}
```

```
/*********************************************************************
*  Hier beginnt die Funktion prim.                                   *
*  Sie bekommt einen int-Wert und gibt 1 (WAHR) zurueck, wenn diese *
*  Zahl eine Primzahl ist und sonst gibt sie 0 (FALSCH) zurueck.    *
*********************************************************************/
int prim(int x)
{
  int i;
  /* Es wird versucht, einen Teiler von x zu finden. Wird keiner */
  /* gefunden, so ist die Zahl prim.  Es reicht zu testen, ob    */
  /* die Zahlen von 2 bis x/2 Teiler von x sind oder nicht.      */
  i=2;
  while(i <= x/2)
  {
    if (0 == x%i) /* x modulo i ist gleich 0, also ist i ein Teiler */
      return 0;   /* von x und deshalb x keine Primzahl.            */
    ++i;
  }
  return 1;       /* Es wurde kein Teiler gefunden; also ist x prim.*/
}
```

Beispiele: Funktionsaufrufe

```
int main (void)
{
  int x ,y;
  printf("Bitte gebe eine natuerliche Zahl ein: ");
  scanf("%d", &x);
  printf("Bitte gebe noch eine natuerliche Zahl ein: ");
  scanf("%d", &y);
  /* Ausgabe mit Aufruf der Funftion maximum */
  printf("Das Maximum von %d und %d ist %d\n", x, y, maximum(x,y));
  return 0;
}
```

```
int main (void)
{
  int i, n;
  printf("Bis zu welcher Zahl willst Du die Primzahlen wissen? ");
  scanf("%d", &n);
  i=2;     /*** 1 ist sowieso keine Primzahl ***/
  while (i<=n)
  {
  if(prim(i))
      printf("%d ist eine Primzahl.\n", i);
    ++i;
  }
  return 0;
}
```


Lokale und Globale Variablen

Lokale Variablen

Lokale Variablen, deklariert am Beginn eines Blockes, sind nur innerhalb dieses Blockes gültig. Und auch nur innerhalb dieses Blockes bekannt.

Die Variablennamen der Argumente einer Funktion sind lokalen Variablen gleichgestellt. Sie können innerhalb der Funktion ohne Einschränkung wie lokale Variablen behandelt werden, d.h. man kann ihnen also auch neue Werte zuweisen.

Globale Variablen

Variablen, die außerhalb jeder Funktion deklariert sind, werden globale Variablen. Sie sind allen Funktionen bekannt und können von allen Funktionen manipuliert werden, sofern nicht lokale Variablen diese überdecken, d.h. die globale und lokale Variable denselben Namen haben. In diesem Falle hat die lokale Variable Vorrang vor der globalen.

Wann verwendet man welche Variablen

Sind Variablen nur innerhalb einer Funktion bzw. eines Blocks bekannt, so läßt sich leicht feststellen, wo diese Variablen manipuliert werden. Globale Variablen hingegen können, da sie allen Funktionen bekannt sind, an ganz verschiedenen Stellen im Programm verändert werden. Dies kann die Klarheit und Lesbarkeit eines Quelltextes beeinträchtigt.

Andererseits kann es sinnvoll sein, bestimmte *Standardwerte* allen Funktionen zugänglich zu machen. Würde man sich hier auf lokale Variablen beschränken, müßte man jeder Funktion diese *Standardwerte* als Funktionsargumente mitgeben - was die Lesbarkeit/Klarheit eines Programmes ebenso stark beeinträchtigen kann.

Ein weiterer Unterschied zwischen lokalen und globalen Variablen ist außerdem, daß lokale Variablen bei jedem Aufruf der Funktion neu kreiert werden und beim Verlassen wieder freigegeben werden - im Gegensatz zu globalen Variablen, die einmal zu Beginn kreiert werden und während des ganzen Programmes *leben*. Dies kann bei oft aufgerufenen Funktionen eine wichtige Rolle spielen. (Wie wir später aber noch sehen werden, kann diese Eigenschaft auch bei lokalen Variablen erzwungen werden.)

lokales a

globales a

globales b

```
#include<stdio.h>
int a=10;
int b=5;

int plus_1 (int a)
{
      a=a+1;
      return a;
}

int b_minus_1 (void)
{
      b=b-1;
      return b;
}

int main(void)
{
      int c;
      c=plus_1(a);
      c=b_minus_1();
      printf("a=%d und b=%d und c=%d\n",a,b,c);
      return 0;
}
```


Datentypen und Ausdrücke

Wir beschäftigen uns nun etwas näher mit den fundamentalen Datentypen in C. In der Tat gibt es nur zwei, nämlich *ganze Zahlen* und *Fließkommazahlen*.

Integers

Es gibt verschiedene Arten von Integer–Zahlen:

`long int` Eine `long int` ist in der Regel 4 Byte lang und kann somit betragsmäßig Werte bis $2^{31} - 1 = 2147483647$ annehmen. Sie ist doppelt so lang wie eine `short int`.

`short int` Eine `short int` ist in der Regel 2 Byte lang und kann somit betragsmäßig Werte bis $2^{15} - 1 = 32767$ annehmen. Sie ist halb so lang wie eine `long int`.

`int` Das ist der Basisdatentyp für Integer. Abhängig von der Architektur der Maschine ist eine `int` entweder eine `short int` oder eine `long int`. Man kann diese maschinenabhängigen Werte in der Datei `limits.h` in Erfahrung bringen.

`unsigned int` Das Schlüsselwort `unsigned` kann vor jedem der drei obigen Typen stehen und bedeutet, daß das erste Bit, welches normalerweise für das Vorzeichen benutzt wird, nun noch für den Wert selbst genutzt wird, d.h. während eine `short int` Werte zwischen $-2^{15} - 1$ und $2^{15} - 1$ annehmen kann, kann eine `unsigned short int` Werte zwischen 0 und $2^{16} - 1$ annehmen.

Wie kann ich nun einer Integer–Variablen Werte zuweisen? C ist in dieser Hinsicht sehr vielseitig. Man kann die Werte in drei verschiedenen Zahlensystemen darstellen:

dezimal Wie gewohnt. Dezimale ganze Zahlen sind z.B. $3,\ -72,\ 25000$.

oktal Durch das Voranstellen einer Null. Oktale ganze Zahlen sind z.B. $043,\ -072,\ 025000$.

hexadezimal Durch das Voranstellen von $0x$. Hexadezimale ganze Zahlen sind z.B. $0x3,\ -0x72,\ -0x0$

Um die Typüberprüfung zu unterstützen, kann man auch angeben, ob die Zahl eine `long` oder `unsigned` Integer ist. Ersteres durch Nachstellen von l oder L, letzeres durch Nachstellen von u oder U. So ist z.B. $0x77bacuL$ eine unsigned, long integer, welche den hexadezimalen Wert $77bac$ hat.

Characters

Zeichen oder *character* werden in C als ganze Zahlen repräsentiert. Das Zeichen mit dem Wert 65 korrespondiert zum 65. Zeichen des auf dem System benutzten Zeichensatzes. Gebräuchliche Zeichensätze sind z.B. der EBDSI– und der ASCII–Zeichensatz. Der Character mit Wert 65 würde im ASCII–Zeichensatz einem 'A' entsprechen. Um es wieder zu betonen: in C sind Zeichen genauso dargestellt wie Integer. Anweisungen wie:

```
char a='G';
int  i=78;
int  j=a;
printf("%d\t%c\n",a,i");
```

machen also durchaus Sinn. Obiger Programmteil erzeugt als Ausgabe `71 N` . Auf der Tastatur darstellbare Zeichen werden durch das Zeichen selbst in Hochkommata eingeschlossen dargestellt (z.B. `char c='A'`).
Nicht darstellbare Zeichen, wie z.B. das Tabulatorzeichen oder das Newlinezeichen werden mit Hilfe des sogenannten *Escapezeichen* \ dargestellt. (z.B. `char tab='\9'`). Desweiteren müssen Zeichen mit spezieller Bedeutung wie z.B. \ oder ' mit Hilfe des Escapezeichens als \\ oder \' dargestellt werden. Für oft gebrauchte Zeichen, wie `newline` oder `tab` hat C bereits Symbole wie `\n` und `\t` reserviert.

Gleitkommazahlen

Es gibt zwei verschieden Gleitkommazahlen, welche sich in der Präzision voneinander unterscheiden:

`float` Dies ist der Datentyp mit sogenannter einfacher Genauigkeit. Eine `float` dient dazu, reelle Zahlen zu repräsentieren. Im allgemeinen besteht eine `float` aus einem Byte für den Exponent und drei für die eigentlichen Dezimalstellen. `floats` sind z.B. 3.565, $-.93e5$, $454.2e-2$, $45e+23$. Zu große Zahlen zuzuweisen, führt im besten Fall zum Programmabsturz, unter Umständen allerdings zu mehr oder minder erheiternden Resultaten.

`double` Wie der Name schon sagt sind `doubles` Gleitkommazahlen doppelter Genauigkeit. Sie bestehen im allgemeinen aus 8 Byte. Die zusätzlichen byte werden dazu benutzt, mehr Dezimalstellen darstellen zu können.

Zeichenketten

Zeichenketten - alias Strings - sind in C als Felder von `char`'s realisiert. Es gibt also keinen separaten Datentyp string. Das *besondere* Zeichen `'\0'` (Null) wird bei Zeichenketten dazu benutzt, das Ende eines String zu kennzeichnen. Jeder String muß mit `'\0'` abgeschlossen werden. Folgendes definiert zum Beispiel einen String:

```
int main(void)
{
        char name[4];
        name[0]='S';
        name[1]='a';
        name[2]='m';
        name[3]='\0';
        return (0);
}
```

String Konstanten werden in “ eingeschloßen. Man darf bei der Deklaration einer Zeichenfeldes (Zeichenfeld==String==Zeichenkette) auch folgendes schreiben:

```
int main(void)
{
    char name[4]="Sam";
    ⋮
```

Dies gilt jedoch ausschließlich für die Initialisierung von Strings. C erlaubt keine direkten Vergleiche von Felder. Es ist auch nicht möglich ein Feld einem anderen zuzuweisen. Als Befehl ist folgendes also nicht möglich:

```
char name[4];
name="Sam";  /* <--- nicht moeglich !!!! */
```

Dazu stellt C eine Reihe von Standardfunktionen zur Verfügung; z.B. kann man mit `scanf("%s",name)` die String Variable *name* einlesen. Sinnvoller Weise terminiert `scanf` den String gleich mit `'\0'`. Will man einem String einen anderen zuweisen, kann man `strcpy` benutzen. Soll ein String an einen anderen angefügt werden, kann man `strcat` benutzen. Zum Vergleich zweier Strings gib es die Funktion `strcmp`. Zum Ausgeben des Strings kann man `printf` mit %s benutzen.

Unbedingt zu beachten ist das folgende: `"A"` ist ein String, d.h. ein `char` Feld, und `'A'` ist ein Char.

Folgendes Programm ließt zwei Strings `vorname` und `nachname` ein und kopiert diese in einen String `name`, der dann mit `printf` ausgegeben wird. (Vorläufig) wird vorausgesetzt, daß keine Namen auftreten, die länger als 30 Zeichen sind. Bevor man die Funktionen benutzen kann, muß man die Datei string.h einbinden.

```
#include<stdio.h>   /* hier ist u.a. scanf und printf deklariert */
#include<string.h>  /* hier sind die Stringfunktionen deklariert */

int main (void)
{
        char vorname[31];  /* hier ist Platz reserviert fuer */
        char nachname[31]; /* 30 normale Zeichen und ein \0  */
        char name[61];     /* 60 normale Zeichen und ein \0  */

        printf("bitte gib den Vorname ein:");
        scanf("%30s",vorname); /* lese hoechstens 30 Zeichen ein */
        printf("bitte gib den Nachnamen ein:");
        scanf("%30s",nachname);

        strcpy(name, vorname);  /* kopiere vorname in name   */
        strcat(name, " ");      /* fuege ein Leerzeichen an  */
        strcat(name, nachname); /* fuege nachname dazu       */

        printf("Der Vorname ist %s und ",vorname);
        printf("der Nachname ist %s\n",  nachname);
        printf("Der ganze Name ist %s\n",name);

        return (0);
}
```


Konvertierungszeichen - Übersicht

Typ	Konvertierungszeichen
int	`%d`, `%i`
long	`%ld`
short	`%hd`
unsigned	`%u`
unsigned long	`%lu`
unsigned short	`%hu`
octal	`%o`
hexadecimal	`%x`, `%X`
long octal	`%lo`
float	`%f`, `%e`, `%E` , `%g`, `%G`
double	`%lf`
long double	`%Lf`

Konstanten

Es gibt drei Möglichkeiten, in C Konstanten zu definieren:

1. Das Präprozessor–Kommando `#define` (die übliche Art in Vor-ANSI C).
2. Das Schlüsselwort `const` vor einem Datentyp (erst mit ANSI C eingeführt).
3. Der Integer–Aufzählungstyp `enum`

Das Präprozessor Kommando `#define` definiert eine *symbolische* Konstante. Das Kommando `#define PI 3.1415` setzt die Zeichenfolge `PI` mit der Zeichenfolge `3.1415` gleich, d.h. der Compiler wandelt die Anweisung
`umfang = 2*PI*r;`
in einem ersten Lauf in
`umfang = 2*3.1415*r;`
um.
Ein beliebter Fehler ist es, die `#define` Anweisung wie folgt zu benutzen
`#define PI 3.1415;`
Dies würde durch den Compiler in die fehlerhafte Anweisung
`umfang = 2*3.1415;*r;`
umgewandelt werden.
Auch hier gilt: Im besten Fall resultiert aus einem solchen Versehen ein Fehler beim Compilieren. Im schlechtesten Fall läuft das Programm, liefert aber mysteriöse Ergebnisse. Aus diesem Grunde sollte man `#define` zur Deklaration numerischer Variablen vermeiden; vor allen Dingen, da es in ANSI C das `const` Kommando gibt.

Das `const`–Schlüsselwort wird einfach vor eine übliche Variablendeklaration gestellt. Die konstante Variable muß initialisiert werden und darf von diesem Zeitpunkt an nicht mehr geändert werden. Wird die Initialisierung vergessen, so erhält die Konstante einen nicht vorhersagbaren Wert. Wird versucht, die Konstante zu ändern, so gibt der Compiler eine Warnung aus. Leider wird er sich nicht weigern, die Änderung durchzuführen.
C besitzt auch einen Aufzählungstyp. Es gibt zwei prinzipielle Möglichkeiten, ihn zu benutzen.

1. `enum tiere {vogel, spinne, einhorn};`
   setzt die Zeichenfolge `vogel` gleich dem Integerwert 0, `spinne` gleich 1 usw.

2. `enum zahlen {eins=1, vier=4, acht=8};`
   weist den Zeichenfolgen explizit die angegebenen Werte zu.

Eine Variable vom Aufzählungstyp `tiere` wird wie folgt deklariert
`enum tiere meintier;`
und kann die Werte `vogel, spinne` und `einhorn` annehmen. `meintier = spinne;`
Leider besitzt C auch hier keine Typüberprüfung und Zuweisungen wie
`meintier = 4711;`
werden nicht als falsch angesehen.

Arithmetische Ausdrücke

Bislang haben wir in jedem Programm Ausdrücke benutzt. Sie spielen eine zentrale Rolle in den meisten C–Programmen. Arithmetische Ausdrücke bestehen aus einem arithmetischen *Operator* und einem oder zwei *Operanden*. Unäre Operatoren sind:
`+` und `-` (z.B. `-i, -(-4)`)
Binäre Operatoren sind:

`+` Die Addition. `4+7` hat den Wert 11.

`-` Die Subtraktion. `i-(-b)` hat den Wert 8, wenn $i = 4$ und $b = 4$ ist.

`*` Die Multiplikation. `4.5*2.0` hat den Wert 9.0.

`/` Die Division. `3.0/2.0` hat den Wert 1.5, wohingegen `3/2` den Wert 1 hat.

`%` Die Modulo–Operation. Sie liefert den Rest der ganzzahligen Division. `11/4` hat den Wert 2, `11%4` hat den Wert 3.

Logische und Vergleichsoperatoren

Vergleichsoperatoren nehmen als Argumente zwei Zahlen eines beliebigen Typs. Ist der Vergleich erfolgreich, so liefern sie als Ergebnis eine 1, ansonsten eine 0. Vergleichsoperatoren sind:

`< und >` Die Kleiner–und Größer–Relationen.
   `7<4` hat den Wert 0. `5 > (i*3 < 4)` hat immer den Wert 1.

`<= und >=` Die Kleinergleich– und Größergleich– Relationen.

`== und !=` Die Gleich– und Ungleich–Relationen.

Logische Operatoren verknüpfen Wahrheitswerte. In C steht dabei die 0 für 'falsch'. Alle Werte verschieden von Null werden als 'wahr' betrachtet. Logische Operatoren sind:

`&&` Das logische UND.
   `7>4 && 0815 <= 4711` hat den Wert 1.

`||` Das logische ODER.
   `5 < (i*3 < 4) || 2+4` hat den Wert 1.

`!` Das logische NICHT.
   `a || b && !(a && b)` hat den Wert 1, wenn genau eine der beiden Variablen `a` und `b` 'wahr' ist.

Bitoperatoren

`&` Bitweises UND.
   `38&7 (00100110 & 00000111)` hat den Wert `00000110=6`.

`|` Bitweises ODER.
   `38|7 (00100110 | 00000111)` hat den Wert `00100111=39`.

`~` Bitweises Komplement.
   `~38 (~00100110` hat den Wert `11011001=-39`.

`^` Bitweises exclusives ODER.
   `38^7 (00100110 ^ 00000111)` hat den Wert `00100001=33`.

`>> und <<` Bitweise Shiftoperatoren.
   `38>>2 (00100110>>2)` hat den Wert `00001001=9`
   `38<<1 (00100110<<1)` hat den Wert `01001100=76`

Priorität und Assoziativität

Um einen Ausdruck, der aus mehreren Operatoren besteht, richtig interpretieren zu können, muß man die Prioritäten der Operatoren kennen. Wie wird der Ausdruck `a<b || c && !d` ausgewertet? Die folgende Tabelle gibt die Operatoren in der Reihenfolge ihrer Priorität mit der Auswertungsreihenfolge an:

Operatoren	Assoziativität
`() [] -> .`	v. links n. rechts
`! ~ (bit. NOT) ++ -- + - * & (typ) sizeof`	v. rechts n. links
`* / %`	v. links n. rechts
`+ -`	v. links n. rechts
`<< >>`	v. links n. rechts
`< > <= >=`	v. links n. rechts
`== !=`	v. links n. rechts
`& (bit. AND)`	v. links n. rechts
`^ (bit. XOR)`	v. links n. rechts
`| (bit. OR)`	v. links n. rechts
`&& (log. AND)`	v. links n. rechts
`|| (log. OR)`	v. links n. rechts
`? :`	v. rechts n. links
`= += -= /= *= %= ^= |= <<= >>=`	v. rechts n. links
`,`	v. links n. rechts

Arithmetische Zuweisungen

Sehr oft werden in C Programmen *arithmetische* Zuweisungen wie z.B. `i= i+2;` ausgeführt. Für diese Arten der Anweisung gibt es Kurzformen in C, nämlich `+=,-=,*=,/=,%=`. Man kann also anstatt `i=i+3; i+=3;` schreiben.

Die gleichen Kurzformen gibt es für die binären bitweisen Operatoren `|=, &=, ^=,<<=, >>=`. Desweiteren gibt es sogenannte *increment* und *decrement* Operatoren.

`i++` liefert den Wert von `i` und erhöht *danach* den Wert um eins.

`++i` erhöht den Wert von `i` um eins und liefert ihn dann.

`--` Der Dekrementoperator arbeitet analog.

Bedingte Ausdrücke

Es gibt bedingte Ausdrücke in C, welche im Prinzip wie eine `if` Anweisung arbeiten, jedoch einen Wert liefern.

```
max = (a > b ?  a :  b);
```

überprüft den ersten Ausdruck `a > b`. Wenn er wahr ist, so hat der bedingte Ausdruck den Wert von `a`, anderenfalls den von `b`.

Im– und explizite Typumwandlung

Man kann die arithmetischen Operationen zwischen Variablen verschiedenen Typs ausführen. Dabei wird immer die Variable, welche die kleinere interne Darstellung besitzt, in das Format der Variable mit der größeren internen Darstellung umgewandelt.

```
int num=6;
float cost;
cost = num*5.0;
cost = num*5;
```

Im ersten Ausdruck wird `num` in eine float umgewandelt und mit der float `5.0` multipliziert. Das float Ergebnis wird der float Variablen `cost` zugewiesen. Im zweiten Ausdruck wird `num` mit der integer `5` multipliziert. Das integer Ergebnis wird in eine float umgewandelt und der float Variablen `cost` zugewiesen. Weist man eine float einer integer Variablen zu, so wird die float in eine integer umgewandelt. Dabei bleibt nur der ganzzahlige Anteil erhalten.

```
int num;
float pi=3.1415;
num = pi;
```

Nach der letzten Anweisung hat `num` den Wert 3.
Man kann in C auch explizit eine Typumwandlung mit Hilfe des *cast* Operators `(typ)` durchführen.

```
(int) 2.0*3.3;
```

wandelt den float Wert 6.6 in eine ganze Zahl um. Der Sinn dieses Operators wird erst später klar werden.

Zusammenfassung

- Funktionen haben i.a. einen Rückgabewert und Argumente. Wird eine Funktion vor der eigentlichen Deklaration aufgerufen, muß vorher ein Funktionsprototyp aufgetaucht sein.
- Es gibt globale und lokale Variablen. Diese unterscheiden sich in ihrem Zugriffsbereich und der Lebensspanne.
- Es gibt zwei grundlegende Datentypen in C: ganze Zahlen und Gleitkommazahlen.
- String sind `char` Felder, die mit '\0' abgeschlossen sein müssen.
- Man kann auf drei unterschiedliche Arten Konstanten deklarieren. Mit `#define`, mit `const` und mit `enum`.
- In Ausdrücken finden die eigentlichen Berechnungen eines C Programms statt. Ausdrücke werden entsprechend ihrer Priorität ausgewertet. Wenn zwei Operatoren gleicher Priorität den gleichen Operanden teilen, so wird der Ausdruck entsprechend der Assoziativitätsregeln ausgewertet.
- Zuweisungen sind Ausdrücke. Der Wert des Ausdrucks ist der der Variablen zugewiesene Wert.
- Wenn Operanden unterschiedlichen Typs in einem Ausdruck vorkommen, so kann es zu einer Typumwandlung kommen. Normalerweise wird der kleinere Datentyp in den größeren umgewandelt. Bei Zuweisungen zu kleineren Datentypen, kann es zu Informationsverlust kommen.

Kontrollstrukturen

Die `while`–Anweisung

Es gibt drei Schleifenarten in C. Die `while`–Schleife, die `do-while`–Schleife und die `for`–Schleife. Die `while`–Schleife besteht aus dem Schlüsselwort `while`, einem zu testenden Ausdruck in Klammern und einer Anweisung
`while(test) anweisung;`
Normalerweise ist der zu testende Ausdruck ein Ausdruck mit einem Vergleichsoperator wie z.B. `i<3`. Da Ausdrücke in C stets einen Wert zurückgeben, sind jedoch alle Ausdrücke erlaubt, unter anderem auch Zuweisungsausdrücke. Somit ist

```
while(i = fac(j))
{
  .....
}
```

eine zulässige Anweisung. Die Schleife wird solange ausgeführt, bis der zu testende Ausdruck, in diesem Fall die Zuweisung, den Wert 0 hat.
Ein weit verbreiteter Fehler ist das Verwechseln von Zuweisungsoperator `=` und dem Vergleichsoperator `==`. Für den Compiler sind die Anweisungen `while(i == 5) ...` und `while(i = 5) ...` beide zulässig. Jedoch wird die zweite Schleife ewig laufen.

Die `for`–Anweisung

Die `for`–Anweisung ist wesentlich flexibler als ihre Verwandten in anderen strukturierten Programmiersprachen wie z.B. Pascal. Sie besteht aus dem Schlüsselwort `for`, drei Ausdrücken in Klammern und einer Anweisung.
`for(ausdr1;ausdr2;ausdr3) anweisung;`
Der erste Ausdruck dient zur Initialisierung und wird nur einmal, am Anfang der Schleife ausgeführt. Der zweite Ausdruck ist der zu testende Ausdruck. Wenn er 'falsch' (==0) ergibt, so wird die Schleife beendet. Der dritte Ausdruck dient zur Aktualisierung von Variablen. Eine typische `while`–Schleife

```
i=0;
while(i<3)
 {
  ...
  i++;
 }
```

könnte man als `for`–Schleife wie folgt angeben

```
for(i=0; i<3; i++) ....
```

Die `for` Schleife ist sehr flexibel. Man kann auch

```
i=0;
for(;i<10;i=next_prime(i))
{
  ...
}
```

für die obige Schleife schreiben. Mit Hilfe des Kommaoperators können auch mehrere Variablen initialisiert und aktualisiert werden

```
for(i=0, pi=3.1415, f=2.4; f<10000; i++, f*=f)
 ;
printf("%d Iterationen",i);
```

Obige Schleife zählt, wie oft man `f` mit sich selbst multiplizieren muß, bis `f` größer als 10000 wird. Man beachte, daß die Anweisung in der `for` Schleife die leere Anweisung ist. Die ganze Arbeit findet im dritten Ausdruck statt.

Die `do-while`–Anweisung

Die `do-while`–Schleife ist eine kleine Abwandlung der `while`–Schleife. Während in der `while` Anweisung der zu testende Ausdruck *vor* der ersten Ausführung der Anweisung getestet wird, so wird er in der `do-while`–Anweisung erst *nach* der ersten Ausführung der Anweisung ausgeführt.

```
 int stop;
 do
 {
   int i;

   for(i=0, printf("Hallo\n"); i<3; i++)
     printf("%d --te Iteration",i);
   scanf("%d",&stop);
  }
  while(stop != 0);
```

Die `if`–Anweisung

Die `if`–Anweisung besteht aus dem Schlüsselwort `if`, einem zu testenden Ausdruck in Klammern und einer Anweisung.

`if(ausdruck) anweisung;`
In C gibt es *kein* Schlüsselwort `then`, wie z.B. in Pascal. Wenn der zu testende Ausdruck 'wahr' (!=0) ist, so wird die Anweisung nach den Klammern ausgeführt, im anderen Fall, wird das Programm hinter dieser Anweisung fortgesetzt.
Oft will man jedoch, abhängig vom Wahrheitswert des zu testenden Ausdrucks, eine von zwei möglichen Anweisungen ausführen. Dazu dient die Anweisung
`if(ausdruck) anw1; else anw2;`
Wenn der zu testende Ausdruck wahr ist, so wird die erste Anweisung ausgeführt, anderenfalls die zweite. `if-else`–Anweisungen können auch geschachtelt werden.

```
 if(a<b)
   if(a<c)
     printf("Min = %d",a);
   else
     printf("Min = %d",c);
 else
   if....
```

Die `switch`–Anweisung

Wenn man eine Fallunterscheidung zwischen vielen möglichen Fällen durchführen will, so kann man dazu die `if-else`–Anweisung verwenden. Oft ist jedoch die tiefe Schachtelung eine Quelle für Fehler und man weicht besser auf die `switch`–Anweisung aus. Die `switch`–Anweisung hat folgendes Aussehen

```
switch (ganzzahliger Ausdruck)
 {
  case zahl1:
         anweisungen;
         break;

  case zahl2:
         anweisungen;
         break;
  .....
  default:
         anweisungen;
         break;
}
```

Die `switch`–Anweisung nimmt das Ergebnis des ganzzahligen Ausdrucks und vergleicht ihn mit `zahl1, zahl2,....` Wenn der Ausdruck mit einer Zahl übereinstimmt, so werden die Anweisungen hinter dem Doppelpunkt bis zur break–Anweisung ausgeführt. Falls der Ausdruck mit keiner der Zahlen übereinstimmt, so werden die Anweisungen hinter dem Schlüsselwort default ausgeführt. Die default–Alternative kann weggelassen werden.

```
enum michael_schanze {eins=1,zwei=2,drei=3};
enum michael_schanze mynumber;
scanf("%d",&mynumber);
switch (mynumber)
  {
  case eins:
     printf("Sie haben 1 gewaehlt");
     break;
  case 2:
     printf....
  default:
     printf("Falsche Zahl !");
     break;
  }
```

Die break– und continue–Anweisungen

In der switch–Anweisung haben wir bereits die break–Anweisung kennengelernt. Sie dient dazu, vorzeitig eine Schleife oder eine switch–Anweisung zu beenden. Die continue–Anweisung dient dazu, Teile der Anweisungen während einer Iteration zu überspringen.

```
int i=0;
while( i*i != 49)
{
  i++;
  if(i == 5)
    break;
  if(i%2 == 0)
    continue;
  printf("%d\n",i);
}
```

Obiges Programm wird die Zahlen 1 und 3 ausgeben. Wenn $i = 5$ ist, stoppt die Schleife, obwohl die Schleifenbedingung noch erfüllt ist. Immer wenn i gerade ist, wird die printf()–Anweisung übersprungen. Man sollte break und continue mit Bedacht einsetzten und wenn möglich vermeiden. Eine Ausnahme bildet natürlich die break–Anweisung innerhalb der switch Anweisung.

Die goto– und label–Anweisungen

Wegen seiner engen Beziehung zu Assemblersprachen besitzt C die Möglichkeit, direkte Sprünge auszuführen. Dazu setzt man innerhalb des Quelltextes Marken, sogenannte *label*, zu denen man dann mit Hilfe der goto–Anweisung springen kann.

```
i=0;
L1: if (i > 9)
      goto L2;
    printf("Das ist die %d -te Iteration",i);
    i++;
    goto L1;
L2: ;
```

Obiges Programm simuliert eine Schleife.
Ein strukturiertes Programmieren ist vollkommen *ohne* goto's möglich. Diese Anweisung sollte deshalb nur in begründeten Ausnahmefällen benutzt werden.

Zusammenfassung

- Es gibt zwei Arten von Schleifen die `while`– oder `do--while`–Schleife und die `for`–Schleife.
- Es gibt zwei Arten bedingter Anweisungen. Die `if`– oder `if-else`–Anweisung und die `switch`–Anweisung.
- Die `break`– und `continue`–Anweisungen dienen zur expliziten Programmflußsteuerung in Schleifen. Die `break` Anweisung erzwingt den völligen Abbruch der Schleife, während die `continue` Anweisung den Rest der Schleife während einer Iteration überspringt.
- C erlaubt unbedingte Sprünge mittels der `label`– und `goto`–Anweisungen.

Zeiger

Deklaration von Zeigern

Zeiger sind eines der wichtigsten, wenn nicht das wichtigste Konzept in C. Dazu muß man die Art und Weise verstehen, wie Variablen im Hauptspeicher des Rechners gespeichert werden.

Wir wissen, daß ganze Zahlen und Gleitkommazahlen unterschiedlich viele Bytes besitzen. Zur Vereinfachung stellen wir uns nun vor, daß der Speicher aus einer Reihe durchnummerierter Kästchen besteht. In jedem Kästchen können wir einen Wert ablegen, ob es nun eine `double` oder eine `int` ist, sei dahingestellt.

Wenn wir eine Variable deklarieren, wie z.B. `int i;`, so kann man sich dies so vorstellen, daß der Compiler ein freies Kästchen sucht, sagen wir die Nummer 1234 und sich merkt, daß er immer dann dort nachschauen muß, wenn die Variable `i` benutzt wird. Wenn `i` benutzt wird, so ist dies gleichbedeutend mit der *Adresse* 1234.
Ein Zeiger ist eine Variable, welche als Wert die Adresse eines Kästchens (Speicherzelle) hat. Sie werden wie folgt deklariert:

```
int i;
int *intpointer;
```

Die Variable `intpointer` ist vom Typ "Zeiger auf `int`", d.h. vom Typ `int*`. In der obigen Deklaration könnte der `*` auch hinter `int` stehen, jedoch sollte man sich die obige Schreibweise angewöhnen. Will man nämlich zwei Zeiger auf `int` deklarieren, so ist
`int* p1,p2;`
der falsche Weg. `p2` ist dann nämlich vom Typ `int`. Richtig ist
`int *p1,*p2;` oder
`int* p1,*p2;`
Durch die Zuweisung `intpointer=1234;` *zeigt* der Zeiger `intpointer` auf die Variable `i`, d.h. in dem Kästchen, welches für die Variable `intpointer` vom Compiler reserviert wurde, steht als Wert die Zahl 1234. Der Typ von `intpointer` sagt uns, daß man diesen Wert als Adresse interpretieren muß.

Der Adressoperator &

Im allgemeinen kennt man natürlich nicht die Adresse einer Variablen, so daß wir Zuweisungen wie
`intpointer = 1234;`
selten schreiben werden. C kennt allerdings einen Operator, welcher uns als Ergebnis die Adresse einer Variablen liefert, nämlich `&`. Nun können wir einfach den Zeiger `intpointer` auf die Variable `i` zeigen lassen.

```
int i=5;
int *intpointer;
intpointer = &i;
```

Wenn `i` die Speicherzelle mit der Nummer 234 zugewiesen bekam, so hat `intpointer` nun den Wert 234.

Der Verweisoperator (indirection)

Wir wissen, daß wir mit Hilfe des Namens einer Variablen auf ihren Wert zugreifen können. Würden wir dies mit einem Zeiger machen, so erhalten wir eine Adresse. Was wir allerdings wirklich wollen, ist es, auf die Variable zuzugreifen, auf die der Zeiger zeigt. Wir wollen einen *indirekten* Zugriff auf die Variable. C besitzt für diesen Zweck den Verweisoperator `*` (indirection).

```
int i = 5;
int *ip;

ip = &i;
*ip = 8;
printf("%d\n",i);
```

Obiges Programm druckt den Wert 8 aus, da wir indirekt durch `*ip = 8` auf die Variable `i` zugegreifen und ihr den Wert 8 zuweisen.

Zeiger und Felder

In C sind Zeiger und Felder eng miteinander verwandt. Wir wissen, wie wir ein Feld (Vektor) von 10 `int`'s deklarieren:
`int a[10];`
Wir können nun mit Hilfe des Feldoperators `[ ]` auf die einzelnen Variablen zugreifen. `a[3]` ist die vierte `int` Variable des Feldes. In C gibt es allerdings keinen Feldtyp. In Wirklichkeit ist `a` ein Zeiger auf eine `int`, nämlich auf `a[0]`. Die folgenden Anweisungen sind also äquivalent

```
int a[10];
int *p1,*p2;
p1 = &a[0];
p2 = a;
```

Adressarithmetik

Die engen Beziehungen zwischen Feldern und Zeigern gehen noch weiter. Der Feldoperator `[ ]` kann durch *Adreßarithmetik* ersetzt werden.
`i = a[3];` und `i=*(a+3)`
sind äquivalente Anweisungen. Der C–Compiler sorgt dafür, daß die Adreßarithmetik abhängig vom Datentyp, korrekt ausgeführt wird. Wenn ein `int` Zeiger um eins erhöht wird, so werden 4 Byte übersprungen. Bei einem `double` Zeiger 8.
Strings werden in C als Felder von `char` dargestellt. Als Konvention enden sie immer mit dem Nullzeichen `\0`. z.B.

```
char aname[] = "Markus";
char *pname = "Markus" ;
```

weist in beiden Fällen dem `char` Zeiger `aname` bzw `pname` die Adresse der Speicherzelle zu, in der `'M'` steht. In den nächsten Speicherzellen steht `'a','r','k','u','s','\0'`. Die \0 dient dazu das Ende eines Strings zu erkennen.
Es gibt jedoch folgenden Unterschied:

```
pname: [•]──► [Markus\0]

aname: [Markus\0]
```

aname ist ein Feld; gerade groß genug, um `'a','r','k','u','s','\0'` zu fassen. Einzelne Buchstaben können verändert werden, aber `aname` wird immer auf den gleichen Punkt im Speicher zeigen.
Im Gegensatz dazu ist `pname` ein Pointer, den man auch verbiegen darf. Aber den String kann/darf man nicht verändern.

Folgende Funktion liefert die Länge eines Strings (ist eine vordefinierte Standardfunktion, deklariert in `strings.h`).

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

  for(n = 0; *s != '\0'; s++)
    n++;
  return n;
}
```

Diese Funktion ist identisch mit dem folgenden, vielleicht vertrauter aussehenden:

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

  for(n = 0; s[n] != '\0'; n++)
    ;
  return n;
}
```

Zeiger und Funktionen

Am Anfang des Kurses haben wir die beiden Standardfunktionen `printf()` und `scanf()` kennengelernt. In C werden Funktionsargumente immer als Werte *call by value* übergeben, d.h. es werden lokale Kopien der Variablen angelegt. Oft ist dies nicht ausreichend. Wenn man z.B. den Wert zweier Variablen tauschen will, so könnte man eine Funktion `swap()` wie folgt schreiben:

```
 void swap(int i, int j)
 {
   int help;
   help = i;
   i = j;
   j = help;
 }
```

Wenn wir nun folgendes Programm aufrufen:

```
main()
 {
   int a=3;
   int b=6;
   swap(a,b);
   printf("%d\n",a);
}
```

so erhalten wir als Ausgabe den Wert 3. Es ist also nichts geschehen. Mit Hilfe von Zeigern können wir das Problem umgehen und die sogenannte *call by reference* Parameterübergabe simulieren.

```
 void swap(int *i, int *j)
 {
   int help;
   help = *i;
   *i = *j;
   *j = help;
 }
```

Wenn wir nun unser Hauptprogramm wie folgt gestalten:

```
main()
 {
   int a=3;
   int b=6;
   swap(&a,&b);
   printf("\%d\n",a);
}
```

gibt es den erwarteten Wert 6 aus. C übergibt nun die Adressen von `a` und `b` mit dem *call by value* Mechanismus. Mit dem Verweisoperator kann man nun indirekt auf `a` und `b` zugreifen und ihre Werte verändern. Nun wird auch klar, warum `scanf(''format'',&var1,..)` aufgerufen wird. Es müssen die Adressen der Variablen übergeben werden, damit ihre Werte eingelesen werden können.

Speicherallokierung

Bislang haben wir Variablen immer *statisch* allokiert, d.h. wir wußten schon zum Zeitpunkt, da wir den Quellcode schrieben, welche Variablen wir benutzen werden. Diese sind auch dann dem Compiler während des Übersetzens bekannt. Deklarationen wie

```
int i;
double *j;
char s[10];
```

veranlassen den Compiler dazu, im Speicher Platz für eine `int` (=4 Byte), einen Zeiger auf `double` (=**4** Byte) und 10 Zeichen (=10 Byte) zu reservieren.

Oft ist es jedoch wünschenswert, erst während der Programmausführung Speicherplatz für Variablen *dynamisch* zu allokieren.
C hat für diese Zwecke eine Reihe von Standardfunktionen in `malloc.h` definiert. Die zwei grundlegenden sind
`void* malloc(int size)` und `int free(void*)`.

`malloc()` allokiert `size` viel Bytes und gibt als Ergebnis die Adresse des ersten Bytes zurück. Da noch kein Typ bekannt ist, ist der Rückgabetyp ein Zeiger auf `void`. Hier wird der `cast`-Operator sinnvoll. Ein Feld von n `int`'s wird wie folgt dynamisch allokiert und wieder freigegeben:

```
int *feld;
int n; scanf("%d",&n);
feld = (int*) malloc(n*sizeof(int));
.....
free(feld);
```

Wie bereits erwähnt, hängt die aktuelle Größe einer `int` oder `double` von dem jeweiligen Rechner ab. C befreit uns jedoch von der lästigen Arbeit, nachzuschlagen, wieviele Byte denn nun 10 `int`'s benötigen. Der `sizeof`–Operator nimmt uns die Arbeit ab.
Nachdem wir das Feld nicht mehr brauchen, müssen wir den benötigten Speicherplatz wieder zur weiteren Verwendung freigeben. Dies geschieht nicht automatisch. Die `free`–Funktion gibt den gesamten für das Feld allokierten Speicher wieder frei, der vorher mit `malloc` angefordert wurde.

Zusammenfassung

- Zeiger sind Variablen, welche Adressen anderer Variablen als Wert haben.
- Zeiger werden mit dem Datentypnamen und einem nachgestellten Verweisoperator `*` deklariert.
- Die Adresse einer Variablen kann mit Hilfe des Adreßoperators `&` festgestellt werden.
- Ein indirekter Zugriff auf die Variable, auf die der Zeiger zeigt ist mit Hilfe des Verweisoperators `*` (indirection) möglich.
- Adrešarithmetik kann anstatt des Feldoperators benutzt werden und umgekehrt.
- C kennt nur eine Art der Parameterübergabe, *call by value*. Sollen Werte von Parametern verändert werden, so müssen die Adressen der Parameter übergeben werden.
- Mit Hilfe der `malloc`–Funktion kann dynamisch Speicher allokiert werden. Der `sizeof`–Operator liefert als Ergebnis die Größe des Datentyps auf dem jeweiligen Rechner. Die Standardfunktion `free` gibt den dynamisch allokierten Speicherplatz wieder frei.

Speicherklassen

In C gibt es verschiedene Speicherklassen, `auto, extern, static, register`. Die einzige, welche wir bislang benutzt haben, ist die Klasse `auto`. Sie ist die standardmäßige Speicherklasse und muß deshalb nicht explizit mit dem Schlüsselwort `auto` vereinbart werden. Die folgenden Variablen Deklarationen sind also äquivalent:

```
auto int count;
int count;
```

`auto`–Variablen werden innerhalb eines Funktionsblockes deklariert. Sie sind *lokale* Variablen, die bei jedem Funktionsaufruf neu angelegt werden. Beim Verlassen des Funktionsblockes sind sie nicht mehr gültig. Man kann innerhalb von Funktionsblöcken andere Blöcke von Anweisungen verwenden, an deren Anfang wieder Variablen deklariert werden dürfen. In diesem Block sind alle Variablen der umschließenden Blöcke gültig. Verwendet man jedoch einen Variablennamen neu, so ist nur die Variable im innersten Block bekannt. z.B.

```
void main(void)
{
   int a,b=3;
   int i;
   for(i=0;i<3;i++) {
      float a=3.14;
      printf("%f\t%d\n",a,b);
   }
}
```

In der `for`–Schleife sind die `int`'s `i` und `b` bekannt und *nicht* die `int a`. Sie wird von der `float a` *überdeckt*.
Das Schlüsselwort `register` vor einer Variablendeklaration weist den Compiler an, die Variable in einem *Register* des Rechners zu halten. Normale Variablen müssen, wenn sie benutzt werden, zuerst vom Hauptspeicher in ein Register des Prozessors geladen werden. Wenn ein Prozessor viele Register besitzt und eine Variable häufig benutzt wird, ist es sinnvoll sie in einem Register zu belassen. Hat der Prozessor kein freies Register verfügbar, so wird die `register` Anweisung ignoriert und die Variable wie eine normale `auto` Variable behandelt.

```
register int i;
for(i = 0;i < 10000; i++)....
```

Da eine `register` Variable sich (eventuell) nicht im Speicher befindet, kann der Adreßoperator nicht auf sie angewendet werden. Eine Anweisung der Form:


```
int *ip = &i;
```

wäre also unzulässig.
Eine als `static` deklarierte Variable stellt in etwa das Gegenteil zu einer `auto` Variable dar. Während eine `auto` Variable bei jedem Betreten des Blocks, in dem sie definiert ist, neu erzeugt wird, und beim Verlassen desselben zerstört wird, wird für eine `static` Variable nur einmal am Programmanfang Speicher reserviert, der während aller Funktionsaufrufe erhalten bleibt.

```
void stattest(void)
{
  static int num=1;
  printf("%d\n",num);
  num++;
}

void main(void)
{
  stattest();stattest();stattest():
}
```

obiges Programm druckt die Werte 1, 2, 3 aus. Fehlt das Schlüsselwort `static`, so druckt es 1, 1, 1 aus. Wird eine `static` Variable initialisiert, so geschieht dies nur *einmal*.
Eine *externe* Variable ist eine Variable, welche außerhalb von Funktionsblöcken deklariert wird. Sie wird beim Beginn des Programms angelegt und behält ihre Gültigkeit während des gesamten Programmablaufes. Eine externe Variable spielt eine ähnliche Rolle wie eine *globale* Variable in anderen Sprachen, sie kann nicht als `auto`– oder `register`–Variable vereinbart werden.

```
int result;
int total = 2;

void main(void)
 {
  int count;
  ....
 }
```

Im obigen Beispiel sind `result` und `total` zwei externe Variablen. `total` wird explizit mit 2 initialisiert. Da alle externen Variablen initialisiert werden, wird `result` implizit mit 0 initialisiert.
Somit sind externe Variablen ähnlich zu `static`–Variablen. Sie unterscheiden sich jedoch im Geltungsbereich. Im Gegensatz zu `static`–Variablen, können externe

Variablen mehrfach *deklariert* werden. Dies geschieht durch Voranstellen des Schlüsselwortes `extern`.

```
int result;

void main(void)
{
  int count;
  extern int result;
  ....
}
```

Eine externe Variable kann nur *einmal definiert* werden, wohingegen sie mehrfach *deklariert* werden kann. Sie kann nur in der Definition initialisiert werden. Eine Anweisung der Form:
`extern int result=1;`
wäre also unzulässig. Im obigen Beispiel macht die `extern` Deklaration in `main` nicht viel Sinn, da die Variable `result` in der gleichen Datei definiert ist. Wäre `result` in einer anderen Datei als externe Variable definiert, so müßten wir sie in `main` als `extern` deklarieren, wenn wir sie benutzen wollen.

Geltungsbereiche

In C gibt es drei verschiedene Strukturen, welche im bezug auf den Geltungsbereich von Variablen relevant sind. Funktionen, Blöcke und Quellcode Dateien. Bislang haben wir Programme immer in einer Datei geschrieben. Im Normalfall besteht ein C Programm aus sehr vielen Quellcodedateien, in denen Sinneinheiten zusammengefaßt sind.

Wie bereits erwähnt sind Funktionen die Struktur, welche interne (oder lokale) Variablen von externen (oder globalen) unterscheiden. Nur interne Variablen können als `auto`– oder `register`–Variablen definiert werden. Beide, externe und interne, können als `static` deklariert werden.

Eine interne statische Variable ist nur innerhalb der Funktion gültig, in der sie definiert wurde. Sie bleibt im Gegensatz zu `auto`–Variablen während mehrerer Funktionsaufrufe bestehen.

Eine externe statische Variable ist nur innerhalb der Datei gültig, in der sie definiert wurde. Normale externe Variablen werden zwar wie statische Variablen am Anfang des Programms erzeugt, können jedoch durch eine `extern`–Deklaration in anderen Quelldateien sichtbar gemacht werden.


```
FILE 1:
void calc(void);  /* Definition */
int result;
static int num;

void main(void)
{
  int count = 10;    /* auto variable */
  extern int result; /* Definition, kann hier
                        weggelassen werden */
  result = count*2;
  calc();
}

FILE 2:
void calc(void)
{
  int num = 2;
  extern int result; /* erlaubt */
  extern int num;   /* nicht erlaubt */

  result = num/2;
}
```

Man kann auch Funktionen als `static` definieren. Sie können dann nur innerhalb der Datei, in der sie definiert sind, aufgerufen werden.
`static int calc(int);`
Man kann also sagen, daß die Definition einer Funktion oder Variablen als `static` einen gewissen Schutz bietet, da diese Objekte nicht von Funktionen außerhalb der Datei benutzt werden können.

In C ist also die Quelldatei selbst eine Struktur, welche Geltungsbereiche bestimmt. Dies ist recht ungewöhnlich und kommt in anderen Programmiersprachen nicht vor.

Will man externe Variablen auch in anderen Quelldateien bekanntmachen, so benutzt man eine `extern` Deklaration.

Programmorganisation

Der C–Präprozessor

Der C–Präprozessor dient dazu, die Compilierung des Sourcecodes zu steuern. Man kann andere Quellcodeteile *includen*, man kann abhängig von Variablen Teile des Quellcodes von der Übersetzung ausschließen und man kann symbolische Konstanten mit `#define` definieren.

#define Man kann mit diesem Kommando symbolische Konstanten definieren.
`#define PI 3.1415`
setzt die Zeichenkette `PI` mit der Zeichenkette `3.1415` gleich. `#define` läßt sogar Argumente zu. Hier ein Macro, welcher den bedingten Ausdruck benutzt, um das Maximum von zwei Zahlen zu berechnen.
`#define max(a,b) (a>b) ?  a :  b`
Man beachte, daß es sich hierbei *nicht* um eine Funktion handelt. `a` und `b` sind *keine* Variablen. Wir können den Macro wie folgt benutzen:

- `max(3,4)` wird ersetzt durch `(3>4) ?  3 :  4`
- `max(Bla,e(@)` wird ersetzt durch `(Bla>e(@) ?  Bla :  e(@`

#include Variablen– und Funktionsdeklarationen werden oft in vielen Dateien eines Programmes gebraucht. Anstatt Sie in jeder Datei neu zu deklarieren, schreibt man solche Deklarationen in sogenannte *Headerfiles*. Sie besitzen als Konvention die Endung `.h`.

#ifdef,#ifndef,#endif Diese Kommandos schließen bedingt Quellcodestücke von der Compilierung aus bzw. in die Compilierung ein.

```
#define MS_DOS
....
#ifdef MS_DOS
...
#endif
```

Diese Kommandos testen nur, ob eine symbolische Konstante definiert ist. Sie braucht wie im Beispiel keinen Wert zu haben.

#if,#elif,#endif Diese Kommandos gestatten ebenfalls bedingte Compilierung. Sie testen eine symbolische Konstante jedoch auf ihren Wert.

```
#define MYDEBUG 2
...
```



```
#if MYDEBUG > 0
printf(...
#elif MYDEBUG == 0
error(..
#endif
```

defined() Dieses Kommando liefert als Wert 1 wenn die symbolische Konstante definiert ist, 0 sonst. Sie kann z.B. in Verbindung mit dem `#if` Kommando gebraucht werden.

```
#define UNIX
...
#if defined(UNIX)
#include"unix.h"
#elif defined(DOS)
......
```

Will man den *Text* sehen, den der Präprozessor dem C-Compiler serviert - d.h. alle `#define`'s etc. aufgelöst - so kann man den Präprozessor auch gesondert arbeiten lassen. Dies geht mit dem Befehl `cpp` bzw. `gcc -E`.

Aufteilen von Programmen

Es gibt zwei Hauptgründe, C Programme auf mehrere Dateien aufzuteilen.

1. Übersichtlichkeit.
   Wenn man ein Programm schreibt, welches eine Datenbank verwaltet, so wird man etwa alle Funktionen, welche die Daten einlesen in eine Datei `input.c` schreiben. Ebenso alle Ausgabefunktionen in eine Datei `output.c`, die Sortierfunktionen in `sort.c` und die `main()` Funktion in die Datei `main.c`.
2. Zeitersparnis.
   Man kann in C Quellcodedateien getrennt übersetzen. Wenn man z.B. im obigen Beispiel schon alle Ausgabefunktionen geschrieben hat, wäre es unnötig, sie bei jedem Compilerlauf wieder mitzuübersetzen. In großen Programmen bedeutet dies eine beträchtliche Zeiterparnis. (Die Compilierungszeit ist nicht selten 10 min oder länger, die Zeit eine kleine Sinneinheit zu übersetzen liegt im Sekundenbereich).

In obigem Beispiel könnten wir zunächst alle Dateien in *Objektdateien* übersetzen.

`gcc -c input.c output.c sort.c main.c`

Dieser Befehl übersetzt nun alle Dateien in die Maschinensprache. Als Resultat erhält man die Dateien `input.o, output.o, sort.o, main.o`. Diese bilden noch kein ausführbares Programm. Vielleicht ruft die `main` Funktion ein Funktion `einlesen()` auf, welche in der Datei `input.c` steht. `main` kennt zwar die *Deklaration* dieser Funktion, jedoch nicht die *Definition*. Um ein ausführbares Programm zu bekommen müssen die obigen Objektdateien zusammengebunden (gelinkt) werden.

```
gcc -o db input.o output.o sort.o main.o
```

Der C Compiler erkennt, daß er die`.o` Dateien nicht mehr übersetzen braucht und bindet die Objektdateien lediglich zusammen. Dabei werden standardmäßig auch die benutzten Standardfunktionen aus den Standardbibliotheken dazugebunden.

```
FILE : input.h
void einlesen(char*);

FILE : input.c
void einlesen(char* name)
{ ......}
....
FILE : main.c
#include"input.h"

void main(void)
{
 char name1[10];
 ......
 einlesen(name1);
}
```

Nun hat man ein ausführbares Programm `db`. Es sei denn, der Linker meldet einen Fehler. Es ist durchaus möglich, daß alle Programmstücke korrekt compiliert werden, der Linker jedoch eine Fehlermeldung produziert. Wenn man sich z.B. in der Datei `input.c` verschreibt und
`void einlsen(char* name)`
schreibt, so wird das Programm fehlerfrei compiliert. Die `main` Funktion kennt durch `input.h` die Deklaration einer Funktion `void einlesen(char*)` und nimmt an, daß sich deren Definition in einer anderen Datei befindet.

Die Datei `input.c` wird ebenfalls fehlerfrei übersetzt. Beim Zusammenbinden sucht der Linker nun nach einer Funktion `void einlesen(char*)`. Da er sie nicht findet, bricht der Vorgang mit einer Fehlermeldung folgender Art ab.


```
FPReinlesen(char*) undefinded symbol
```

Headerfiles

Im vorigen Beispiel haben wir bereits ein *Headerfile* benutzt. Oft gibt es viele externe Variablen und Funktionen, die in vielen verschiedenen Quelldateien benutzt werden. Es ist mühselig und fehleranfällig, bei jeder Programmänderung, alle Dateien zu editieren und `extern` Deklarationen von Variablen und Funktionen zu ändern. Vielmehr schreibt man solche Deklarationen in Headerfiles und benutzt `#include`, um sie in die jeweiligen Dateien einzubinden.

Alle Standardfunktionen sind in einer Reihe vordefinierter Headerdateien `stdio.h, malloc.h, stdarg.h, math.h, ...` deklariert.
Die eigentliche Definition befindet sich bereits compiliert in den Standardbibliotheken.

Ändert man nun einen Funktionsnamen oder definiert eine neue Funktion, so braucht man nur das Headerfile zu editieren und zu ändern.

```
FILE : input.h
#ifndef INPUT_H
#define INPUT_H

void einlesen(char* name);
#endif

FILE : output.h
#ifndef OUTPUT_H
#define OUTPUT_H
.....
void ausgabe(char* name);
#endif

FILE : sort.h
#ifndef SORT_H
#define SORT_H
.....
enum which_sort { bubble, quick, merge };
void quicksort(char*,int);
......
#endif
```

```
FILE : headers.h
#ifndef HEADERS_H
#define HEADERS_H
#include"input.h"
#include"sort.h"
......
#endif

FILE : input.c
#include"input.h"
...
void einlesen(char* name)
{ ..... }
....

FILE : main.c
#include"headers.h"
#include"input.h"
#include"sort.h"
...
void main(void)
{ .....}
.....
```

Im obigen Beispiel binden wir in die Datei `main.c` die Headerfiles `headers.h` und `input.h` ein. In `headers.h` geschachtelt ist jedoch wieder `input.h`.

Damit man Headerfiles durch geschachtelte `#include` Anweisungen nicht mehrfach einbindet, benutzt man die obige Art der bedingten Compilierung. Beim ersten Einbinden von `input.h` wird das Symbol `INPUT_H` definiert. Beim zweiten Aufruf von `#include″input.h″` ist dieses Symbol bereits definiert und `input.h` wird nicht mehr eingebunden.

Diese Vorgehensweise verhindert die doppelte Deklaration von Variablen und Strukturen. Ohne die `#ifndef,#endif` Schachtelung würde der Datentyp `enum which_sort` zweimal in `main.c` eingelesen werden, was zu einem Fehler führt, da kein Typ gleichen Namens doppelt definiert werden darf.

Makefiles

Normalerweise bestehen C Programme aus einer Vielzahl einzelner Quellcode– und Headerdateien. Es ist umständlich und fehleranfällig sich zu merken, welche

Quelldateien von welchen Headerdateien abhängig sind, was wann neu compiliert werden muß und was nicht.

UNIX bietet ein hervorragendes Hilfsprogramm `make` an, welches den Programmierer bei dieser Aufgabe unterstützt. Dieser Mechanismus ist so populär unter C-Programmierern, daß auch andere Betriebssysteme wie MS-DOS oder OS/2 meist in Zusammenhang mit einem C–Compiler diesen Mechanismus anbieten. Das Kommando `make` benutzt eine Datei, welche standardmäßig `Makefile` oder `makefile` genannt wird. In ihr stehen make-Befehle der folgenden Form

```
target : abhaengige Dateien
\tab     UNIX Befehl
Leerzeile
```

Das *target* kann ein Dateiname, oder irgendein beliebiges Kürzel sein. Ein Makefile für unser Datenbankprogramm könnte wie folgt aussehen:

```
db : main.o input.o output.o sort.o
        gcc -o db main.o output.o input.o sort.o

main.o : main.c headers.h
        gcc -c main.c

sort.o : sort.c sort.h
        gcc -c sort.c

input.o : input.c input.h
        gcc -c input.c

output.o : output.c output.h
        gcc -c output.c

remove :
        rm *.o
```

Um das Programm zu compilieren schreibt man `make db` oder `make`. Wenn man ein spezielles target angibt, so wird nur dieses ausgeführt, anderenfalls geht `make` die Liste der targets sequentiell durch.

`make db` geht also zum target `db`. Es durchsucht sequentiell die Liste der abhängigen Dateien, also zuerst `main.o`, und überprüft, ob diese selbst ein target ist. In unserem Beispiel ist dies der Fall. `make` hält also hier kurz an und geht zum target `main.o`.

Hier geht es wieder zur ersten abhängigen Datei, nämlich `main.c` und überprüft, ob diese ein target ist. Keine der abhängigen Dateien von `main.o`, nämlich `main.c` und `headers.h` ist ein target. Nun überprüft `make` das Datum der Dateien. Ist eines der Daten der abhängigen Dateien jünger als das Datum des targets, so wird das UNIX Kommando, in unserem Fall also `gcc -c main.c` ausgeführt, anderenfalls nicht.

Danach geht `make` wieder zum ursprünglichen target `db` zurück und setzt die Abarbeitung der abhängigen Dateien fort. `make` ist ein sehr mächtiges Werkzeug. In den Übungen wird das Erstellen von Makefiles ausreichend eingeübt werden. Ein typisches Makefile für das `db` Programm könnte so aussehen:

```
CFLAGS = -g -O
cc = gcc
OBJECTS = main.o output.o sort.o input.o

db : $(OBJECTS)
        $(cc) -o db $(CFLAGS) $(OBJECTS)

main.o : main.c headers.h
        $(cc) -c main.c

sort.o : sort.c sort.h
        $(cc) -c sort.c

input.o : input.c input.h
        $(cc) -c input.c

output.o : output.c output.h
        $(cc) -c output.c

remove :
        rm *.o
```

In der Praxis wird das Makefile jedoch noch ein wenig anders aussehen. Der Make–Mechanismus hat viele *implizite* Regeln, wie z.B. das Übersetzen einer `.c` Datei in eine `.o` Datei oder das Übersetzen einer TEX– Datei in eine `.dvi` Datei. Bei einer solchen impliziten Abhängigkeit ist es auch nicht notwendig, die entsprechende Datei in der Liste der Abhängigkeiten aufzulisten. Somit würde ein reales Makefile für unser Datenbankprogramm wie folgt aussehen:

```
CFLAGS = -g -O
cc = gcc
OBJECTS = main.o output.o sort.o input.o
```



```
db : $(OBJECTS)
        $(cc) -o $@ $(CFLAGS) $^

main.o : headers.h

sort.o : sort.h

input.o : input.h

output.o : output.h

remove :
        rm *.o
```

Zusammenfassung

- Der C–Präprozessor steuert die Compilierung und ordnet das Zusammenbinden von Quellcodestücken.
- C Programme werden in viele kleine Quelldateien aufgeteilt, um zum einen die Übersichtlichkeit zu erhöhen, und zum anderen ein effizientes Compilieren zu ermöglichen.
- Funktionen, Variablen und Datentypen werden in Headerdateien deklariert, um sie gesammelt an einem Ort zu haben und somit die Übersichtlichkeit zu erhöhen.
- Um das mehrfache Einbinden von Headerdateien zu vermeiden, sollten sie durch Präprozessorkommandos `#ifndef` und `#endif` eingeschlossen werden.
- Der `make`–Mechanismus von UNIX ist ein mächtiges Hilfsmittel, um größere Programmprojekte zu verwalten.

`structure` und `typedef`

Die Deklaration eines Verbundes

Ein Verbund ist ein Zusammenschluß mehrerer Variablen zu einer Struktur, die als Einheit betrachtet werden soll. Will man z.B. einen Student durch seinen Namen und seine Matrikelnummer charakterisieren, so kann man einen Verbund wie folgt definieren:

```
struct {
  char name[20];
  long int matrikelnummer;
  };
```

Dies ist eine Typdeklaration. Es wurde noch keine Variable definiert. Dazu müssen wir allerdings nur einen Variablennamen angeben;

```
struct {
  char name[20];
  long int matrikelnummer;
  } paul;
```

`paul` ist nun eine Verbundvariable vom Typ
`struct { char name......` Man kann dem Verbund auch einen Namen geben. Dies geschieht durch Angabe des Namens direkt hinter dem `struct` Schlüsselwort.

```
struct student {
  char name[20];
  long int matrikelnummer;
  } paul;
```

Nun ist `paul` eine Variable vom Typ `struct student`. Dieser Name kann auch zur Definition von Variablen benutzt werden.

```
struct student {
  char name[20];
  long int matrikelnummer;
  };

int i;
struct student hans;
struct student karin;
```


Da eine Verbunddeklaration eben nur eine Deklaration ist, ist sie ein perfekter Kandidat, um sie in Headerdateien zu schreiben.

`typedef` und Verbunde

Das `typedef` Schlüsselwort erlaubt dem Programmierer, einem Typ einen anderen Namen zu geben. Folgende Anweisungen sind alle zulässig:

```
typedef int integer;
typedef char* char_pointer;
typedef struct student {
  char name[20];
  long int matrikelnummer;
  } STUDENT;
```

Wir können nun die hinteren Namen als Synomyme für die davor stehenden Typen benutzen und Variablen wie folgt definieren:

```
integer i;
char_pointer c;
STUDENT karl;
struct student peter;
```

Man beachte, daß hier keine textuelle Ersetzung wie bei `#define` stattfindet. In der typedef Anweisung sind nur gültige C Datentypen erlaubt, deren Verwendung durch den C Compiler überprüft wird.

Durch eine `typedef`–Anweisung für einen Verbund wird der Verbundname überflüssig, kann jedoch weiter benutzt werden. Im allgemeinen ist es üblich Verbundtypen in Zusammenhang mit einer typedef–Anweisung zu definieren.

Komponenten eines Verbundes

Ein Verbund besteht aus dem Zusammenschluß mehrerer Variablen. Eine Verbundvariable selbst kann in vielerlei Hinsicht wie eine Variable fungieren. Sie kann als Rückgabe– oder Parametertyp einer Funktion dienen und im Zuweisungsoperator als Argument verwendet werden. Der *Wert* einer Verbundvariablen besteht aus der Gesamtheit der Werte ihrer Komponenten. Folgende Zuweisungen sind also zulässig:

```
typedef struct {
  char name[20];
  long int matrikelnummer;
```

```
  } student;
 student s1 = {"Paul",1000000};
 student temp = s1;
```

Auf die Komponenten selbst kann man mit Hilfe des Punktoperators `.` zugreifen.

```
student s1;
s1.name = "Paul";         /* falsch */
strcpy(s1.name,"Paul");   /* richtig */
scanf("%ld",&s1.matrikelnummer);
```

Man beachte, daß keine Klammerung notwendig ist, da der Punktoperator höhere Priorität als der Adreßoperator besitzt. Also ist `&s1.matrikelnummer` gleich `&(s1.matrikelnummer)`.

Geschachtelte Verbunde

Da Verbundvariablen ganz normale Variablen sind, ist es möglich innerhalb von Verbunden als Komponente einen Verbund zu deklarieren. Die einzige Bedingung ist, daß die Schachtelung hierarchisch sein muß, d.h. kein Verbund kann als Komponente einen Verbund seines eigenen Typs haben, oder aber einen Verbund, in dem er selbst als Komponente vorkommt.

```
typedef struct {
  int vorwahl;
  int nummer;
} telefon;

typedef struct {
  char name[20];
  long int matrikelnummer;
  telefon nummer;
} student;

student s1;
strcpy(s1.name,"Paul");
s1.nummer.vorwahl = 0681;
s1.nummer.nummer  = 4711;
```

Innerhalb geschachtelter Verbunde dürfen Komponenten wie im obigen Beispiel `nummer` gleiche Namen besitzen.

Zeiger auf Verbunde

Ein Zeiger auf einen Verbund–Datentyp wird genauso wie ein Zeiger auf einen einfachen Datentyp definiert und benutzt.

```
int *inptr;
student *studptr;

student s1;
studptr = &s1;
studptr = (student*) malloc(sizeof(student));
```

Will man auf die Komponenten eines Zeigers auf einen Verbund zugreifen, so muß der Verweisoperator geklammert werden, da der Punktoperator die höhere Priorität hat.

```
printf(*studptr.name);   /* falsch */
printf((*studptr).name); /* korrekt */
```

Die zweite Anweisung ist äquivalent zu `*(studptr.name)`. Es wird also versucht, mit dem Verweisoperator auf die Komponente `name` zuzugreifen. Allerdings führt der Compiler die Zuweisung nicht aus, da `studptr` ein Zeiger auf einen Verbund ist und keine Verbundvariable.

Da der Zugriff auf Komponenten von Zeigern auf Verbunde recht häufig vorkommt, besitzt C eine abkürzende Schreibweise dafür, den *Pfeiloperator* `->`.

```
printf(studptr->name);
```

Zeiger auf Komponenten

Ein Verbund kann als Komponente einen Zeiger auf einen anderen Datentyp besitzen. Will man auf diesen zugreifen, so kann man dazu den Verweisoperator benutzen. Ein Beispiel:

```
typedef struct {
  int *vorwahl;
  int *nummer;
} telefon;

typedef struct {
  char name[20];
  long int matrikelnummer;
  telefon *nummer;
} student;
```

Um die so definierten Pointereinträge zu füllen müssen wir den notwendigen Speicher dynamisch allokieren.

```
student s1 = {"Paul", 1234567,
               (telefon*) malloc(sizeof(telefon))};
student *sptr1,*sptr2;
telefon mynum;
int tel = 4711;

sptr1 = &s1;
sptr1->nummer->vorwahl = (int*) malloc...;
*(*sptr1).nummer->vorwahl = 0681;
(*sptr1->nummer).nummer = &tel;

sptr2 = (student*) malloc(sizeof(student));
sptr2->nummer = &mynum;
```

Verbunde und Funktionen

Verbundtypen können genauso wie einfache Datentypen als Rückgabe– und Parametertyp von Funktionen dienen. Um im obigen Beipiel dynamisch Speicherplatz für die Zeiger Komponenten von Variablen vom Typ `student` und `telefon` zu allokieren und ihn wieder freizugeben, könnte man folgende Funktionen schreiben:

```
void allocate_telefon(telefon t)
{
  t.vorwahl = (int*) malloc(sizeof(int));
  t.nummer = (int*) malloc(sizeof(int));
}

void allocate_student(student s)
{
  s.nummer = (telefon*) malloc(sizeof(telefon));
  allocate_telefon(*s.nummer);
}
```



```
void main(void)
{
student s1;
  allocate_student(s1);
  strcpy(s1.name,"Paul");
  *s1->nummer->vorwahl = 0681;
  *s1->nummer->nummer  = 4711;
  .....
}
```

Zusammenfassung

- Verbunde dienen dazu, aus zusammengesetzte Datentypen zu definieren.
- Verbunde können bei der Deklaration einen Namen erhalten, üblicherweise werden Sie jedoch in einer `typedef`–Anweisung als Typ deklariert und erhalten dort einen Typnamen.
- `typedef`–Anweisungen sind Deklarationen und sollten somit in Headerdateien plaziert werden.
- Auf die Komponenten eines Verbundes wird mit dem Punktoperator `.` zugegriffen.
- Auf die Komponenten eines Zeigers auf einen Verbund wird mit dem Pfeiloperator `->` zugegriffen.
- Verbunde können geschachtelt werden, jedoch nicht rekursiv.
- Verbundtypen können als Rückgabe– und Parametertyp verwendet werden.
- Eine `union`–Variable ist *eine* Variable, welche verschiedene Typen annehmen kann. Die Typen müssen innerhalb des `union` Konstrukts deklariert werden. Auf den jeweils möglichen Typ wird mit Hilfe des Punktoperators zugegriffen.

Rekursive Strukturen

Listen

Verbunde dürfen als Komponenten nicht eine Variable vom eigenen Typ besitzen, wohl aber einen Zeiger auf einen Verbund des gleichen Typs. Wir wollen diese Art von Verbunden im folgenden *rekursive* Verbunde nennen. Der einfachste Typ ist eine *Liste*. Man kann sich eine Liste als eine Kette von Verbunden vorstellen. Jeder Verbund besitzt dabei einige Komponenten, in denen sich seine eigentlichen Daten befinden und einen Zeiger, der auf das nächste Element der Kette verweist.

```
typedef struct sp {
  student s;
  struct sp *next; /* NICHT : item *next */
} item;
```

Man beachte, daß der Verbund einen Namen `sp` bekommen hat und wir den Zeiger auf den Verbund als `struct sp *next;` definieren. Wir können *nicht* den Typnamen `item` benutzen, da dieser noch nicht bekannt ist. Da man oft einen Zeigertyp für den Verbund braucht, wird auch oft folgende Anordnung für rekursive Verbunde gebraucht:

```
typedef struct sp *item_ptr;

typedef struct sp {
  student s;
  item_ptr next;
} item;
......
item_ptr head;
head = (item*) malloc(sizeof(item));
```

Eine Liste wird immer dann gebraucht, wenn sich die Anzahl der Objekte in der Datenstruktur im Verlauf des Programms ändert. Ist die Anzahl vor der Programmausführung bekannt, so können wir statische Felder benutzen. Wird die Anzahl während des Programmablaufes bekannt, ändert sich aber nicht, so können wir dynamische Felder benutzen. Ändert sich die Anzahl, so können wir für neue Objekte dynamisch Speicher allokieren und sie in die Liste einfügen, oder den belegten Speicher freigeben und sie aus der Liste entfernen. Funktionen, die diese Aufgaben durchführen, benutzen die `malloc` und `free` Funktionen und könnten wie folgt aussehen:

```
void insert(item_ptr it, student s)
```



```
{
  item_ptr new = (item_ptr) malloc(sizeof(item));
  new->student = s;
  new->next = it->next;
  it->next = new;
}

void delete_next(item_ptr it)
{
  item_ptr tmp = it->next;
  it->next = tmp->next;
  free(tmp);
}
```

Nach dem Verlassen der Funktion `insert` wird die lokale Variable `new` zerstört. Die einzige Möglichkeit, auf unsere neu allokierte Variable zuzugreifen, ist es, den `next`–Zeiger des in der Liste davor befindlichen Verbundes zu benutzen.

Hätten wir die Anweisung

```
it->next = new;
```

vergessen, so würden wir Speicher allokieren, auf den wir nie mehr zugreifen können, und den wir somit nicht mehr freigeben können. Nach einigen Aufrufen der `insert` Funktion, wäre der Speicher des aufgebraucht. Oder aber man würde beim Durchmustern der Liste in Speicherbereiche laufen, die schon freigegeben wurden. Die Resultate sind dann nicht vorhersehbar.

```
item_ptr init(student s)
{
  item_ptr new = (item_ptr) malloc(sizeof(item));
  new->next = (item_ptr)  0;
  new->s = s;
  return new;
}
```

Deshalb sollte man darauf achten, daß bei der Allokierung eines Zeigers diesem direkt ein Wert zugewiesen wird. Ansonsten kann der sich in dem Speicher befindliche Wert fälschlicherweise als Zeiger interpretiert werden. Obige Funktion initialisiert eine Liste, und liefert einen Zeiger auf eine Element zurück, dessen `next` Zeiger den Wert 0 hat. Dieser Wert ist oft als Konstante `nil` definiert. Man kann mit Hilfe eines solchen Wertes überprüfen, ob man beim Durchmustern einer Liste ihr Ende erreicht hat.

Bäume

Der Nachteil einer linearen Liste liegt darin, daß man sie eventuell vollkommen durchsuchen muß, wenn man in ihr nach einem Element sucht. Ein *Suchbaum* ist zum Lokalisieren eines Objektes in einer Menge bessser geeignet.

Ein Suchbaum besteht aus einer Reihe rekursiver Verbunde der folgenden Art:

```
typedef struct node *item;

typdef struct node {
  ..... /* IRGENDWELCHE DATEN */
  int key;
  item leftchild;
  item rightchild;
} NODE;
```

Einen solchen Verbund nennt man *Knoten* in einem Suchbaum. Jeder Knoten hat ein linkes und rechtes *Kind*. Knoten, die keine Kinder besitzen, nennt man *Blätter*. Jeder Knoten hat einen *Schlüssel* `key`, der dazu dient, die Suche nach einem Objekt zu steuern. In allen Knoten, welche dem linken Kind folgen ist der Schlüssel kleiner als der Schlüssel des Knotens, in allen, die dem rechten Kind folgen ist er größer. Dies erlaubt eine effiziente Suche in logarithmischer Zeit, wenn der Baum balanciert ist. Die Suche beginnt bei einem ausgezeichneten Knoten, welcher *Wurzel* genannt wird. Eine Suchoperation könnte wie folgt aussehen:

```
item suche(int searchkey, item start)
{
  if(start->key == searchkey) {
    return start;
  } else {
    if(start-> key < searchkey) { /* nach rechts */
      if(start->rightchild != (item) NULL) {
        return suche(searchkey, start->rightchild);
      } else {
          printf("Nicht gefunden");
          return (item) NULL;
      }
    } else { /* nach links */
       if(start->leftchild != (item) NULL) {
        return suche(searchkey, start->leftchild);
       } else {
```



```
          printf("Nicht gefunden");
          return (item) NULL;
       }
    }
  }
}
```

Ein und Ausgabe mit und ohne Dateien

Öffnen einer Datei

Damit ein Programm auf eine Datei zugreifen kann, muß das Betriebssystem eine Verbindung zwischen dem Programm und der physikalischen Datei herstellen. Dazu dient die Funktion

```
FILE *fopen(char *name,char *modus)
```

Sie bekommt als Argumente den Namen der Datei und den *Modus*, in dem die Datei geöffnet werden soll. Es gibt drei verschiedene Modi:

r Erlaubt nur das Lesen aus der Datei. Wenn die Datei `name` nicht existiert, wird ein Fehler zurückgegeben.

w Erlaubt nur das Schreiben in die Datei. Wenn die Datei `name` nicht exisitert, wird sie neu angelegt, andernfalls wird die Bestehende überschrieben.

a Erlaubt das Anhängen von Daten an das Dateiende. Wenn die Datei `name` nicht exisitiert, wird sie neu angelegt, andernfalls wird sie nicht zerstört, sondern nur geöffnet.

Ein Beispiel für eine solche Anweisung ist

```
FILE *myfileptr;
myfileptr = fopen("myfile","r");
```

Die Funktion liefert einen Zeiger auf eine Datei zurück (*file pointer*). Er besitzt den Typ `FILE*`, der in `stdio.h` vordefiniert ist. Dieser file pointer stellt praktisch die Leitung dar, über welche das Programm mit der Datei kommuniziert. Er dient vielen Standardfunktionen als Argument.

Wenn wir die Leitung zu einer Datei nicht mehr brauchen, so können wir die Datei *schließen*. Dies geschieht mit der Funktion

```
fclose(FILE *stream)
```

Zu den eben vorgestellten drei Dateimodi gibt es noch Erweiterungen:

r+ Erlaubt Lesen und Schreiben in oder aus der Datei. Wenn die Datei nicht existiert, wird ein Fehler zurückgegeben.

w+ Prinzipiell gleich zu r+. Jedoch wenn die Datei nicht exisitert, wird sie neu angelegt, anderenfalls wird die bestehende Datei überschrieben.

a+ Erlaubt das Anhängen von Daten an das Dateiende und das Lesen. Wenn die Datei nicht exisitiert, wird sie neu angelegt, anderenfalls wird sie nicht zerstört, sondern nur geöffnet.

Alle Funktionen, welche Zeiger zurückgeben, geben als Fehlerwert den Nullzeiger `NULL` zurück. Alle Operationen, welche einen `int` Wert zurückgeben, wie z.B. `fclose` geben als Fehlerwert den in `stdio.h` vordefinierten Wert `EOF` (End Of File) zurück. (Meist ist dieser Wert -1.)

```
#include<stdio.h>
#define MAXLAENGE 50
#define ZEILENLAENGE 256

int whitespacekiller(char *);

int main(int argc, char **argv)
{
char zeile[ZEILENLAENGE];
char filename[MAXLAENGE]="teste_datei";
FILE *in_file;

if ( argc == 1) {
        printf("bitte gib einen Filenamen ein:");
        fgets(filename,sizeof(filename),stdin);
        }
else
        sprintf(filename,"%.*s",MAXLAENGE,argv[1]);

(void) whitespacekiller(filename);

in_file=fopen(filename,"r+");
if (in_file==NULL) {
        fprintf(stderr, "War wohl nixx!\n");exit(8);
        }
...
if (fclose(in_file)==EOF) {
        fprintf(stderr, "Auch noch beim close Probleme!\n");exit(8);
        }
}
```

ASCII Dateien

normale Textdateien

Die beiden Funktionen `getc` und `putc` sind speziell für das Schreiben und Lesen von `char`'s in und aus einer Datei vorgesehen.

```
void main(void)
{
  FILE *fp;
  int inchar;

  fp = fopen("myfile","r");
  /* fehlerbehandlung fopen */
  while((inchar = getc(fp)) != EOF)
    putc(inchar,stdout);
  fclose(fp);
  /* fehlerbehandlung fclose */
}
```

UNIX und C kennen nur einen Dateityp. `getc` kann also eine Datei lesen, in welche Sie vorher `float`'s geschrieben haben. Jedes Byte wird jedoch als `char` interpretiert.

Textdateien werden oft dazu benutzt Strings abzuspeichern. Strings sind Zeichenketten, welche durch das Nullzeichen \0 abgeschlossen werden. Die Funktion

```
fgets(char *feld, int max, FILE *fp)
```

liest aus der Leitung `fp` solange Zeichen in das Feld `feld`, bis es das Zeilenende–Zeichen \n liest, höchstens jedoch `max-1` viele Zeichen. Als das letzte Zeichen wird automatisch das Stringende Zeichen \0 in das Feld geschrieben. Die Funktion

```
fputs(char *feld,  FILE *fp)
```

schreibt den string `feld` über die Leitung `fp` in die angeschlossene Datei. Sie schreibt *nicht* das Nullzeichen des Strings in die Datei. Allerdings auch nicht das Zeilenende–Zeichen - es sei denn es wurde dem String mitgegeben.

Formatierte Textdateien

Formatierte Textdateien benutzen die Kommandos

```
fprintf(FILE *fp, char *format,...)
fscanf(FILE *fp, char *format,...)
```

um formatierte Daten in eine Datei zu schreiben. Die beiden obigen Funktionen arbeiten wie die bekannten Funktionen `printf()` und `scanf()`, mit dem Unterschied, das sie in Dateien schreiben und aus ihnen lesen.

Die gesamte Interpretation der aus der Datei gelesenen Bytes liegt im Formatstring von `fscanf()`. Die Anweisung

```
int i;
fscanf(fp,"%d",&i);
```

liest aus der Leitung `fp` solange Zeichen bis sie auf einen *white space* (= Leerzeichen, Tabulator oder Zeilenende) trifft. Diese Zeichenkette wird dann in eine `int` umgewandelt. ('1','3','2' wird also in 132 umgewandelt.) Wenn `fscanf` auf für eine Umwandlung ungeeignete Zeichen trifft, so stoppt es und liefert die Anzahl der erfolgreichen Umwandlungen zurück.

```
int valid;
char name[20];
int id;
float gehalt;
valid = fscanf(fp,"%s%d%f",name,&id,&gehalt);
if( (valid < 3)
  printf("Korrupte Daten\n");
else
  printf("%s %d %f\n",name,id,gehalt);
```

Damit kann man prüfen, ob ein Datensatz erfolgreich eingelesen wurde. Wurde ein Fehler erkannt, so möchte man gerne den fehlerhaften Datensatz überspringen. Nehmen wir an, obige Anweisung würde die Daten

```
peter tt 32.12
```

lesen. `peter` würde in den string `name` gelesen werden. Beim Lesen von `tt` würde die Konvertierung in eine `int` nicht gelingen. `fscanf` würde stoppen und den Wert 1 liefern. Ein erneuter Aufruf von

```
valid = fscanf(fp,"%s%d%f",name,&id,&gehalt);
```

würde nun tt an name, 32 an id und .12 an gehalt zuweisen. Das ist natürlich nicht das was man will.

Wenn man einen fehlerhaften Datensatz entdeckt, so kann man den Rest des fehlerhaften Datensatzes mit Hilfe des * Zeichens überspringen. `%[^\n]` wird alles überspringen bis es auf ein Zeilenende trifft.

```
 valid = fscanf(fp,"%s%d%f",name,&id,&gehalt);
 if(valid != 3)
   valid = fscanf(fp,"%*[^\n]");
.....
```

Mann kann aber auch etwas leichter verständlichen Code schreiben, indem man zuerst die ganze Zeile einließt (mit `fgets` und dann die Zeile mit `sscanf` analysiert.

Verbunddateien

Formatierte Textdateien haben den Vorteil, daß man sie mit einem beliebigen Texteditor verändern kann und ihr Inhalt in einer leserlichen Form ist. Ein Nachteil besteht darin, daß wir für die Zahl 123456 sechs bytes verbrauchen. Dabei könnten wir sie als eine `int` in vier Bytes darstellen. Diese direktere Art der Abspeicherung wird von den Funktionen `fread()` und `fwrite()` benutzt. Sie schreiben einfach ein Bild des Hauptspeichers in die Datei.

```
fwrite(void *adr, int groesse, int anz, FILE *fp)
fread(void *adr, int groesse, int anz, FILE *fp)
```

Beide Funktionen haben vier Argumente. Eine Adresse, die angibt von wo oder wohin Daten gelesen bzw. geschrieben werden, die Größe des Blocks (= Anzahl der Bytes), wie oft dies wiederholt werden soll und einen Dateizeiger.

```
char block[3]={'A','B','C'};
fwrite(block,3*sizeof(char),1,fp);
fwrite(block,sizeof(char),3,fp);
/* identische Anweisungen */

rewind(fp);
char *test = (char*) malloc(3*sizeof(char));
fread(test,sizeof(char),3,fp);
```

Die `rewind` Funktion 'spult' die Position des Dateizeigers wieder an den Anfang zurück. Würden wir das nicht machen, so könnte die `fread` Funktion nichts einlesen. Alternativ kann man die Datei schließen und wieder zum Lesen öffnen. Beim Öffnen wird der Zeiger automatisch an den Dateianfang gesetzt.


```
student teilnehmer[100];
...
fread(teilnehmer,sizeof(student),100,fp);
```

Man kann natürlich auch Verbundvariablen schreiben und lesen.

Aktualisieren von Daten

Bisher haben wir bis auf das Anhängen von Daten noch keine Möglichkeit gehabt, Dateien zu ändern. Außerdem konnten wir nur sequentiell auf sie zugreifen, d.h. um an den hundertsten Datensatz zu kommen mußten wir neunundneunzig andere einlesen.

Beide Probleme lassen sich mit den Funktionen `fseek` und `ftell` beheben.

```
fseek(FILE *fp, long offset, int start)
ftell(FILE *fp)
```

`fseek` kann den Dateizeiger beliebig verschieben. Sie hat als drittes Argument eine von drei vordefinierten Werten für den Start des Verschiebens:

SEEK_SET Vom Anfang der Datei. (Zeiger auf dem ersten Byte.)

SEEK_CUR Von der augenblicklichen Position aus.

SEEK_END Vom Ende der Datei aus. (Zeiger auf dem letzten Byte.)

Das `offset` Argument gibt die Anzahl der Bytes vom `start` aus an. Will man also das das vierte Element in einer Datei voller `double` lesen und das siebte ändern, so könnte man dies wie folgt tun:

```
double d;

fp = fopen("myfile","r+");
fseek(fp,4*size(double),SEEK_SET);
fread(&d,sizeof(double),1,fp);

fseek(fp,3*sizeof(double),SEEK_CUR);
d = 3.1415;
fwrite(&d,sizeof(double),1,fp);
printf("%ld",ftell(fp));

fclose(fp);
```

Die `ftell` Funktion liefert als Ergebnis die Position, an der man sich gerade befindet, vom Anfang der Datei aus gezählt. In obigem Beispiel wären wir also an Position 7*`sizeof(double)`.

Außerdem kann man diese Funktion auch dazu nutzen, um die Größe einer Datei in Bytes zu berechnen.

```
...
fseek(in_file,0l,SEEK_END);
printf("es wurden %ld Zeichen gelesen.\n",ftell(in_file));
rewind(in_file);
...
```

Zusammenfassung

- In C gibt es nur eine physikalische Dateiart, den Byte–Stream.
- Man kann eine logische Unterteilung in vier Dateiarten vornehmen, reiner Text, Zeilentext, formatierter Text und Verbunddateien.
- Es gibt prinzipiell drei Dateimodi, nämlich Lesen, Schreiben und Anhängen.

Mehr über Funktionen

Zeiger auf Funktionen

Wir haben Zeiger auf elementare Datentypen kennengelernt. Man kann in C auch Zeiger auf Funktionen definieren. Beim Aufruf einer Funktion wird der Funktionsname ähnlich wie eine Zeiger benutzt. Er repräsentiert die Adresse im Speicher, an der die Funktion beginnt. Der Funktionsaufruf–Operator `()` benutzt diese Adresse, um die Funktion auszuführen.

Somit ist es sinnvoll Zeiger auf Funktionen zuzulassen. Die Deklaration sieht wie folgt aus:

`return_typ (* funktionsname) (argumentliste);` Eine Deklaration und

Zuweisung sieht z.B. wie folgt aus:

```
int (*sort) (int*,int);
int bubblesort(int*,int);  /* Prototyp */
int quicksort(int*,int);
......
sort = bubblesort;
sort(feld,50);       /* ANSI C */
(*sort)(feld,50);    /* Vor ANSI C */
sort = quicksort;
sort(feld,50);
```

Die obigen Aufrufe von `sort()` bzw. `(*sort)()` nehmen den Wert des Zeigers `sort`, nämlich die Adresse von `bubblesort` und übergeben Sie an den Funktionsaufruf–Operator `()`. Zunächst wird das Feld mit der Funktion `bubblesort`, dann mit `quicksort` sortiert.

Funktionen mit variierenden Parametern

Es gibt in C eine Möglichkeit, Funktionen mit einer unbestimmten Anzahl von Parametern zu deklarieren. Dies geschieht mit Hilfe der Ellipse ... und einigen Standardfunktionen in `stdarg.h`. Wir betrachten hier ein einfaches Beispiel, eine Funktion `sum()`, welche eine variierende Anzahl von `int`'s einliest und alle aufsummiert.

```
#include<stdarg.h>
#include<sdtio.h>
int sumvar(int,...);

void main(void)
  {
    int res;
    res = sumvar(4,10,20,30,40);
    res = sumvar(2,3,4);
  }

int sumvar(int count,...)
  {
    va_list parnum;
    int i, total = 0;

    va_start(parnum,count);

    for(i=0; i<count; i++)
      total += va_arg(parnum,int);

    va_end(parnum);
    return total;
  }
```

Am Anfang einer Funktion mit Ellipse muß eine Variable vom Typ `va_list` deklariert werden. Um auf die Variablen zuzugreifen, wird die Funktion `va_start` aufgerufen, und zwar mit der `va_list` Variable und dem letzten "richtig" deklarierten Parameter. Nun kann mit Hilfe von `va_arg` nacheinander auf die Variablen zugegriffen werden. Dabei muß immer der Typ der Variablen mit übergeben werden.