RZTU

Rechenzentrum der

Technischen Universität Hamburg-Harburg

C-Kurs

Dokumentationsschlüssel: MIT.C

Version B

1. Auflage, September 1992

Herausgeber	:	Technische Universität Hamburg-Harburg – Rechenzentrum – Denickestr. 17 2100 Hamburg 90
Autoren	:	Peter Junglas, Sabine Plischke
Bearbeitung	:	Oliver Rueß

Inhaltsverzeichnis

1 Ein Schnellkurs für Eilige **6**

1.1 Kontaktaufnahme: `hello world!` 6

1.2 Arithmetik 8

1.3 Ein/Ausgabe von Zeichen: `charcount, wordcount` 10

1.4 Vektoren und Zeichenketten: `digitcount, longestline` 13

2 Datentypen, Operatoren und Ausdrücke **18**

2.1 Datentypen 18

2.1.1 Typattribute 18

2.2 Konstanten 19

2.3 Variablen 21

2.4 Typumwandlungen 21

2.5 Operatoren 23

2.5.1 Arithmetische Operatoren 23

2.5.2 Logische und Vergleichsoperatoren 23

2.5.3 Inkrement- und Dekrementoperatoren 23

2.5.4 Bit-Manipulationen (optional) 24

2.5.5 Zuweisungsoperatoren 24

2.5.6 Sonstige Operatoren 25

2.5.7 Vorrang und Reihenfolge 25

3 Kontrollstrukturen **26**

3.1 if - else 26

3.2 switch 27

3.3 while 28

3.4 for 28

3.5 do - while 29

3.6 break, continue 29

3.7 goto (optional) 29

4 Funktionen und Programmstruktur **32**

4.1 Funktionen 32

4.2 Lokale und globale Variablen 32

4.3 Speicherklassen . . . 34
4.3.1 auto . . . 34
4.3.2 register . . . 35
4.3.3 static . . . 35
4.3.4 extern . . . 35
4.4 Initialisierung . . . 37
4.5 Der C-Preprozessor . . . 38
4.5.1 Einfügen von Definitionsdateien . . . 38
4.5.2 Textersetzung . . . 38
4.5.3 Bedingte Übersetzung . . . 39

5 Zeiger und Vektoren **41**
5.1 Zeiger und Adressen . . . 41
5.2 Zeiger und Funktionsargumente . . . 42
5.3 Zeiger und Vektoren . . . 43
5.4 Adreß-Arithmetik . . . 44
5.5 Konstante Zeichenketten . . . 45
5.6 Rekursion . . . 47
5.7 Vektoren von Zeigern, Zeiger auf Zeiger . . . 47
5.8 Mehrdimensionale Vektoren . . . 48
5.9 Zeiger auf Funktionen . . . 49
5.10 Argumente aus der Kommandozeile . . . 50

6 Strukturen **52**
6.1 Grundbegriffe . . . 52
6.2 Vektoren von Strukturen . . . 53
6.3 typedef . . . 54
6.4 Unionen (optional) . . . 55
6.5 Rekursive Strukturen (optional) . . . 56

7 Standardbibliotheken in ANSI-C **58**
7.1 Standard-Ein-/Ausgabe . . . 58
7.1.1 printf . . . 59
7.1.2 scanf . . . 60

7.2 Dateibehandlung 61
7.3 Stringfunktionen 62
7.4 Mathematische Funktionen 63
7.5 Sonstige Routinen 63

Einleitung

C gilt als modern, mächtig, universell, gleichzeitig Hochsprache und maschinennah, aber auch als kryptisch, schwer zu verstehen und noch schwerer zu debuggen.

Folgendes Beispiel[1] scheint diesen Unkenrufen recht zu geben:

```
char *a="char*a=%c%s%c;main(){printf(a,34,a,34);}";main() {printf(a,34,a,34);}
```

Dies ist ein Beispiel für ein Programm, das sich selbst, d.h. seinen eigenen Source-Code, ausdruckt. Nun ist es zwar möglich – und in diesem Fall sogar nötig –, so kryptischen Code zu schreiben, ich hoffe aber, im Laufe dieses Kurses zu zeigen, daß man in C genauso gut lesbare Programme schreiben kann wie z.B. in Pascal.

C ist ursprünglich als **Entwurfssprache zur Programmierung des UNIX-Betriebssystems Anfang der 70er von Dennis Ritchie** bei den Bell Laboratories entwickelt worden. Daher verbreitete es sich zunächst zusammen mit UNIX. Alle Tools, Compiler etc. sind in C geschrieben, außerdem der größte Teil des Kerns. Zunächst diente das Buch *The C Programming Language* von Kernighan und Ritchie als Referenz. Seit der Implementierung auf vielen Nicht-UNIX-Maschinen mußten einige Mehrdeutigkeiten geklärt werden, um einen Vorteil von C – seine Einheitlichkeit – zu erhalten. Bei der Erarbeitung eines gültigen **ANSI**-Standards (**A**merican **N**ational **S**tandards **I**nstitute) wurde gleich die Gelegenheit genutzt, einige kleine Verbesserungen vorzunehmen. Der neue Standard wurde 1989 verabschiedet, inzwischen sind fast alle gängigen C-Compiler ANSI-kompatibel (z.B. auf der Convex), und auch das hervorragende Standardwerk von **Kernighan / Ritchie** basiert inzwischen (in seiner **2. Auflage**) auf dem ANSI-Standard.

Da dieses Buch nicht nur der Klassiker der C-Literatur ist, sondern sehr klar und mit vernünftigen Beispielen geschrieben ist, werde ich mich im weiteren in meinem Kurs darauf stützen.

[1] Alle Beispiele dieses Kurses stehen in der directory: /usr/tuhh/info/tu_beispiele/c/ckurs/vorl_rztsp.

	Gliederung	Std.
1	**Ein Schnellkurs für Eilige**	2
	Kontaktaufnahme: `hello world!`	
	Arithmetik	
	Ein-/Ausgabe von Zeichen	
	Vektoren und Zeichenketten	
2	**Datentypen, Operatoren und Ausdrücke**	1
	Datentypen	
	Konstanten	
	Typwandlungen	
	Operatoren	
3	**Kontrollstrukturen**	1
	if - else	
	switch	
	while	
	for	
	do - while	
	break, continue	
4	**Funktionen und Programmstrukturen**	2
	Funktionen	
	Lokale und globale Variablen	
	Speicherklassen (auto, register, static, extern)	
	Initialisierung	
	Der C-Preprozessor	
5	**Zeiger und Vektoren**	2
	Zeiger und Adressen	
	Zeiger und Funktionsargumente	
	Zeiger und Vektoren	
	Adreß-Arithmetik	
	Konstante Zeichenketten	
	Rekursion	
	Vektoren von Zeigern, Zeiger auf Zeiger	
	Mehrdimensionale Vektoren	
	Zeiger auf Funktionen	
	Argumente aus der Kommandozeile	
6	**Strukturen**	1
	Grundbegriffe	
	Vektoren von Strukturen	
	typedef	
	Unios	
	Rekursive Strukturen	
7	**Standardbibliotheken in ANSI-C**	1
	Standard-Ein-/Ausgabe (printf, scanf)	
	Dateibehandlung	
	Stringfunktionen	
	Mathematische Funktionen	
	Sonstige Routinen	

1 Ein Schnellkurs für Eilige

Dieses Kapitel soll die einfachsten Konstruktionen von C so weit vorstellen, daß nützliche Programme geschrieben werden können. Dazu gehören:

- Variablen und Konstanten
- Arithmetik
- Kontrollstrukturen
- Funktionen
- Einfache Ein-/Ausgabemöglichkeiten

Ziel soll sein, möglichst schnell einfache Übungsprogramme schreiben zu können und im weiteren Kursverlauf immer vollständige Beispielprogramme, nicht bloß Fragmente zu verwenden.

1.1 Kontaktaufnahme: `hello world!`

Das erste Programm ist traditionellerweise "Hello, world". Der Text steht im o.a. directory im File `kap1_1.c`.

Das erste Problem ist, dieses File zu übersetzen und in ein lauffähiges Programm zu verwandeln. Dazu kopiere man sich zunächst das File in den eigenen Bereich (meine Verzeichnisse sind begreiflicherweise schreibgeschützt), z.B. unter dem Namen `hello.c`.
Mit Hilfe des Kommandos `cc hello.c` wird der C-Compiler aufgerufen, das Programm übersetzt und gelinkt und das Ergebnis in ein File `a.out` geschrieben. Mit `a.out` kann das Programm dann gestartet werden. Wer einen schöneren Programmnamen vorzieht, kann es gleich mit `cc -o hello hello.c` übersetzen.

Auf Nicht-UNIX-Systemen befrage man das Handbuch des C-Compilers oder hangele sich (wie bei Turbo-C) durch das Menü.

Nun zum Beispiel:
Ein C-Programm besteht im wesentlichen aus 1.) **Variablen**, die Daten enthalten, und Funktionen, in denen die 2.) **Aktionen** beschrieben werden.
C-Funktionen entsprechen den Funktionen und Prozeduren in Pascal oder FORTRAN.
Die Namen von Funktionen sind völlig beliebig, mit einer Ausnahme: Es muß in jedem Programm **genau eine Funktion mit dem Namen** main geben. Das Programm startet mit dem Aufruf von main, alle anderen Funktionen werden direkt oder indirekt von main aus aufgerufen.
Weitere Funktionen kann man direkt in seinen Programmtext einbauen oder aus vorhandenen **Bibliotheken** übernehmen. C stellt eine ganze Reihe von solchen Bibliotheken zur Verfügung; die wichtigste darunter ist die **Standard-Ein/Ausgabe-Bibliothek**. Um dem Compiler Informationen über die Funktionen in dieser Bibliothek zu geben, wird am Anfang mit der #include-Anweisung das File stdio.h mit eingebunden, das diese Informationen enthält.

Die beiden ersten Zeilen sind **Kommentare** und dienen nur der Erklärung. Der Compiler ignoriert alles, was zwischen /* und */ steht, völlig. Kommentare können überall stehen, wo ein Blank steht, und sollten eifrig benutzt werden.

Eine Methode, **Daten zwischen Funktionen auszutauschen**, besteht darin, einer aufgerufenen Funktion eine **Werteliste**, die Argumente, mitzugeben. Die Klammern (...), die dem Funktionsnamen folgen, umschließen diese Liste.
In unserem Beispiel hat die Funktion main keine Parameter, dies wird gekennzeichnet durch das Schlüsselwort void als Parameterliste.

Eine Funktion liefert typischerweise einen Wert zurück. In Pascal oder FORTRAN gibt es noch Prozeduren; das sind Funktionen, die über ihren Namen nichts zurückgeben. In C gibt es keine spezielle Syntax für Prozeduren: stattdessen wird eine Funktion, die nichts zurückgibt, durch das Wort void eingeleitet.

Nach dem Funktionskopf kommen **geschweifte Klammern** {...}, die die Anweisungen umschließen, aus denen die Funktion besteht. In unserem Fall ist das nur die einzige Anweisung:

```
printf ("Hello, world!\n");
```

Eine Funktion wird aufgerufen, indem man ihren Namen angibt, gefolgt von der Liste der Argumente in Klammern. Ein Funktionsaufruf wird, wie jede andere Anweisung, mit dem Semikolon abgeschlossen. Das Semikolon ist also kein Trennzeichen wie in Pascal, sondern ein **Abschlußzeichen**. In unserem Fall wird also die Funktion printf mit dem Argument `Hello, world!\n` aufgerufen.

printf ist eine Bibliotheksfunktion zur Ausgabe. In unserem Beispiel wird einfach die Zeichenfolge zwischen den Anführungszeichen ausgegeben. Eine solche Zeichenfolge wird auch **String-Konstante** genannt, sie enthält zwischen den Anführungszeichen eine im wesentlichen beliebige Folge von ASCII-Zeichen, die genau so von printf in die Ausgabe geschrieben werden. printf erzeugt keinen Zeilenvorschub, sondern schreibt alle Zeichen direkt hintereinander. Man könnte also statt obiger Zeile genauso gut schreiben:

```
printf("Hello");
printf(" wor");
printf("l");
printf("d!\n");
```

Einige ASCII-Zeichen sind von der Tastatur aus schwer zu erreichen. Um sie in eine String-Konstante einfügen zu können, wurden spezielle Symbole eingeführt, die mit \ (Backslash) beginnen; darunter \n für **Newline** (neue Zeile in UNIX: 0x0a ≡ LF) \t für **Tabulator**, \" für **Anführungszeichen** und \\ für den **Backslash**.
Der Versuch, das \n direkt durch Drücken der Return-Taste einzugeben, scheitert:

```
printf(Hello world!
");
```

liefert eine Fehlermeldung vom Compiler; denn innerhalb einer String-Konstante darf kein (explizites) Newline stehen. Ansonsten ist der Compiler bei der **Aufteilung des Programms** sehr entgegenkommend: Außer in String-Konstanten kann fast überall, wo ein Blank steht, auch ein Newline stehen. Die Aufteilung des Textes ist natürlich Sache des persönlichen Geschmacks. Allerdings ist es **sinnvoll, sich ein Format auszudenken und konsequent einzuhalten**. Verschiedene weit verbreitete Möglichkeiten werden z.B. vom Emacs-Editor (der auf der Convex vorhanden ist) unterstützt.

1.2 Arithmetik

Unser nächstes Beispiel soll eine Umwandlungstabelle von Fahrenheit nach Celsius ausgeben, gemäß der Formel:

$$T[°C] = \frac{5}{9} * (T[°F] - 32)$$

Das Programm befindet sich im file `kap1_2.c`.

Im Gegensatz zu unserem letzten Beispiel sind jetzt ein paar Variablen nötig. In C müssen alle Variablen vereinbart werden (wie in Pascal und in sauber geschriebenem FORTRAN), bevor sie benutzt werden.
Normalerweise geschieht dies am Anfang einer Funktion, vor allen ausführbaren Anweisungen. Eine Vereinbarung besteht aus einem **Typ** und einer Liste von **Variablen**, die diesen Typ besitzen, z.B.

```
int    lower, upper, step;
float  fahr, celsius;
```

Der Typ int beschreibt ganzzahlige Werte, deren Wertebereich von der Implementation abhängt – typisch sind 16 Bit oder 32 Bit Variablen. Variablen vom Typ float enthalten Gleitkommazahlen, deren Werte durch Mantisse und Exponent dargestellt werden – typisch sind 32 Bit, das bedeutet etwa 7 Dezimalstellen und einen Wertebereich um $\pm 10^{\pm 38}$.
Weitere elementare Datentypen in C sind:

Typ	Bedeutung	typ. Größe
char	ein einzelnes Zeichen	8 Bit
short	kleiner ganzzahliger Wert	16 Bit
long	großer ganzzahliger Wert	32 Bit
double	Gleitkommazahl doppelter Genauigkeit ca. 15 Dezimalen, $\pm 10^{\pm 308}$	64 Bit

Unser Beispielprogramm beginnt mit einigen Zuweisungen:

```
lower =   0;
upper = 300;
step  =  20;
fahr  = lower;
```

mit denen den Variablen hier Anfangswerte zugewiesen werden. Spannend ist die letzte Zeile: `lower` ist vom Typ int, `fahr` dagegen float. Hier gilt folgende Regel: Solange wie möglich wird mit integer-Werten gerechnet, bei Kombinationen mit floats aber implizit umgerechnet. Damit sind z.B. folgende Zeilen äquivalent:

```
fahr = 0;
fahr = 0.0;
```

Die Berechnung der Temperaturwerte geschieht in einer while-Schleife:

```
while(fahr<=upper)
  {
     ...
  }
```

die folgendermaßen funktioniert: Nach dem Schlüsselwort while steht eine Bedingung in Klammern. Trifft sie zu, wird der Rumpf – der Bereich zwischen den geschweiften Klammern – ausgeführt, dann die Bedingung neu geprüft und ggf. wieder ausgeführt, bis die Bedingung nicht mehr erfüllt ist. Dann wird die Schleife beendet und das Programm hinter den geschweiften Klammern fortgesetzt.

Im Rumpf von while können beliebig viele Anweisungen stehen. Ist es nur eine einzige, dürfen die geschweiften Klammern weggelassen werden, etwa:

```
while(i<j)
   i = 2 * i;
```

Ich empfehle dringend, die Anweisung einzurücken, um sie als Schleife zu kennzeichnen. Darüberhinaus ist es oft von Nutzen, auch für eine einzige Anweisung geschweifte Klammern zu spendieren, z.B. im Hinblick auf spätere Erweiterungen.

Die Rechenarbeit geschieht in der Zeile:

```
celsius = (5.0/9.0) * (fahr - 32.0);
```

Hier ist zu beachten, daß 5/9 Null ergibt, denn es wird so lange wie möglich mit ganzen Zahlen gerechnet. Dagegen ist die Null an der 32.0 reine Schönheit.

Nun sollen die Ergebnisse auch ausgedruckt werden. Dazu benutzen wir wieder printf[2], allerdings in einer komplizierten Form. printf steht für **print f**ormatted, also für formatiertes Drucken. Das geht auf folgende Weise: Das erste Argument von printf ist ein Formatstring, der normale Zeichen enthält, die genauso gedruckt werden, und Formatelemente, die mit % beginnen. Danach folgen so viele Argumente wie Formatbezeichner, die jeweils an der entsprechenden Stelle eingefügt werden.
Z.B. verwendet man %d für ganzzahlige Argumente, so daß man für `i = 2, j = 3;` mit

```
printf("Wert 1: %d \t Wert 2: %d\n", i, j);
```

erhält: `Wert 1:  2 Wert 2:  3`
Für float-Zahlen verwendet man analog %f.

Zu jedem Formatbezeichner muß ein passendes Argument vorhanden sein, sonst erhält man falsche Ergebnisse bei der Ausgabe. Damit man Zahlen schön rechtsbündig in Feldern fester Länge ausgeben kann, gibt es viele Möglichkeiten, Formatelemente mit zusätzlichen Zahlen und Zeichen zu verzieren. Die wichtigsten zeigt folgende Tabelle:

[2] printf ist übrigens nicht Bestandteil von C im engeren Sinne, sondern eine Funktion aus einer Bibliothek. Allerdings legt der ANSI-Standard eine Menge solcher Funktionen fest.

Format	Erklärung
%d	ganze Zahl
%6d	ganze Zahl in mindestens 6 Zeichen langem Feld, rechtsbündig
%f	Gleitkommazahl
%6f	Gleitkommazahl, mindestens 6 Zeichen breit
%.2f	Gleitkommazahl, 2 Zeichen nach dem Dezimalpunkt
%6.2f	Gleitkommazahl, 2 Zeichen nach dem Dezimalpunkt, mindestens 6 Zeichen breit

Damit ist das Beispiel `kap1_2.c` vollständig beschrieben.

Eine leichte Abwandlung für das gleiche Problem zeigt Beispiel `kap1_3.c`.

Zunächst wollen wir die ominösen **Konstanten** 0, 300, 20, die im Moment noch im Innern von main begraben sind, an den Anfang holen. Das ist guter Stil, verbessert die Übersicht und erleichtert spätere Änderungen, wenn diese Größen mehrmals auftauchen.
Dazu können wir mit #define symbolische Konstanten vereinbaren:

```
#define   lower   0
```

Damit wird keine Variable definiert, sondern ein Text-Makro beschrieben, d.h. im folgenden Text wird `lower` durch den Text ersetzt, der hinter `lower` steht – bis zum Ende der Zeile. Man beachte, daß hier kein Semikolon steht!

Zum Schluß wollen wir noch die while-Schleife durch eine for-Schleife ersetzen. Diese hat in C folgende Form:

```
for (Intitalisierung; Endetest; Inkrementierung)
    {
      ...
    }
```

wobei man dies ersetzt denken kann durch:

```
Initialisierung;

while(Endetest)
{
   ...
   Inkrementierung;
}
```

Welche Form besser ist, hängt davon ab, was klarer ist. In jedem Fall kann man die üblichen do- bzw. for-Schleifen von FORTRAN und Pascal damit problemlos nachmachen.

1.3 Ein/Ausgabe von Zeichen: `charcount, wordcount`

Die nächsten Beispielprogramme lesen und analysieren Texte. Als erstes betrachten wir das Programm `charcnt.c (kap1_4.c)`, das die Zahl der Zeichen in der Eingabe zählen soll.

Zunächst brauchen wir eine Funktion, die einzelne Zeichen aus der Eingabe liest. Solche Funktionen sind in der Standard-Ein/Ausgabe-Bibliothek vorhanden; die einfachste ist getchar. Nach der Anweisung c = getchar() enthält c das nächste Zeichen aus dem Eingabestrom.
Um einzelne Zeichen zu speichern, gibt es den Typ char. Er ist in der Regel 8 Bit groß, dies ist aber grundsätzlich implementationsabhängig. Jedes Zeichen aus dem Zeichensatz des Rechners kann in einer Variablen vom Typ char gespeichert werden. Dabei verhält sich char genau wie eine kleine Integerzahl, der Integer-Wert eines Zeichens entspricht dabei der internen Darstellung des Zeichens, meist also seinem ASCII-Code. Eine Character-Konstante wird in Hochkommata eingeschlossen, also etwa 'a', 'b', '%' oder auch '\n', '\t'.

Um beim Einlesen von Zeichen mit getchar() das Dateiende feststellen zu können, wird folgender Trick verwendet: Die Deklaration von getchar() lautet:

```
int getchar(void);
```

nicht

```
char getchar(void);
```

Im allgemeinen sind die Werte, die getchar() zurückgibt, so klein, daß sie in eine char-Variable passen, mit einer Ausnahme: Es gibt eine spezielle **Zahl**, EOF, die keinen char-Wert repräsentiert, und die genau dann zurückgegeben wird, wenn getchar auf das Ende des Eingabestroms stößt. Kommt der Eingabestrom aus einer Datei, ist dies das Dateiende (**E**nd **O**f **F**ile), beim Lesen vom Terminal wird es durch ein spezielles Zeichen erzeugt, unter UNIX meist Control-D. Der spezielle Wert von EOF ist völlig unwichtig – er steht in <stdio.h> – , solange er mit keinem char-Wert übereinstimmt.

Unser Beispielprogramm funktioniert so: Zunächst vereinbaren wir die Variable nc, die die Zahl der bisher eingelesenen Zeichen enthalten soll, als long, um mit sehr großen Datenmengen keine Probleme zu bekommen, und initialisieren sie auf 0. In einer while-Schleife lesen wir die Zeichen einzeln ein und testen dabei, ob wir das Dateiende schon erreicht haben. Dazu brauchen wir den Vergleichsoperator "!="(ungleich). Analog steht "==" für Gleichheit, mit Zuweisung "=" nicht zu verwechseln.

	Zuweisung	Gleichheitsoperator
C	=	==
Pascal	:=	=
FORTRAN	=	.EQ.

Wenn ein Zeichen eingelesen wurde, erhöhen wir den Wert von nc um eins mit dem Inkrement-Operator ++, der vor oder hinter dem Variablennamen stehen kann (s. Kap. 2). Es bedeutet also ++nc; dasselbe wie nc = nc + 1;. Analog steht --nc; für nc = nc - 1;.
Wenn wir auf das Dateiende gestoßen sind, verlassen wir die while-Schleife und geben das Ergebnis mit printf aus. Dabei ist noch zu beachten, daß für Variablen vom Typ long der Formatbezeichner %ld verwendet wird.

Unser nächstes Beispielprogramm, `wordcnt`, `kap1_5.c`, soll die Zahl der Zeilen, Wörter und Zeichen der Eingabe bestimmen.

Wir benutzen dazu eine Variable `state`, die angibt, ob sich das Programm innerhalb oder außerhalb eines Wortes befindet. Zunächst ist `state` auf `OUT` gesetzt.
Die drei Zähler `nl, nw, nc` werden auf 0 gesetzt. Daß die Schreibweise `a=b=c` funktioniert, liegt daran, daß in C **eine Zuweisung ein Ausdruck** ist und auch einen Wert hat, nämlich den zugewiesenen Wert. I.a. benötigt man diesen Wert nicht, er wird dann einfach ignoriert. Man muß nämlich einen Rückgabewert nicht benutzen. Z.B. ist die Anweisung `1;` legal, wenn auch eine seltsame Formulierung der Null-Operation. Dabei ist `a= (b= c );` verständlich. Außerdem werden Anweisungen **von rechts nach links** ausgeführt, so daß die Klammern entfallen können.

In einer `while`-Schleife werden wieder bis zum Dateiende die einzelnen Zeichen eingelesen. Der Zeichenzähler `nc` wird jedesmal erhöht, die anderen Zähler natürlich nur in bestimmten Fällen. Um solche Fälle unterscheiden zu können, brauchen wir die `if`-Anweisung, die in C so aussieht:

```
if (Bedingung)
   statement
```

oder

```
if (Bedingung)
   statement1
else
   statement2
```

Dabei wird `statement` im allgemeinen ein Block sein, also die Form:

```
 { st1
   st2
   .
   .
   stn
 }
```

haben.

Für kompliziertere Bedingungen brauchen wir die üblichen logischen Operatoren ; sie lauten: || oder — && und — ! nicht.

Damit können wir nun `wordcount` zu Ende schreiben. Falls das Zeilenende '\n' auftaucht, erhöhen wir den Zeilenzähler um eins. Bei '␣', '\t' und '\n', den sogenannten *white space*-Zeichen oder Zwischenräumen, sind wir auf jeden Fall außerhalb eines Wortes (state = `OUT`). Beim Wechsel von `OUT` nach `IN` haben wir ein neues Wort gefunden und erhöhen `nw`. Alle übrigen Zeichen interessieren uns nicht weiter.
Zum Schluß wollen wir auch sehen, was wir – oder besser: der Computer – mühsam berechnet haben.

1.4 Vektoren und Zeichenketten: `digitcount, longestline`

Im nächsten Beispiel (`digitcnt, kap1_6.c`) wollen wir die Eingabezeichen einer genaueren Analyse unterziehen: Wir wollen wissen, wie oft die einzelnen Ziffern auftauchen und wie viele Zwischenräume, Buchstaben und sonstige Zeichen er enthält.
Um die Häufigkeit der Ziffern festzuhalten, benutzen wir nicht 10 Variable, sondern praktischerweise einen **Vektor** (ein eindimensionales Array). Er wird vereinbart mit

```
int ndigits [10];
```

als ein Vektor von 10 Integer-Variablen, die mit `ndigit[0], ndigit[1] , ..., ndigit[9]` angesprochen werden. Vektoren beginnen in C immer mit 0! Als Indizes sind beliebige Integer-Variablen erlaubt. **Vorsicht: Der Compiler prüft in der Regel keine Bereichsüberschreitungen**!

Nach der Deklaration der benötigten Variablen werden zunächst alle Zähler auf 0 gesetzt. Für die Ziffern-Zähler geschieht dies zweckmäßigerweise in einer for-Schleife. In der üblichen Einlese-Schleife prüfen wir, in welche der betrachteten Kategorien das aktuelle Zeichen fällt. Bei den Prüfungen benutzen wir Eigenschaften des benutzten Zeichencodes, die zwar nicht vom ANSI-C-Standard verlangt werden, aber in allen praktisch vorkommenden Codes erfüllt sind: Die Ziffer '0', '1', ..., '9' folgen ohne Lücken direkt aufeinander, ebenso die Kleinbuchstaben 'a' ... 'z' und die Großbuchstaben 'A' ... 'Z'.
Unter dieser Voraussetzung stellen wir mit der Bedingung

```
c >= '0' && c <= '9'
```

fest, ob `c` eine Ziffer enthält.
Daher wird mit der ersten if-Anweisung der richtige Zähler `ndigits[i]` hochgezählt.
Der nächste else-if-Zweig prüft, ob es sich um ein Leerraumzeichen handelt, also '␣', '\n', '\t', der dritte Zweig fischt die Buchstaben (klein oder groß) heraus, und die letzte else-Anweisung sammelt die Übriggebliebenen, also Satz- und Sonderzeichen sowie evtl. vorhandene Control-Zeichen. Zum Schluß drucken wir unsere Ergebnisse aus, den Ziffernzähler natürlich über eine for-Schleife.

Wem es Unbehagen bereitet, sich auf nicht standardisierte Eigenschaften des Zeichencodes verlassen zu müssen, dem hat das ANSI-C-Komittee die Datei <ctype.h> zur Seite gestellt, in der Funktionen wie `isalpha, isdigit` und `isspace` zu finden sind, die alle als

```
int isblabla(char c);
```

deklariert sind und 0 ($\equiv$falsch) oder $\neq 0$ ($\equiv$wahr) zurückgeben.

Das Beispielprogramm `longline (kap1_7.c)` enthält erstmals mehr als die eine Funktion main. Bevor wir uns mit dem Programm selbst beschäftigen, müssen wir also noch etwas über **Funktionen** lernen.

Wie Funktionen **definiert** werden, haben wir schon an main gesehen:

```
Resultattyp Funktionsname( Parameter )
{
   Vereinbarungen
```

```
    Anweisungen
}
```

Falls eine Funktion keine Parameter hat, wird das Wort void für die Parameterliste verwendet. Gibt eine Funktion nichts zurück, wird dies ebenfalls mit dem Resultattyp void gekennzeichnet. Dies entspräche einer SUBROUTINE in FORTRAN bzw. einer PROCEDURE in Pascal.
Im Gegensatz zu Pascal **werden in C die Funktionen nicht ineinander verschachtelt**, sondern stehen auf gleicher Ebene, ggf. in verschiedenen Files. Die Funktion main ist nur dadurch ausgezeichnet, daß bei ihr die Abarbeitung startet (und u.U. Aufrufparameter an sie übergeben werden).

Da Funktionen zum Zeitpunkt des Aufrufs möglicherweise noch gar nicht definiert wurden, muß man sie **deklarieren**, d.h. ihren Resultattyp und ihre Parameterliste angeben. Das geschieht durch:

```
Resultattyp Funktionsname( Parameterliste );
```

startet also wie die **Definition**, enthält dann aber keine Anweisung, sondern schließt mit dem Semikolon ab.
Eine solche Deklaration nennt man auch **Funktions-Prototyp**. Er versetzt den Compiler in die Lage, schon vor der Definition einer Funktion zu prüfen, ob die Funktion mit Parametern vom richtigen Typ aufgerufen wurde.

Nun haben wir schon Funktionen benutzt, anscheinend ohne sie vorher zu deklarieren, nämlich printf und getchar. Hier wird die Bedeutung des **Header-Files** <stdio.h> deutlich: Es enthält u.a. nämlich gerade die Deklarationen einer Menge von Funktionen. Solche Header-Files gibt es für alle Funktionen, die laut ANSI zur Standard-C-Bibliothek gehören.

Ein wichtiger Aspekt von C-Funktionen ist die **Parameterübergabe**: Wie in Pascal werden im Gegensatz zu FORTRAN Parameter normalerweise über ihren **Wert** übergeben. Das bedeutet, daß etwa folgende Funktion

```
void blabla(int i)
{
    i++;
}
```

die mit `blabla(main_int);` aus main aufgerufen wurde, den Wert von `main_int` im Hauptprogramm **nicht verändert**. Diese Praxis verringert – möglicherweise unerwünschte – Seiteneffekte und erhöht damit die Übersichtlichkeit. Außerdem können übergebene Parameter skrupellos in einer Routine verwendet werden. Im Kapitel über Pointer werden wir sehen, wie man trotzdem Variable übergeben kann, die geändert werden können (vgl. VAR-Variable in Pascal).
Eine Ausnahme von der Wert-Übergabe bilden Arrays: Sie werden nicht kopiert, sondern die Adresse des ersten Elements wird übergeben. Dies spart vor allem Zeit und Speicherplatz.

Nach dieser Vorbereitung können wir uns jetzt dem Beispielprogramm zuwenden. Das letzte Beispielprogramm dieses Kapitels soll die längste Zeile der Eingabe finden und ausgeben. Dazu wollen wir noch zwei Themen anreißen: **Strings** und **Funktionsaufrufe**.

Strings (Zeichenketten) sind uns schon als *Konstanten* begegnet in der Form "`Ich bin der kleine String\n`". Sie werden in C als Zeichenvektoren abgelegt.
Eigene String*variablen* deklariert man also mit

```
char   meinstring[25];
```

Die Länge eines Strings wird nirgendwo festgehalten, in der Deklaration (≡ Definition) wird sie nur gebraucht, um den Speicherplatz zu reservieren. Um zu merken, wo der String aufhört, wird das Zeichen mit dem Code `0` als Markierung ans Ende angehängt, es wird mit '\0' bezeichnet. Daher folgt natürlich, daß ein String dieses Zeichen nicht selbst enthalten kann, er würde einfach dort aufhören.

Der konstante String "`juhu`" würde also abgespeichert als | 'j' | 'u' | 'h' | 'u' | '\0' |, also als Vektor von fünf (!) Zeichen. Daher muß man bei Deklarationen von Strings immer darauf achten, daß man ein Zeichen mehr reserviert als maximal nötig ist.
Um einen String auszudrucken, verwendet man `printf` mit dem Formatbezeichner %s, ein einzelnes Zeichen gibt man mit %c aus.
Die Standardbibliothek stellt mit <string.h> eine Menge Funktionen zur Bearbeitung von Strings bereit. Ich werde sie bei Bedarf kurz angeben und benutzen. Eine systematische Zusammenstellung erfolgt im Kapitel über Libraries.
Die Struktur des Beispielprogramms ist

```
while ( es gibt noch eine Zeile )
{
   if ( sie ist laenger als die bisher laengste )
      {
         Zeilenlaenge und Zeile speichern;
      }
}
laengste Zeile ausgeben;
```

Zunächst legen wir die maximale Länge einer Zeile auf 999 Zeichen fest (+ '\0' am Ende) und deklarieren die Hilfsfunktion `getline` zum Lesen einer Zeile und `copy` zum Kopieren von Strings. Ansonsten folgt das Programm obiger Struktur. Mit `getline` lesen wir eine neue Zeile nach `line` ein und bekommen auch gleich ihre Länge nach `len` zurück. In `max` steht das bisherige Längenmaximum. Mit `copy` wird ggf. die neue Zeile nach `longest` kopiert. Zum Schluß wird ausgegeben.

Die Funktion `getline` holt eine Zeile aus der Eingabe, die nicht länger als `lim` ist, und schreibt sie in den String `s`. Außerdem gibt sie die wirkliche Länge dieser Zeile zurück. Das Einlesen geschieht hier in einer `for`-Schleife, die abbricht, wenn das Ende einer Zeile oder der Eingabe auftaucht oder wenn die maximale Zeilenlänge erreicht wird. Zum Schluß wird noch die Stringende-Markierung angebracht und die Länge zurückgegeben.

Die Funktion `copy` kopiert einen String von `from` nach `to`. Bemerkenswert ist die Abbruchmethode: Es wird so lange kopiert, bis in `from[i]` die `'\0'` auftaucht, wobei wieder der Wert der Zuweisung benutzt wird. Für eine allgemein verwendbare `copy`-Funktion sollte man noch eine Längenkontrolle vorsehen, sonst schreibt `copy` u.U. gnadenlos in nicht dafür gedachten Speicher!

Zum Schluß betrachten wir noch eine andere Version `longlinb` (`kap1_8.c`), die sich in zwei Dingen unterscheidet:

- Die Funktion `copy` wurde gestrichen, dafür wird die Funktion `strcpy` der Standard-Bibliothek benutzt, die das gleiche leistet. Sie ist in <string.h> deklariert. Ist für eine Aufgabe eine Bibliotheksfunktion vorhanden, ist es meist vorteilhafter, sie zu benutzen statt neu zu schreiben. Zum einen spart man die Zeit (vor allem zum Debuggen), zum anderen sind die Bibliotheksfunktionen oft effizienter geschrieben und/oder beinhalten bessere Fehlerbehandlungen.

- Die Funktion `getline` hat keine Argumente mehr, sie wird jetzt über die **globale Variable `line`** versorgt. Variablen, die im Innern einer Funktion definiert sind, sind **lokal**, d.h. sie sind außerhalb dieser Funktion nicht verfügbar. Außerdem sind sie **automatisch**, d.h. sie entstehen bei jedem Funktionsaufruf neu und verschwinden beim Verlassen wieder; insbesondere behalten sie keine Werte zwischen zwei Aufrufen der sie definierenden Funktion.
  Im Gegensatz dazu stehen **globale Variable**, die außerhalb aller Funktionen definiert sind: Sie sind in allen nachfolgenden Funktionen (im selben File) verfügbar und behalten ihre Werte. In diesem Sinn entsprechen sie den Variablen im äußersten Block in Pascal oder den COMMON-Blöcke in FORTRAN.
  Vorteil der Verwendung globaler Variablen ist, daß die Parameterliste der Funktionen kurz gehalten werden kann und daß sie immer verfügbar sind.
  Das ist aber auch ein Problem: Es ist nicht mehr offensichtlich wie die Daten im Programm fließen, und in den Funktionen sind die Variablen fest eingebaut, was die Wiederverwendbarkeit von Funktionen fast völlig verhindert.
  Grundsätzlich gilt: So wenig globale Variable wie möglich.

Bemerkungen:

1. Der Operator `sizeof` wird zur Compile-Zeit ausgewertet und liefert die Speicherplatzbelegung. Für einen String bedeutet das also:

   ```
   char s[10] = "hello";
   ```

   `ergibt`

   ```
   sizeof s   =  10
   ```

2. Die Funktion `strlen` liefert die aktuelle Länge eines Strings (ohne '\0') indem bis zur '\0' gesucht wird. Also:

   ```
   strlen(s) = 5
   ```

   `strlen` ist in <string.h> deklariert.

3. Die Funktion `strcpy(s,t)` kopiert `t` nach `s`, **ohne** auf die reservierte Größe von `s` zu achten!

   `strncpy(s,t,n)` kopiert maximal `n` Zeichen von `t` nach `s`. Wird dabei '\0' **nicht** mitkopiert, muß man dies selber machen.

4. Übergibt man ein Array, z.B. einen String, an eine Funktion und versucht mit `sizeof` die Größe des Feldes zu bestimmen, so erhält man meistens 4 als Resultat.
   Erklärung: Es wird nur ein Pointer übergeben, und die Größe eines Pointers ist oft 4 Byte.

Man kann natürlich mit strlen die aktuelle Größe eines Strings, der einer Funktion übergeben wird, auch von dort aus feststellen; dabei wird allerdings **nicht** reservierter Speicherplatz, sondern die aktuelle Länge ausgegeben.

2 Datentypen, Operatoren und Ausdrücke

In diesem Abschnitt werden die einfachen Datentypen beschrieben, sowie die Operatoren, die auf sie wirken.

2.1 Datentypen

Der grundlegene **Integertyp** heißt int, seine Größe hängt von der Implementation ab, beträgt aber mindestens 16 Bit. Für kleine oder sehr große ganze Zahlen gibt es short int oder einfach short und long int bzw. long. short hat mindestens 16 Bit, long mindestens 32 Bit und es gilt:

$$\text{sizeof(short)} \leq \text{sizeof(int)} \leq \text{sizeof(long)}$$

Dabei gibt der sizeof-Operator die Größe eines Typs als Vielfache eines Grundtyps zurück, in der Regel ist dies 1 Byte.
Ein noch kleinerer Integertyp ist char (normalerweise 1 Byte), er ist groß genug, um alle Zeichen des Zeichensatzes darzustellen.
Alle Integertypen gibt es signed oder unsigned, wobei letzteres Zahlen von 0 bis max darstellt. Beispielsweise umfaßt signed char in der Regel die Werte [-128, 127] und unsigned char [0, 255]. Ohne Angabe sind die Integertypen signed, nur bei char hängt es von der Implementation ab.

Für **Fließkommazahlen** gibt es in aufsteigender Reihenfolge die Typen float, double und long double. Schließlich gibt es noch den Typ void. Er wird verwendet, um eine leere Parameterliste zu bezeichnen oder den Typ einer Funktion, die nichts zurückgibt (*Prozedur*). Eine dritte Verwendungsmöglichkeit werden wir im Zusammenhang mit Pointern kennenlernen.

2.1.1 Typattribute

Jeder Typ kann mit Attributen versehen werden, die dem Typbezeichner vorangestellt werden und besonderre Eigenschaften ausdrücken. Es gibt zwei solche Attribute:

- const zeigt an, daß die Variable (außer zur Initialisierung) nicht geändert wird, d.h. sie darf nicht auf der linken Seite einer Zuweisung stehen. Beispiel:

  ```
  const   double   e = 2.718281828;
  const   char     message[] = "Vorsicht!";
  ```

  Bei Vektoren bedeutet const, daß die Elemente nicht geändert werden dürfen.

- volatile: Eine solche Variable kann ohne Einfluß des Programms ihren Wert ändern, z.B. I/O-Ports u.ä. Dies ist nötig, um Optimierer davon abzuhalten, scheinbar redundante Zugriffe wegzuoptimieren.

2.2 Konstanten

- **Integerkonstanten:**
  Ganze Zahlen wie 1234 werden als int interpretiert, es sei denn, sie sind zu groß, dann werden sie long. Um sie explizit long zu machen, hängt man `l` oder `L` an: 1234L. Unsigned-Konstanten bekommen die Endung `u` oder `U`.
  Normalerweise werden Integer als Dezimalzahlen interpretiert; beginnen sie mit 0, sind sie oktal, mit 0x oder 0X hexadezimal:

  ```
  30 = 036 = 0X1E = 0x1e
  ```

- **Gleitkomma-Konstanten:**
  Gleitkomma-Konstanten enthalten einen Dezimalpunkt oder einen Exponenten (2.3, 4., 1e5, 2.0E7). Normalerweise sind sie vom Typ double, durch die Endung f (F) werden sie float, mit l (L) long double.

- **Zeichenkonstanten:**
  Ein einzelnes Zeichen wird von Hochkommata eingeschlossen, etwa 'x', '!', 'O'. Der Wert der Konstanten entspricht dem Wert im Maschinen-Zeichensatz. Im ASCII-Zeichensatz etwa gilt: '0' == 48.
  Um alle Zeichen – auch die nicht druckbaren, darstellen zu können, gibt es Ersatzdarstellungen: Mit '\ooo' bzw. '\xhh', o ∈ {0, 1...7}, h ∈ { 0...9, A...F, a...f }, kann man alle Zeichen direkt oktal oder hexadezimal angeben. Man kann natürlich auch die entsprechende Dezimalzahl angeben. Darüberhinaus gibt es u.a. folgende Abkürzungen:

'\a'	Klingel	'\t'	tabulator
'\b'	Backspace	'\v'	vertical tab
'\f'	form feed	'\\'	\
'\n'	newline	'\''	'
'\r'	carriage return	'\"'	"
'\?'	?	'\0'	NUL (Zahl 0)

Was für '\z', z ein beliebiges Zeichen ohne Extrabedeutung, gesetzt wird, ist abhängig von der Implementation.

Beziehung: char – int Beispiel:

00011001

Character	Integer
EM	25
'\111'	25
'\x19'	25

```
char x;
int y;
```

```
1)  x = '\x19';   x = '\111';   x = 25;
2)  y = '\x19';   y = '\111';   y = 25;
```

Die Anweisungen in 1) bewirken alle dasselbe: Der Speicherzelle, in der `x` abgelegt ist, wird die Bitfolge 00011001 zugewiesen.

Die Anweisungen in 2) bewirken dasselbe für die Speicherzelle, in der `y` abgelegt wird; sie ist nur etwas länger (2 Bytes).

Die markierten Anweisungen erscheinen *natürlich*, die anderen sind aber auch möglich.

Weiterhin ist möglich:

```
x = 0x19;   x = 0111;
y = 0x19;   y = 0111;
```

Die Bitfolge in der Speicherzelle (für `x` und `y`) ist jedesmal dieselbe. Vgl. hierzu das Beispielprogramm `charint.c`.

- **Stringkonstanten**
  Eine Stringkonstante ist eine Folge von Zeichen, eingeschlossen von ", etwa :

  ```
  "juhu", "a", "zeile1 \n Zeile2", "" (Leerstring)
  "\"so so \", murmelte sie".
  ```

  Ein String darf kein explizites *Newline* enthalten:

  ```
  " ja
    ja "
  ```

  Aber Strings, die nur durch Leerräume (space, tab, newline) getrennt sind, werden zusammengezogen:

  ```
  "ach "        "nee"     == "ach nee"
  ```

  Intern sind Strings Vektoren von Zeichen, die durch das Zeichen '\0' (also ASCII - 0) abgeschlossen werden. Daher ist z.B. der String "a" zwei Zeichen lang:

  "a" : | 'a' | '\0' |, also von 'a' verschieden.

  Bei der Deklaration von Strings ist auf das '\0'-Zeichen extra zu achten und ein Zeichen mehr zu reservieren!

- **Aufzählungskonstanten**
  Ähnlich zu den Aufzählungstypen in Pascal gibt es in C die Möglichkeit, Konstanten mit fortlaufenden Werten zu versehen, mit sogenannten enumeration-Listen, z.B.:

  ```
  enum   days { MO, DI, MI, DO, FR, SA, SO};
  ```

  Dies defieniert die Konstanten `MO ... SO` als fortlaufende Zahlen: `MO == 0, DI == 1`, etc., die als Konstanten verwendet werden können. Variablen von solchem Typ werden deklariert mit:

  ```
  enum   days   wochentag;
  ```

  und benutzt:

  ```
  wochentag = MO;
  ```

  In einer `enum`-Liste kann man auch explizit Werte angeben, die nachfolgenden Konstanten zählen von da weiter. Beispiel:

  ```
  enum   bla {NULL,EINS,VIER=4,FUENF,SECHS,ZEHN=10,DROELF=14};
  ```

2.3 Variablen

Variablennamen können aus Klein- und Großbuchstaben, Ziffern und dem Unterstrich '_' bestehen, dürfen aber nicht mit einer Ziffer beginnen. Mit '_' zu beginnen, empfiehlt sich nicht, da der Linker solche Namen erzeugt. 31 Zeichen sind segnifikant, bei externen mindestens 6 (Linker!). Natürlich dürfen Worte wie if, else long usw. nicht als Variablennamen benutzt werden. Normalerweise verwendet man Kleinbuchstaben für Variable und Großbuchstaben für #define-Konstanten, aber das ist nur Konvention.

Variablen müssen vor dem Benutzen definiert werden, etwa:

```
int     minimum, maximum;
float   summe;
char    zeile[100];
int     nocheins;
```

Variablen können gleich zu Beginn einen Wert bekommen ("initialisiert werden") durch

```
int min = 25;
char zeile[] = "Beispiel Zeile";
```

bei Strings wird dabei gleich die Länge (incl. '\0') bestimmt.

2.4 Typumwandlungen

Um Variablen von einem Typ in einen anderen zu verwandeln, gibt es mehrere Möglichkeiten:

- **implizite Umwandlung**
  Werden Variablen verschiedenen Typs miteinander verknüpft oder einander zugewiesen, wandelt der Compiler sie so in einen gemeinsamen Typ um, daß möglichst wenig Information verloren geht.

  Beispiel:

  ```
  int    i;
  long   l;
  float  f;
  double d;
  ```

Operation	Ergebnis
i + l	long
i + f	float
f + d	double
l = i	klar
i = l	o.k. falls l klein, sonst undefiniert
f = i	klar
i = f	Dezimalstellen werden abgeschnitten
f = d	falls d im float-Bereich liegt: Verlust an Genauigkeit, sonst undefiniert.

Die gleichen Regeln gelten bei Funktionsaufrufen. Nach den Deklarationen

```
void intproc  (int   j);
void floatproc(float x);
```

wird bei Aufrufen `intproc(f);` oder `floatproc(i);` wie oben beschrieben umgewandelt. Sind die Funktionen jedoch vorher nicht deklariert worden, ist das Ergebnis undefiniert.

- **Umwandlungsoperator (cast)**
  Um explizit den Wert einer Variablen in einen anderen Typ umzurechnen, gibt es den cast-Operator:
  (neuer Typ) Variable;
  Z.B.

  ```
  (double) f;
  (long)   i;
  ```

  etc.
  Ist z.B. obige Funktion `floatproc` nicht deklariert, so funktioniert

  ```
  floatproc( (float) i);
  ```

- **Umwandlungsfunktionen**
  Natürlich kann man sich für spezielle Aufgaben auch eigene Umwandlungsroutinen schreiben. Das Beispielprogramm `kap2_1.c` z.B. benutzt die Routine `lower`, um Groß- in Kleinbuchstaben zu verwandeln.
  Diese Routine benutzt Annahmen über den Zeichensatz, die bei ASCII erfüllt sind, bei EBCDIC aber nicht. Um vom benutzten Zeichensatz unabhängig zu sein, stellt ANSI C ähnliche Funktionen in der Standardbibliothek zur Verfügung. Sie werden deklariert in der Datei <ctype.h>.
  Beispiel:

  ```
  int   tolower(int c);
  int   toupper(int c);
  ```

  Weiter gibt es verschiedene Funktionen, die Eigenschaften von Zeichen testen, z.B. `isupper(c), islower(c), isdigit(c) ...` Diese Funktionen liefern 0 zurück, falls die Bedingung nicht erfüllt ist (FALSE), sonst irgendeinen von 0 verschiedenen Wert (TRUE).
  Weitere Umwandlungsfunktionen gibt es in <stdlib.h>, z.B., um aus Ziffern (Strings) die entsprechenden Zahlen (double oder int) zu machen:

  ```
  double atof(const char s[]);
  int    atoi(const char s[]);
  ```

  und in <math.h>

  ```
  double ceil(double x);
  double floor(double x);
  ```

2.5 Operatoren

Nachdem wir Variablen einer Menge von Typen kennen, wollen wir auch etwas mit ihnen tun. Dazu bietet uns C eine Menge von Operatoren an.

2.5.1 Arithmetische Operatoren

Bei den Grundoperatoren $+, -, *, /$ ist die Funktion eindeutig; wobei zu beachten ist, daß *int / int* $\rightarrow$ *int* ergibt. Den Rest der Integerdivision erhält man mit % (modulo). Das Verhalten für negative Argumente ist maschinenabhängig.

2.5.2 Logische und Vergleichsoperatoren

Vergleichsoperatoren: >, >=, <, <=, ==, !=

Beachte: `if(a=b){}` ist gültiger Ausdruck, aber wahrscheinlich nicht das Gewünschte!

Die Vergleichsoperatoren haben geringeren Rang als die arithmetischen, daher ist $a < b + 1 \equiv a < (b + 1)$.

Für logische Operationen gibt es && (und), || (oder) und **!** (nicht). Wichtig ist, daß die einzelnen Ausdrücke bei && und || **von links nach rechts** ermittelt werden, bis das Ergebnis klar ist, d.h. bei && bis zum ersten FALSE, bei || bis zum ersten TRUE. Viele C-Programme verlassen sich auf diese Eigenschaft, z.B. in folgender Einleseschleife für eine Zeile:

```
for(i=0;(c=getchar())!='\n' && c!=EOF; i++)
{
  line[i] = c;
}
```

Erst muß `c` eingelesen werden und dann darf auf EOF getestet werden!

2.5.3 Inkrement- und Dekrementoperatoren

Um den Wert einer Variable um eins zu erhöhen oder zu erniedrigen, gibt es die Operatoren ++ und −−. Das Ungewöhnliche an ihnen ist, daß man sie als Postfix- und Prefix-Operatoren verwenden kann:

```
++i;
i++;
```

Der Effekt für `i` ist in beiden Fällen derselbe, aber in einem komplizierten Ausdruck macht es einen Unterschied:
1.Fall

```
n = 5;
x = ++n; /* Erst inkrementieren, dann zuweisen ( x = 6 ) */
```

2.Fall:

```
n = 5;
x = n++; /* Erst zuweisen, dann benutzen ( x = 5 ) */
```

Als Beispiel betrachten wir Programm `kap2_2.c` , das eine Funktion `strcat` verwendet, um einen String `t` an einen anderen `s` anzuhängen. Die Zeiger `i` und `j` laufen jeweils durch die Plätze von `s` und `t`, bis sie auf die Stringende-Marke '\0' laufen.

2.5.4 Bit-Manipulationen (optional)

Um Bit-Manipulationen mit Integern auszuführen, gibt es in C sechs Operatoren:

Operator	Bedeutung
&	Und-Verknüpfung
|	Oder-Verknüpfung
$\wedge$	EXOR-Verknüpfung
<<	Bit-Verschiebung nach links
>>	Bit-Verschiebung nach rechts
$\sim$	Bit-Komplement unär

Beispiel: Löschen aller Bits bis auf die letzten 3:

```
n = n & 0x7;
```

Die Shift-Operatoren verschieben um so viele Positionen, wie das zweite Argument angibt. << schiebt Nullen nach, >> auch bei unsigned-Typen. Bei Worten mit Vorzeichen wird auf einigen Systemen das Vorzeichen nachgeschoben (arithmetischer Shift), bei anderen Null (logischer Shift).

Als Beispiel diene die Funktion `getbits` in `kap2_3.c`. Das größte Problem: Wie erhält man die Maske $000..\underbrace{111..1}_{n}$, vor allem, da man nicht weiß, wie viele Bits die Variable `x` überhaupt hat? Eine portable Lösung:

$$\begin{aligned} \tilde{\ }0 &= 1111...1 \\ \tilde{\ }0 << n &= 111...1\underbrace{0..0}_{n} \\ \tilde{\ }(\tilde{\ }0 << n) &= 0...0\underbrace{1..1}_{n} \end{aligned}$$

2.5.5 Zuweisungsoperatoren

Als Abkürzung für `i = i + 2;` gibt es `i += 2;` .
Diese Form ist vor allem nützlich, wenn `i` ein komplizierter Ausdruck ist. Außerdem ist es i.a. lesbarer (*erhöhe i um 2*). Analoge Formen gibt es für die meisten binären Operatoren $(+, -, *, /, \%, <<, >>, \&, \hat{}, |)$.

2.5.6 Sonstige Operatoren

Eine Möglichkeit, manche Dinge sehr kompakt zu schreiben, bietet der ? -Operator:

```
Ausdruck1 ? Ausdruck2 : Ausdruck3
```

Falls `Ausdruck1` wahr ist, wird `Ausdruck2` berechnet und als Wert des Ausdrucks bestimmt, sonst `Ausdruck3`. Beispiel:

```
z = (a > b) ? a : b; /* z = max(a,b) */
```

Vorsicht: Solche Konstruktionen führen leicht zu unlesbarem Code!

Der Komma-Operator:

```
Ausdruck1, Ausdruck2, Ausdruck3, ... ,Ausdruckn;
```

Die Ausdrücke werden der Reihe nach berechnet, der letzte bestimmt den Rückgabewert (wird vor allem für for-Schleifen benutzt).

2.5.7 Vorrang und Reihenfolge

Der Vorrang der Operatoren entspricht i.w. der Erwartung. Im Zweifelsfall klammern oder in einer Tabelle nachsehen.
Außer bei && und || ist nicht festgelegt, in welcher Reihenfolge die Operanden **einer** Operation berechnet werden, ebensowenig die Reihenfolge, in der die Argumente einer Funktion berechnet werden. Dies kann zu unerwarteten Ergebnissen führen, wenn man Operatoren, die Variablen ändern, verschachtelt.
Beispiel:

```
a[i] = i++;
```

Die Postfix-Operation legt fest, daß die Zuweisung den neuen (inkrementierten) Wert bekommt, aber nicht, ob der Index noch der alte ist. Dies ist nicht festgelegt und daher maschinenabhängig.

3 Kontrollstrukturen

→ *Test 1*

Eine **Anweisung** ist ein mit ; abgeschlossener Ausdruck, z.B.:

```
a = 4711;
n++;
printf("Juhu!");
```

Mehrere Anweisungen können zu einem **Block** zusammengefaßt werden durch {...} – auch ein Block ist eine Anweisung.

3.1 if - else

Syntax:

```
if (Ausdruck)
      Anweisung1
else
      Anweisung2
```

Ist der Ausdruck von 0 verschieden, wird `Anweisung1` ausgeführt, sonst `Anweisung2`. Es gibt in C keinen speziellen logischen Typ, sondern es gilt immer:

0	FALSE
$\neq 0$	TRUE

Für Freunde des BOOLEAN gibt es aber folgende Möglichkeit:

```
enum boolean      {FALSE, TRUE};

enum boolean      sein_oder_nichtsein;
```

Damit wurden der Aufzählungstyp (enum) `boolean` und die Variable `sein_oder_nichtsein` vom Typ enum `boolean` deklariert. Wem auch das noch nicht langt, für den gibt es die Möglichkeit, eigene Typnamen mit typedef zu erfinden.
Syntax: Um Typ `alttyp` den Namen `Neutyp` zu verpassen: Deklariere `Neutyp` wie eine Variable vom Typ `alttyp` und schreibe typedef davor:

```
typedef     enum   boolean     Boolean;
```

Danach geht:

```
Boolean     sein_oder_nichtsein;
```

Direkter:

```
typedef     enum  {FALSE, TRUE} Boolean;
```

Bei if-else gibt es folgende Zweideutigkeit (vgl. `iftest.c`)

```
if (Bed1)
   if (Bed2)
       { ... }
  else
       { ... }
```

Gehört das else nun zur ersten oder letzten if-Anweisung?

In C gilt: else gehört zum letzten "offenen" if. Möchte man es anders, muß man Klammern setzen.

Um mehr als zwei Möglichkeiten zu unterscheiden, kann man viele ifs hintereinanderschalten:

```
if (Bed1)
      ...
else if (Bed2)
             ...
else if (Bedn)
else
      ...
```

Als Beispiel betrachten wir die Routine `binsearch` (`kap3_1.c`), die einen Wert `x` in einem sortierten Vektor `v` sucht und seinen Index zurückgibt oder -1.
Die Suche geschieht durch Halbieren: Zunächst wird mit dem mittleren Element verglichen, dann die richtige Hälfte ausgewählt, bis das Element gefunden oder sein Fehlen bemerkt wurde.

3.2 switch

Um zwischen mehreren konstanten Möglichkeiten zu unterscheiden, gibt es die switch-Anweisung (Pascal: case)
Syntax:

```
switch (Ausdruck)
{
      case K1 : Anweisungen
              .
              .
              .
      case Kn : Anweisungen
      default : Anweisungen
}
```

`Ausdruck` muß ganzzahlig sein (incl. char!), die Größen `K1..Kn` sind konstante Ausdrücke (d.h. zur Compile-Zeit bestimmbar) und natürlich auch ganzzahlig.
Paßt der Wert des Ausdrucks auf die Konstante `Ki`, wird dahinter weitergemacht; paßt keine, geht es beim optionalen default-Zweig weiter – falls vorhanden, sonst nach der switch-Anweisung.

Der große **Unterschied zu Pascal**: Trifft Bedingung `Ki` zu, werden die Anweisungen nach `Ki` ausgeführt, und zwar alle, auch die nach `Ki+1, .., Kn`, default! Möchte man — wie eigentlich meistens — nur die Kommandos nach `Ki` ausführen, beendet man sie mit break; damit kann man das switch direkt verlassen.

Vorteil des direkten Übergangs ist die Möglichkeit, mehrere Zweige zusammenzufassen. Allerdings ist dies oft unübersichtlich und sollte mit Bedacht eingesetzt werden. Es empfiehlt sich, auch nach dem letzten Zweig ein break einzufügen für spätere Erweiterungen!

3.3 while

```
while (Ausdruck)
      Anweisung
```

`Ausdruck` wird wieder als Bedingung aufgefaßt; bei 0 (FALSE) wird abgebrochen, sonst werden die Anweisungen wiederholt.

3.4 for

```
for (Initialisierung; Test; Inkrementierung)
      Anweisung
```

ist (mit Ausnahme von continue s.u.) äquivalent zu:

```
Initialisierung;
while (Test)
{
      Anweisung
      Inkrementierung;
}
```

Alle Teile von for können weggelassen werden; für den Test heißt dies: immer TRUE.
Als Beispiel betrachten wir die Routine `atoi` (`kap3_2.c`),die einen String in eine ganze Zahl umwandeln soll.
Struktur:

```
Zwischenraum ueberlesen, falls vorhanden
Vorzeichen merken
Zahlenwert lesen und umwandeln
```

Eine weitere häufig benutzte Möglichkeit der for-Schleife benutzt den Komma-Operator, um mehrere Indizes gleichzeitig laufen zu lassen, wie etwa in der Funktion `reverse` (`kap3_3.c`), die einen String am Platz umdreht.

Komma-Operatoren werden normalerweise nur in for-Anweisungen benutzt; man sollte sparsam mit ihnen umgehen.
Natürlich kann man mit einer so allgemeinen for-Schleife sehr kompakten Code schreiben; insbesondere läßt sich eine ganze Schleife in den Kopf packen. Aus Gründen der Übersichtlichkeit sollte man aber nur wirkliche Schleifen-Kontrollanweisungen im Kopf unterbringen, die sich auch als Initialisierung, Abbruchtest und Inkrementierung interpretieren lassen.

3.5 do - while

Der repeat-Schleife in Pascal (nicht abweisende Schleife) entspricht die do-while-Schleife in C:

```
do
      Anweisung
while (Ausdruck);
```

Die Anweisung wird ausgeführt, dann geprüft, ob der Ausdruck wahr ist; falls ja, wird die Anweisung wiederholt.

Beachte:

- **Abbruchlogik** andersherum als bei repeat - until
- Semikolon am Ende
- gleiches Schlüsselwort while

3.6 break, continue

Manchmal ist es nützlich, Schleifen vorzeitig verlassen zu können. Mit break kann man eine for, while oder do-Schleife oder eine switch-Anweisung sofort verlassen, danach wird die Anweisung dahinter ausgeführt. continue kann man in Schleifen (for, while, do) benutzen, um eine weitere Iteration zu erzwingen. In while - und do-Schleifen wird also sofort zur Abfrage gesprungen, in for- Schleifen wird zunächst noch inkrementiert.
Dies könnte man z.B. benutzen, um etwas nur mit positiven Zahlen zu machen:

```
for (i=0; i<n; i++)
{
      if a[i] < 0       /* negative Elemente ueberspringen */
        continue;
      ....
}
```

3.7 goto (optional)

Wenn es sich jemand absolut nicht verkneifen kann, gibt es auch in C goto. Dazu braucht man ein label :

```
...
kling_klang_records:
...
```

Irgendwo (später) sagt man dann:

```
goto kling_klang_records;
```

`label` gelten nur innerhalb einer Funktion, größere Sprünge sind so nicht möglich.

Eine vielleicht denkbare Anwendung wäre, aus einer tief geschachtelten Schleifen-Struktur auszubrechen, um etwas nicht-Triviales zu machen, z.B.:

```
for (..)
      for (..)
      {
            if ( voelliges Chaos)
                        goto error;
      }
...
error:
      Rette was zu retten ist;
```

Ich persönlich würde mir allerdings stattdessen eine Fehlerbehandlungsroutine schreiben. Oft kann man sich ein `goto` auch ersparen, indem man Variablen als Merker einführt; z.B. bei folgendem Code zum Auffinden eines gleichen Elements in 2 Vektoren:

Schlecht:

```
for (i=0; i<u; i++)
      for (j=0; j<m; j++)
            if (a[i] == b[j])
                  goto found;
/* keins gefunden */

...
...
return;  /* oder noch mehr Springerei ? */
found:
/* gefunden */

...
...
return;
```

stattdessen (**gut**):

```
typedef enum {FALSE, TRUE}  Boolean;
Boolean found = FALSE;
for (i=0; i<n && !found; i++)
      for (j=0; j<m && !found; j++)
            if (a[i] == b[j])
                  found = true;
if (found)
 {
      /* gefunden */
 }
else
 {
```

```
        /* nicht gefunden */
    }
    /*  Hier kommen wir friedlich wieder zusammen. */
```

4 Funktionen und Programmstruktur

4.1 Funktionen

Eine Funktion wird **definiert** durch

```
Rueckgabetyp funktionsname( Parameterliste )
{
      Vereinbarungen
      Anweisungen
}
```

Eine besondere Anweisung ist die return-Anweisung:

```
return (Rueckgabewert);
```

Dabei muß der Typ des Rückgabewerts mit dem Typ der Funktion übereinstimmen, falls nötig – und möglich – wird entsprechend umgewandelt. Mit `return;` oder `return();` wird kein Wert zurückgeliefert (Typ void).
Aus historischen Gründen darf der Rückgabetyp fehlen, es wird dann int verwendet. Außerdem darf die Parameterliste fehlen, es wird dann void angenommen.

Die **Definition** einer Funktion legt ihren Typ und ihren Text fest. Im Gegensatz dazu steht eine **Deklaration**:

```
Rueckgabetyp funktionsname( Parameterliste );
```

Sie sagt dem Compiler nur, welchen Typ die Funktion und ihre Parameter haben[3]. Deklarationen vor dem ersten Verwenden einer Funktion sind notwendig, denn sonst nimmt der Compiler beim ersten Auftauchen des Funktionsnamens einen Rückgabewert vom Typ int an. Außerdem übergibt er die Parameter, wie sie dastehen, was oft zu Problemen führt (siehe Beispiel `testfun.c`).

Ohne Deklaration wird `1.4` als double übergeben, mit Deklaration dagegen vor der Übergabe zum Typ float konvertiert. Also: Funktionen deklarieren!

4.2 Lokale und globale Variablen

Bisher wurden Variable innerhalb von Funktionen vereinbart, genauer gesagt: im äußersten Block einer Funktion.

Man kann auch zu Beginn *innerer* Blöcke Variablen definieren, die dann nur in diesem Block gelten und äußere Variablen gleichen Namens überdecken, etwa:

```
int   a = 1;                         /* a = 1 */
```

[3] Bei einer Deklaration mit leerer Parameterliste wird **nichts** über die Parameter angenommen; also kann auch kein Parametercheck stattfinden (Unterschied gegenüber Definition!)

```
if (i>0)
{
      float a = 1.2;                    /* a = 1.2 */
}
                                        /* a = 1 */
printf("a = %d",a);
```

Häufiger aber werden Variable auch außerhalb von Funktionen vereinbart, es sind dann *globale* oder *externe* Variablen. Globale Variable gelten von der Vereinbarung an bis zum Ende des Files. Sie werden nur einmal initialisiert und behalten dann ihren Wert bei.

- Vorteil: globale Variable sind immer verfügbar, Parameterlisten können kurz gehalten werden.
- Nachteil: Der Fluß der Daten im Programm ist nicht erkennbar; die Variablennamen sind fest in die Funktionen eingebaut, dadurch wird ihre Wiederverwendbarkeit verhindert.

Grundsätzlich: Globale Variable selten benutzen – nur für spezielle Aufgabe (s.u.)

Zur Vertiefung betrachten wir ein größeres Beispiel (`kap4_1.c`): einen Taschenrechner für $+, -, *, /$ in **u**mgekehrt **p**olnischer **N**otation (UPN), d.h. jeder Operator folgt seinen Operanden ($\Rightarrow$ Klammern überflüssig), z.B.:

```
1 2 - 4 5 + *    =    (1-2)*(4+5)
```

Verfahren: Alle Operanden kommen auf einen Stack; kommt ein Operator, holt er sich die benötigten Operanden und schreibt das Ergebnis wieder auf den Stack.
Programmstruktur:

```
while (naechster Operand oder Operator nicht Dateiende)
   if(Zahl)
         auf Stack
   else if (Operator)
      Operanden vom Stack holen
      Operation ausfuehren
      Resultat auf den Stack
   else if (Zeilenende)
      Wert vom Stack holen und anzeigen
   else
      Fehler
```

Entwurfsidee: Das Hauptprogramm weiß nichts über den Stack, es greift nur über `push` und `pop` darauf zu. Das Holen neuer Operanden/Operatoren erledigt eine weitere Funktion: `getop`.

```
#include ...
#define  ...

/* Funktionsdeklarationen fuer main */
main(...)
```

```
{...}

/* Externe Variablen fuer push und pop (stack) */
void  push(double f)
{...}
double pop(void)
{...}
int getop(char s[])
{...}
/* Von getop aufgerufene Funktionen  ... */
```

- `main`:
  Zu beachten: `pop() - pop()` ist falsch, da die Reihenfolge der beiden Aufrufe nicht festgelegt ist. Stattdessen eine temporäre Variable `op2` einführen.

- `push/pop`:
  Stack `val` und Stackposition `sp` als externe Variable.

- `getop`:
  Problem: Erst nach dem Einlesen des nächsten Zeichens merkt man, daß man ein Zeichen zu viel gelesen hat (z.B. Zahl ist zu Ende). Es wäre schön, wenn man das Zeichen einfach wieder "zurücklesen" könnte, also in die Eingabe zurückstellen, als wäre es nicht gelesen worden. Dazu dienen `getch` und `ungetch`.

- `getch/ungetch`:
  Sie benutzen einen Puffer, in den `ungetch` seine Zeichen zurückstellt. `getch` holt das nächste Zeichen aus diesem Puffer oder von der Eingabe, falls der Puffer leer ist. (Standard-I/O: ungetc: **1** Zeichen zurückstellen für alle Eingabefunktionen (scanf, getc, getchar, ...)).

4.3 Speicherklassen

Komplexe Programme müssen nicht in einem File stehen, sie können in mehreren Dateien liegen, die einzeln übersetzt werden können und dann später zusammen mit Bibliotheksroutinen gelinkt werden. Dabei ist zu beachten, daß eine Funktion nicht auf mehrere Files verteilt sein darf.
Schwieriger ist die Frage, welche Variablen und Funktionen wo gelten und wann sie erzeugt werden (**Gültigkeitsbereich** und **Lebensdauer**).
Zunächst unterscheidet man wie bei Funktionen zwischen **Deklarationen** und **Definitionen**: Eine Deklaration legt den Typ einer Variablen fest, eine Definition reserviert auch Speicherplatz. Eine Variable muß in einem Programm genau einmal definiert werden (in einem einzigen File); soll sie in anderen Files verwendet werden, muß sie dort deklariert werden.

Zunächst stellen wir die vorhandenen Speicherklassen zusammen, dann folgen Beispiele:
Es gibt in C folgende 4 Speicherklassen: auto, register, static und extern (syntaktisch zählt man auch noch typedef dazu!). auto und register: nur Variable. extern und static: Variablen und Funktionen.

4.3.1 auto

Die Angabe dieser Speicherklasse ist nur am Anfang von Blöcken erlaubt und dort der Standard, wird daher meist weggelassen.

Lebensdauer: Die Variablen werden beim Blockeintritt erzeugt und nach Verlassen wieder gelöscht.
Gültigkeit: Auto-Variablen gelten genau im umgebenden Block.

4.3.2 register

Gültigkeit und **Lebensdauer** wie bei auto. register zeigt dem Compiler an, daß auf diese Variable oft zugegriffen wird und er sie daher in einem Register halten soll, falls möglich. Ob das der Compiler tut, ist implementationsabhängig. register-Variable haben keine Adresse, d.h. man kann keine Pointer auf sie zeigen lassen (s.u.).

4.3.3 static

Lebensdauer: Statische Lebensdauer, d.h. static-Variablen werden nur einmal initialisiert und ihr Wert bleibt erhalten.
Gültigkeitsbereich: Nur innerhalb der Datei, bei Variablendefinition in einem Block sogar nur in diesem Block.

4.3.4 extern

Lebensdauer: statisch (wie static)
Gültigkeitsbereich: global, d.h. im ganzen Programm.

Grundsätzlich gilt für alle Deklarationen – auch bei extern: Bekannt ist eine Größe nach einer Deklaration außerhalb eines Blocks bis zum Ende des Files, nach einer Deklaration innerhalb eines Blocks bis zum Ende des Blocks.

Ist keine Speicherklasse angegeben, gelten folgende **Defaults**: Die Speicherklasse für Funktionen ist extern, Variablen in einem Block, werden als auto-Variable betrachtet, sonst als extern.

Regeln für die explizite Angabe von Speicherklassen (static und extern):

	Definition	**Deklaration**
Funktionen	nur sinnvoll static Dadurch ist die Funktion nur innerhalb der Datei bekannt.	falls als static definiert, auch (falls nötig vorher) als static deklarieren. falls die Funktionsdefinition außerhalb der Datei steht, sollte eine extern-Angabe erfolgen.
Variablen	Um statische Lebensdauer bei sonst auto-Variablen zu erreichen: static Zur Einschränkung des Gültigkeitsbereiches auf die Datei bei Definitionen außerhalb eines Blocks: static	Falls die Definition in einer anderen Datei steht: extern static ist eine zusätzliche Angabe bei der Definition. Sie darf nicht als "Deklaration" in Funktionen wiederholt werden (vgl. `stat.c`)

Bemerkungen:

1. Um globale Variable zu benutzen, die in einer anderen Datei definiert wurden, sollte neben der Speicherklassen-Angabe `extern` auch der Typ genannt werde (sonst default: `int`).

   ```
   extern float x;
   ```

   bedeutet: In einer anderen Datei wird `x` definiert als `float x;`.

   Die `extern`-Angabe sagt also **nicht**: Mache diese Variable extern verfügbar, **sondern**: die Variable ist verfügbar und soll benutzt werden.

2. Man kann die Variable nur dadurch extern verfügbar machen, daß man sie außerhalb eines Blockes definiert!

Als Beispiel wollen wir unser Taschenrechner-Programm auf mehrere Files verteilen:

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

#define MAXOP  100
#define NUMBER '0'

extern int getop(char []);
extern void push(double);
extern double pop(void);

void main(void) {...}

-------------------------------------------------------------------

#include <stdio.h>

#define MAXVAL  100

static int sp = 0;
static double val[MAXVAL];

void push(double f) {...}

double pop(void){...}


-------------------------------------------------------------------

#define NUMBER '0'
#include <stdio.h>
#include <ctype.h>

static int getch(void);
```

```
static void ungetch(int);

int getop(char s[]){...}

#define BUFSIZE 100

static char buf[BUFSIZE];
static int  bufp = 0;

static int getch(void){...}

static void ungetch(int c) {...}
```

Um nicht in allen Files die Deklaration der externen Funktionen wiederholen zu müssen, faßt man sie meist in einem **Headerfile** (z.B. `calc.h`) zusammen, das man mit `#include`"`calc.h`n alle Files einbindet (s. Folie).

Konflikt:

1. Damit die Deklarationen von getch und ungetch nur in getop eingebunden werden, müßten sie in ein einzelnes Header-File.
2. push/pop-Deklarationen müßten in ein anderes Header-File.

$\Longrightarrow$ Mehrere Header-Files ?
oder
Ein Header-File, das auch die nicht benötigten Funktionsdeklarationen enthält.

4.4 Initialisierung

Bei der Definition können Variable gleich mit Werten versehen werden, z.B.:

```
int   nr = 25;
```

Für auto und register Variable ist dies nur eine Abkürzung für eine nachfolgende Zuweisung, die bei jedem Blockeintritt durchlaufen wird. Nicht initialisierte Variablen haben <u>keinen</u> Wert!

extern und static Variable werden nur einmal initialisiert, und zwar zu Programmbeginn. Daher können sie auch nur auf konstante Ausdrücke initialisiert werden. Ist kein Wert angegeben, wird 0 angenommen.

Vektoren initialisiert man mit

```
int prime[] = {2, 3, 5, 7, 11};
```

Dabei kann die Dimension weggelassen werden, sie wird vom Compiler ermittelt.
Gibt man die Vektorlänge an und gibt zu wenige Werte vor, wird 0 für den Rest angenommen, zu viele Werte sind ein Fehler.
Für **Strings** gibt es noch folgende Abkürzung:
statt:

```
char  text[5]     = {'j','u','h','u','\0'};
```

geht:

```
char  text[]      = "juhu";   /* enthaelt die '\0'! */
```

4.5 Der C-Preprozessor

Vor der Compilierung wird der Programmtext durch einen Preprozessor bearbeitet, der zeilenorientiert arbeitet und zwar auf Zeilen, die mit *WhiteSpace* und # beginnen.

4.5.1 Einfügen von Definitionsdateien

Mit #include "filenameöder #include<filename> wird das File filename an dieser Stelle in den Text eingefügt. Bei ''...'' wird das File normalerweise im aktuellen Direktory gesucht, bei <...> an einer Standardstelle (unter UNIX meist /usr/include).

4.5.2 Textersetzung

#define name newname

ersetzt für name außerhalb eines Strings den Text newname.

Makros können auch Parameter enthalten, z.B.

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

Dies führt zu direkter Textersetzung; daher auch die Klammern. Vgl.:

```
#define square(x) x*x
```

$\Longrightarrow$ square(x+1) $\rightarrow$ x + 1 * x + 1 = 2x + 1!

Bei Seiteneffekten muß man sehr aufpassen, z.B. wird in max(i++, j++) der größte Wert zweimal expandiert!
Definitionen gelten von #define bis zum Fileende; sie können mit #undef name wieder gelöscht werden. Parameter, die in konstanten Strings stehen, werden nicht ersetzt:

```
#define dprint(expr) printf("expr = %g\n",expr) /* geht nicht !!! */
```

Allerdings kann man mit #name im Makro den Text in einen String verwandeln:

```
#define TEST(a)   #a
```

macht aus $TEST(aha) \rightarrow äha"$.

Dabei werden "durch \'' und \ durch \\ ersetzt. Nützlich z.B. für die Fehlersuche:

```
#define dprint(expr) printf(#expr " = %g\n", expr)
```

bewirkt:

$$\begin{aligned} dprint(x/y); &\rightarrow printf("x/y - -\%g\backslash n", x/y); \\ &\rightarrow printf("x/y = \%g\backslash n", x/y); \end{aligned}$$

Mit ## kann man Argumente zusammenhängen, z.B.:

```
#define paste(front, back) front ## back
```

liefert die Umwandlung paste(name, 1) → name1.

4.5.3 Bedingte Übersetzung

```
#if int-Ausdruck1
   Text1
#elif int-Ausdruck2
     Text2
#else
   Textn
#endif
```

bewirkt, daß nur einer der Texte übersetzt wird, je nachdem, welcher Ausdruck ≠ 0 ist. Die Ausdrücke müssen konstant sein (ohne sizeof) und können den Ausdruck [!]defined (name) enthalten, der (ohne !) 1 ist, falls name vom Preprozessor definiert wurde, 0 sonst.
Als Abkürzung für #if [!]defined(name) gibt es noch #ifdef name bzw. #ifndef name.

Anwendungen:

- Bei komplizierten Fileabhängigkeiten sicherstellen, daß ein Headerfile nur einmal eingebaut wird:

```
#ifndef HDR
#define HDR
  /* hdr.h - Text */
#endif
```

- Zum Debuggen: Teile auskommentieren (die selbst Kommentare enthalten können)

```
#if   0
    Text, der weg soll
#endif
```

- Um maschinenabhängige Teile auszuwählen

```
#if SYSTEM == SYSV
      bla
#elif SYSTEM == BSD
      blu
#endif
```

Bemerkungen:

- Preprozessor-output (auf Convex) mit: cc -E
- Makros werden nicht in "... ünd ' ... ' ersetzt.

5 Zeiger und Vektoren

5.1 Zeiger und Adressen

Wir stellen uns den Speicher einer Maschine als einen großen Vektor von Speicherzellen vor, die durchgehend numeriert sind. Die Nummer einer Speicherzelle heißt ihre Adresse. Wie viele Bytes man braucht, um eine solche Adresse anzugeben, hängt natürlich von der Maschine ab, ein typischer Wert sind 4 Byte, entsprechend einer maximalen Speichergr von 4 GB. Die Größe einer solchen Zelle ist ebenfalls maschinenabhängig, typisch ist 1 Byte.

Ein Zeiger (oder Pointer) ist eine Variable, die die Adresse einer anderen Variable enthält. Da die Größe einer Adresse (in Byte) nicht davon abhängt, auf was der Zeiger zeigt, könnte man eigentlich einen einzigen Pointertyp einführen, der auf alles zeigen kann, einen sogenannten *generischen Pointer*. Einen solchen Typ gibt es auch in C, er heißt Pointer auf void. Er wird allerdings nur selten gebraucht (s.u.). Wichtig sind Pointer auf bestimmte Datentypen, z.B. Pointer auf char, int, float, ... Daß man auch den Typ des Zielobjekts angibt, hat im wesentlichen zwei Gründe:

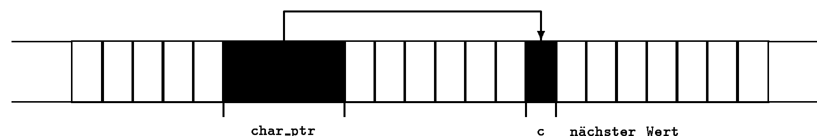
- Zum einen erfordert die Programmierung mit Pointern eine starke Disziplin, um nicht völlig unverständliche und fehleranfällige Programme zu erzeugen. Hierbei kann der **Compiler unterstützen**, wenn er weiß, daß eine bestimmte Variable nur auf einen Integer-Wert zeigen darf.
- Ein noch wesentlich praktischer Grund ist die Möglichkeit von **Pointerrechnungen** (s.u.). Zeigt ein Pointer auf char, so gibt es einen *nächsten* Wert, der um ein Byte weiterzeigt; zeigt ein anderer auf long, so ist der nächstmögliche Wert 4 Byte weiter.

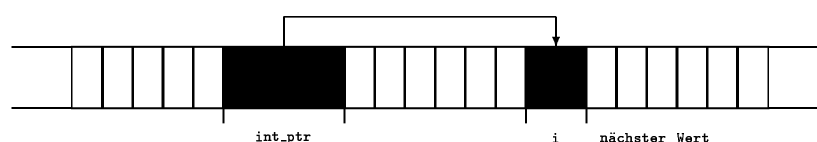
Seien z.B. gegeben:

```
char c;
int  i;
```

und Pointer darauf, die folgendermaßen deklariert werden:

```
char     *char_ptr;
int      *int_ptr;
```





Um die oben deklarierten Pointer auch auf die entsprechenden Größen zeigen zu lassen, braucht man den Adreß-Operator &; z.B.:

```
char_ptr= &c;
int_ptr= &i;
```

Der Adreß-Operator kann nur auf Werte im Speicher angewendet werden, also nicht auf Konstanten, Ausdrücke und register-Variablen.

Der Umkehr-Operator zu & ist der Inhalts-Operator *, der -- auf einen Pointer angewendet -- seinen Inhalt ergibt.
Beispiel:

```
int x = 1, y = 2, z[10];
int *ip;
ip = &x;      /* ip zeigt jetzt auf x  */
y = *ip;      /* y ist jetzt 1         */
*ip = 0;      /* x ist jetzt 0         */
ip = &z[0];   /* ip zeigt auf z[0]     */
```

In diesem Beispiel verhält sich *ip (Inhalt des Pointers ip) wie eine ''normale'' Integervariable und kann auch so verwendet werden, z.B. y = *ip + 1; (* hat Vorrang vor arithmetischen Operatoren, nicht aber vor [ ] (Array-Index).)
Pointer selbst sind ebenfalls normale Variablen, die z.B. einander zugewiesen werden können:

```
float    x = 2.1;
float *f_ptr1, *f_ptr2;
f_ptr1   = &x;
f_ptr2   = f_ptr1;     /* beide zeigen auf x   */
```

5.2 Zeiger und Funktionsargumente

Die Verwendung von Zeigern erlaubt es nun auch, die Werte von Variablen aus Unterprogrammen heraus zu ändern, was bei *call by value* ja nicht geht.
Soll z.B. die Funktion swap(x,y) die Werte ihrer Argumente vertauschen, so geht folgendes **nicht**:

```
void swap(int x, int y)
{
    int temp;

    temp = x;              /* falsch !!! */
    x = y;
    y = temp;
}
```

Da swap nur Kopien der Werte von x und y bekommen, bewirkt der Aufruf swap(x,y) gar nichts!

Übergibt man dagegen Zeiger auf die Werte, dann klappt es:

```
void swap(int *px, int *py)
{
    int temp;

    temp = *px;
    *px = *py;
    *py = temp;
}
```

Nach swap(&x, &y) sind die Werte vertauscht, denn swap bekommt nun Kopien der **Adressen** von x und y und kann folglich auf die Werte von x und y selbst zugreifen. (In FORTRAN werden immer Pointer übergeben!)

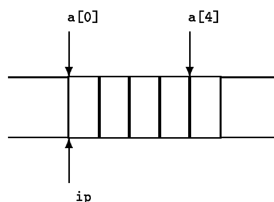
Als weiteres Beispiel betrachten wir die Funktion getint (kap5_1.c), die einerseits einen Integerwert zurückgeben soll, andererseits auch das Dateiende anzeigen muß. Da EOF selbst ein Integerwert ist, klappt der Trick von getchar nicht. Daher geben wir 2 Werte zurück: direkt einen, der EOF anzeigt oder sonst ≥ 0 ist, den Integerwert selbst durch ein Pointerargument.

Danach → *Test 2.*

5.3 Zeiger und Vektoren

In C sind Zeiger und Vektoren eng verbunden. Beispiel:

```
int a[10];
int *ip;
ip = &a[0];
```



Zeigt ip auf ein Vektorelement, dann zeigt ip+1 auf das nächste, ip+2 auf das übernächste usw., und zwar unabhängig davon, wir groß ein einzelnes Element ist (char, double, ...). Falls ip nicht auf das erste Element zeigt, kann man sogar rückwärts laufen: ip-1 zeigt auf das vorhergehende Element. Aber **Vorsicht**: Wenn man die Grenzen des Arrays verläßt, sind alle Werte zufällig!
Zeigt insbesondere also ip auf a[0], dann ist

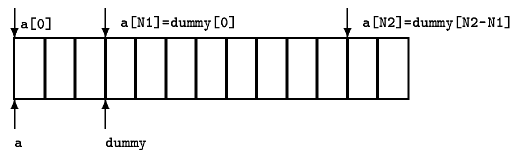
```
a[n] == *(ip+n)
```

Nun ist in C aber der Name eines Arrays mit dem Zeiger auf die Anfangsadresse identisch. Die links und rechts angegebenen Schreibweisen sind äquivalent:

ip = &a[0];	ip = a;
a[n]	*(a+n)
&a[n]	a+n
*(ip+n)	ip[n]

Der einzige Unterschied zwischen ip und a ist, daß ip eine Variable ist, a aber nicht.
Daher geht: ip = a; ip++; aber nicht: a = ip; a++.
Das liegt daran, daß für die Adresse des Arrays a in der Regel kein Speicherplatz reserviert wird, sondern a einem **konstanten Pointer** entspricht: Schon der Compiler legt den Inhalt von a -- d.h. den Speicherplatz, die Adresse -- fest.

Diese Äquivalenz von Pointern und Arrays kann man benutzen, um durch einen Trick Arrays von n1 bis n2 (statt von 0..(n2-n1)) laufen zu lassen, ohne ein Array von 0..n2 zu definieren:




```
#define N1, N2
int dummy[N2-N1+1];
int *a = dummy - N1;

Das bedeutet:

a[N1]    = dummy[0]
  :           :
  :           :
a[N2]    = dummy[N2-N1]
```

Aber wieder **Vorsicht**: Ob man aus dem Array herausläuft, prüft der Compiler nicht!

Bei der Übergabe eines Arrays als Parameter an eine Funktion wird einfach der Zeiger übergeben. Daher wird ein Array automatisch *by reference* übergeben. Außerdem ist innerhalb der Funktion die Größe des Arrays nicht bekannt, wenn man sie nicht als weiteren Parameter oder globale Variable bekannt macht. Somit sind äquivalent:

```
f(int a[]){ ... }
```

und

```
f(int *a) { ... }
```

(Beispiel zum direkten Adressieren von Pointern: kap5_1a.c)

5.4 Adreß-Arithmetik

Zeigen p und q auf Elemente in einem Vektor und ist i klein genug, so daß keine Vektorgr
überschritten werden, dann sind folgende Operationen möglich:

p + i	p - i	p +=i	p -=i	p - q	p = q
p == q	p != q	p < q	p > q	p <= q	p >= q

p - q gibt als Integerwert den Abstand von p und q an; p = NULL ist ebenfalls möglich. Der Null-Pointer wird im Header-file stdio.h definiert und hat den Wert 0, da es keine Adresse 0 in C geben darf.

Explizite Umwandlung von Pointern verschiedener Typen mit **casts** sind ebenfalls erlaubt. Alle weiteren Pointeroperationen sind verboten!

Als Beispiel betrachten wir eine primitive Speicherverwaltung, bestehend aus der Funktion alloc(n), die einen Zeiger p auf n zusammenhängende freie Speicherplätze zurückgibt, und afree(p), die den Platz wieder frei macht (kap5_2.c). In unserem einfachen Beispiel muß afree(p_i) in umgekehrter Reihenfolge der p_i = alloc(n_i) aufgerufen werden.

Ist nicht genug Platz vorhanden, wird der Nullpointer zurückgeben.

Problem: alloc kann nicht wissen, ob es einen Pointer auf char, float, etc zurückgeben soll. Zu diesem Zweck gibt es den *generischen* Pointer vom Typ void:

```
void *alloc(int n);
```

Mann kann ihn problemlos in alle anderen Pointertypen verwandeln[4]. Eine gute Lösung sieht also so aus:

```
void *alloc(int n); /* Deklaration der Funktion alloc */
char *a;
a = (char *) alloc(10);
```

bzw.

```
float *f;
f = (float *) alloc(10*sizeof(float));
```

5.5 Konstante Zeichenketten

Um einen String mit einer Konstanten vorzubesetzen, gibt es zwei Möglichkeiten:

```
char string1[] = "juhu";
char *string2  = "juhu";
```

string1 ist ein Vektor von 5 Zeichen (incl. '\0'), der mit 'j''u''h''u''\0' vorbesetzt ist. string1 ist außerdem ein Zeiger auf char, aber ein konstanter Zeiger, d.h. er zeigt immer auf denselben Platz. Der Inhalt von string1 ('j''u''h''u''\0') kann problemlos *innerhalb der beim Initialisieren angegebenen Größe* geändert werden. string2 ist dagegen ein Pointer, der zunächst dahin zeigt, wo der Text (''juhu'') steht, er kann aber auch woandershin zeigen. Allerdings kann man den Inhalt des Strings direkt nicht verändern, er ist ja konstant.

[4] void-Pointer werden auch implizit umgewandelt.

1) Möglich ist:	2) Nicht möglich ist:
char *cp;	char xxx[10];
⋮	⋮
cp = ''hallo'';	xxx = "hallo";

ad 1) Für die konstante Zeichenkette "hallo"wird beim Compilieren innerhalb des Textsegm des Programms Speicher angelegt. Dorthin darf man aber (normalerweise) nicht schreiben. Z.B. ist **nicht** erlaubt:

```
cp[4] = 'i';
```

Für den konstanten String "hallo"braucht daher (zur Laufzeit) kein Speicherplatz (aus dem Datensegment, das auch zum Schreiben da ist) angefordert zu werden.

Möchte man jedoch den char-Pointer cp auf einen Speicherbereich zeigen lassen, der zunächst "halloënthält und später vielleicht "halli", so muß man Speicher aus dem Datensegment anfordern:

```
#include<stdio.h>
#include<stdlib.h>
void main(void)
{
      char *cp;

      cp = (char *) malloc(6);

      cp[0]='h',cp[1]='a',cp[2]=cp[3]='l',cp[4]='o',cp[5]='\0';

                        /* cp weist auf "hallo" */
      printf("%s\n",cp);
      cp[4] = 'i';      /* cp weist auf "halli" */
      printf("%s\n",cp);
}
```

ad 2) Der konstante Pointer xxx kann **nicht** auf eine andere Speicherzelle zeigen, z.B. nicht in den Speicher im Textsegment, wo ''hallo'' abgelegt ist.

Bemerkung:

int-Pointer können nicht auf Speicherzellen im Textsegment zeigen, denn es gibt keine *konstanten* Arrays aus integern, die über besondere Zeichen ("...") angegeben werden.

Danach → *Test 3*.

Ein weiteres Beispiel für Stringbenutzung ist die Funktion strcpy(s,t), die den String t nach s kopiert. s = t nützt nichts, da es nur den Zeiger kopiert (falls s und t als Pointer vereinbart sind). Dazu → *Test 4*.

Wir betrachten zunächst eine Vektorversion, dann eine Zeigerversion, die man noch in C-typischer Manier zweimal verkürzen kann (kap5_3.c).

5.6 Rekursion

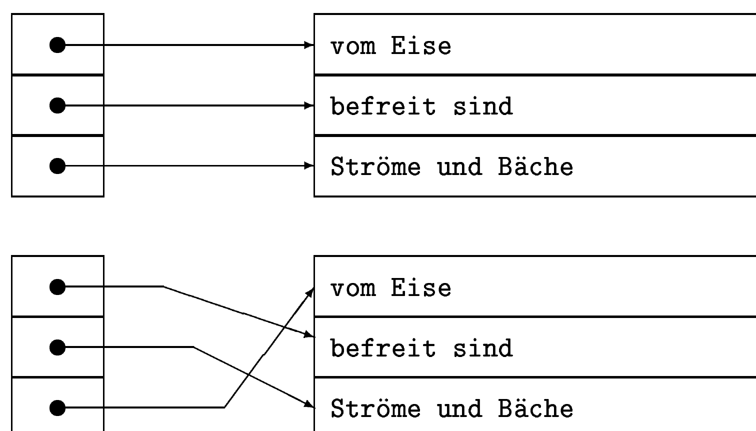
C-Funktionen können sich selbst -- direkt oder indirekt -- aufrufen, d.h. es ist Rekursion möglich.
Da Rekursion in vielen ''klassischen'' Sprachen, z.B. in FORTRAN, nicht möglich ist und zudem die meisten numerischen Anwendungen ohne Rekursion auskommen, wird dieses wichtige Konzept gerade im technisch-wissenschaftlichen Umfeld oft sehr vernachlässigt. Außerdem wird es oft mit Beispielen vorgestellt, die sich besser mit normaler Iteration (for- , while-Schleifen) lösen lassen, so daß seine eigentliche Bedeutung nicht klar wird. Ich möchte dem im folgenden durch ein *richtiges* Rekursions-Beispiel (kap5_3a,b.c) abhelfen.

- **Quick-Sort**
  Idee: man wählt ein Element als mittleres aus und sortiert alle kleineren davor, alle größeren dahinter. Die beiden kleineren Teile sortiert man mittels Rekursion. Sie hört auf, wenn ein Teil nur aus einem Element besteht.

5.7 Vektoren von Zeigern, Zeiger auf Zeiger

Wir wollen im folgenden das Beispiel sort (kap5_4.c) betrachten, das eingegebene Textzeilen alphabetisch sortiert.

Das **Problem** besteht darin, daß die Zeilen verschieden lang sind. Die einfachste Methode: sie in Zeilen einer festen Maximallänge unterzubringen, verschenkt zum einen sehr viel Platz, zum anderen ist der Kopiervorgang von Zeilen sehr mühsam. Daher gehen wir einen anderen Weg: Wir stellen einen ganzen Vektor von Zeigern (*pointer array*) bereit, die jeweils auf den Anfang der Zeile zeigen. Zum Sortieren werden dann nur die Zeiger im pointer array vertauscht.



Bemerkungen zum Programm:

Die entscheidende Variable lineptr wird global definiert.

main enthält die Routinen readlines zum Einlesen der Zeilen, qsort (analog zu kap5_3b.c) zum Sortieren und writelines zum Ausgeben. Bei Fehlern gibt readlines -1 zurück.

readlines liest mit getline (ähnliche Version: kap1_8.c) Zeilen ein und besorgt sich den Platz, sie unterzubringen, mit alloc (kap5_2.c). writelines ist klar.

5.8 Mehrdimensionale Vektoren

Um 2- oder mehrdimensionale Arrays zu bilden, kann man in C den Prozeß der Vektorbildung wiederholen, z.B.: int a; → int a[5]; → int a[5][3];
Der Zugriff auf ein Element geschieht dann mit a[i][j] (**nicht** a[i,j] = a[j])!

Die Initialisierung erfolgt durch doppelt-geklammerte Ausdrücke:

```
int einmaleins[4][4] = { {1,2,3,4},{2,4,6,8},
                         {3,6,9,12},{4,8,12,16}};
```

dabei gelten für jede einzelne Zeile die Regeln wie bei Vektoren. Elemente werden (im Gegensatz zu FORTRAN) zeilenweise abgespeichert, also a[0][0], a[0][1], ... a[0][N] a[1,0], ... a[1][N], ... Bei der Übergabe an Funktionen kann die erste Dimension weggelassen werden (die Programmiererin muß selber darauf achten, daß der zulässige Bereich nicht überschritten wird), die anderen nicht, sonst weiß der Compiler nicht, wo z.B. a[1][0] zu suchen ist (nämlich N Werte nach a[0][0] bei a[M][N]).

Eine Sonderform der Initialisierung gibt es wieder bei Strings, z.B.:

```
char *months[] = { "Sylvester", "Januar", "Februar" };
```

Hier wird soviel Platz erzeugt, wie gebraucht wird (analog zum Beispiel sort (lineptr)).

Eine Bemerkung noch zum *Unterschied zwischen zweidimensionalen-Arrays und Vektoren von Zeigern*. Sei etwa

```
int a[10][20];
int *b[10];
```

Dann sind sowohl a[2][3] als auch b[2][3] grundsätzlich in Ordnung. Während aber für a $10*20$ int-Speicherplätze bereitgestellt werden, die dann gemäß 20*Zeile+ Spalte zugeordnet werden, besteht b zunächst nur aus 10 Zeigern, die noch nirgendwo hinzeigen. b[2] muß also noch irgendwie (durch malloc oder Zuweisungen) auf freien Platz zeigen. Erst dann macht b[2][3] ggf. Sinn (Beispiel multi.c)!

Vergleiche und beachte die Unterschiede:

int *a; →

1. a ist ein Pointer.
2. *a ist ein int.

$\Longrightarrow$ a ist Pointer auf int

Beachte: Für *a gibt es **keinen** Speicherplatz!

---

int a[5]; $\rightarrow$

1. a ist ein konstanter Pointer auf a[0].
2. a[i] ist jeweils ein int.

Hier werden 5 Speicherplätze für je ein int bereitgestellt.

---

int **b; oder int *(*b); $\rightarrow$

1. *b ist ein Pointer.
2. *(*b) ist ein int. aus 1) und 2) $\rightarrow$ *b ist ein Pointer auf int
3. b ist ein Pointer.
4. wie 1) aus 3) und 4) b ist ein Pointer auf Pointer

Beachte: Für *b und **b gibt es keinen Speicherplatz!

---

int b[5][3]; $\rightarrow$

1. b[i] ist jeweils ein konstanter Pointer (auf b[i][0]).
2. b[i][j] ist jeweils ein int.
3. b ist ein konstanter Pointer (auf ein array von Pointern, d.h. auf den Pointer b[0]).
4. wie 1)

5.9 Zeiger auf Funktionen

Gerade in numerischen Anwendungen kommt es häufiger vor, daß man einer Routine eine Funktion übergeben will. Dazu gibt es die Zeiger auf Funktionen. Ist etwa

```
int h(float a, long b);
```

die Deklaration einer Funktion, dann ist

```
int (* h_ptr)(float, long);
```

die Deklaration eines Zeigers h_ptr auf eine solche Funktion. Ähnlich wie bei Arrays steht der Name der Funktion für den Pointer darauf. Die Deklaration einer Funktion f, die eine andere Funktion in der Parameterliste erwartet, ist z.B.:

```
void f(int (* h_ptr)(float, long));
```

Um die Funktion h, die dem geforderten Typ entspricht, zu übergeben, reicht der Aufruf:

```
f(h);
```

Als Beispiel betrachten wir zunächst kap5_4a.c, dann die Integrationsroutine in kap5_5.c.

Unterschiede bei Deklarationen/Definitionen

1. int *a;
   Deklaration + Definition eines int-Pointers a.
2. int *a(double); bzw. int *(a(int));
   Deklaration einer Funktion a, die einen int-Pointer zurückgibt.
3. int a(double);
   Deklaration einer Funktion a, die einen int zurückgibt.
4. int (*a)(double);
   Deklaration und Definition eines Pointers a auf eine Funktion, die einen int zurückgibt.

   Nicht: Deklaration einer Funktion!

5.10 Argumente aus der Kommandozeile

Bisher hatte unser Hauptprogramm immer die Form

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

Nun gibt es aber auch Programme, die man nicht einfach nur mit ihrem Namen aufruft, sondern mit weiteren Parametern von der Kommandozeile aus, z.B.

```
ls -la      UNIX
dir /p/w    MS-DOS
```

Diese Kommandozeilenparameter werden in -- durch Blanks/Tabs getrennte -- Werte zerlegt und an main übergeben, das dazu folgendermaßen deklariert sein muß:

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

Das erste Argument (argc) enthält die Anzahl der Argumente in der Kommandozeile. Es ist immer $\geq$1, denn das erste Argument ist der Name des Programms selbst. In

argv stehen die einzelnen Parameterworte, also argv[0] weist auf den Kommandonamen, argv[1] auf das erste eigentliche Argument usw. bis argv[argc-1].

Außerdem kann das Hauptprogramm auch etwas zurückgeben, das immer vom Typ int sein muß. Dieser Wert wird an den Aufrufer des Programms zurückgegeben (also meist an den Kommandointerpreter, etwa csh oder COMMAND.COM, der darauf reagieren kann oder auch nicht. Dieser Wert wird meist als **Returncode** bezeichnet und oft zur Fehlerdiagnos benutzt.

main hat also i.a. die Form:

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

Die Rückgabe des Wertes geschieht entweder einfach mit return oder mit der Library-Funkti exit (in stdlib.h deklariert). exit hat den Vorteil, daß es auch in Unterprogrammen aufgerufen werden kann, um das Programm abzubrechen und einen Fehlercode zurückzugeben.

Als Beispiel betrachten wir das Programm find (kap5_6.c), das mit find *muster* aufgerufen wird und alle Zeilen des Eingabetextes ausgibt, die das *muster* enthalten. Dazu verwende es die Bibliotheksfunktion

```
char *strstr(const char *cs, const char *ct);
```

die einen Zeiger auf die erste Kopie von ct in cs zurückgibt oder NULL.

6 Strukturen

6.1 Grundbegriffe

Strukturen in C sind die genaue Entsprechung von records in Pascal, nämlich die Ansammlung von mehreren Variablen möglicherweise verschiedenen Typs zu einem neuen Datentyp.

Ein Beispiel ist die folgende Struktur:

```
struct student
{
    char        vorname[21];
    char        nachname[31];
    int         matr_nr;
    unsigned short semester;
    char        fach[31];
};
```

Mit

```
struct student heini;
```

wird die Variable heini deklariert. Auf die einzelnen Elemente greift man zu durch

```
heini.matr_nr = 4711;
strcpy(heini.vorname, "Lisa");
```

(**Nicht:** heini.vorname = Lisa";)
Die einzelnen Komponenten verhalten sich wie ganz normale Variablen, allerdings brauchen ihre Namen nur innerhalb einer Struktur eindeutig zu sein.
Man kann bei der Typendefinition auch gleich Variablen vereinbaren, etwa

```
struct point { int x;
               int y;
             } a, b, c;
```

In diesem Fall kann man auch den Namen der Struktur "point"weglassen. Allerdings kann man sie dann auch nicht für weitere Variablen benutzen.

Ein *Element einer Struktur* kann jeder beliebiger Typ sein, also auch ein Array oder andere Strukturen. Variablen vom Typ Struktur können *initialisiert* werden, indem die Komponentenwerte in geschweiften Klammern angegeben werden:

```
struct point mitte = {512, 640};
```

Ein Beispiel für eine kompliziertere Struktur ist etwa

```
struct rect
```

```
{
  struct point pt1;
  struct point pt2;
};
```

Operationen mit Strukturen:
Man kann Strukturen als Ganzes einander zuweisen, insbesondere auch als Funktionswerte zurückgeben, man kann mit & die Adresse bestimmen und auf die Komponenten zugreifen. Man kann aber **nicht** Strukturen miteinander vergleichen.

Als Beispiel betrachten wir einige Funktionen für Punkte und Rechtecke (kap6_1.c): makepoint, addpoint, ptinrect, canonrect.

Wie die Beispiele zeigen, werden Strukturen genau wie andere Variable (außer Arrays) **als Werte** übergeben. Speziell für große Strukturen ist es jedoch oft effizienter, einen Zeiger auf eine Struktur zu übergeben, etwa

```
struct point *p;
```

auf deren Elemente zugegriffen wird mit

```
(*p).x, (*p).y
```

Die Klammern sind nötig, da der ''.''-Operator höheren Rang hat als der ''*''-Operator. Ohne Klammern wäre *p.x der Inhalt der Komponente p.x, also ein Fehler, da dies kein Pointer ist.

Da Pointer auf Strukturen so oft vorkommen, gibt es folgende **Abkürzung**:
Ist p ein Pointer auf eine Struktur, dann ist p->x $\equiv$ (*p).x die x-Komponente von (*p).

Die Komponenten einer Struktur liegen im Speicher hintereinander, aber nicht unbedingt ohne Lücken. Es könnte nämlich sein. daß eine spezielle Maschinenarchitektur kleinere Zugriffe als 4 Byte gar nicht zuläßt. In diesem Fall würde bei 1 Byte-chars und 4 Byte-ints die Struktur

```
struct a { char c;
           int  i};
```

8 Byte belegen. Die wirkliche Größe einer Struktur erhält man mit dem sizeof-Operator. In unserem Beispiel wäre also sizeof(struct a) = 8.

6.2 Vektoren von Strukturen

Mit Strukturen kann man natürlich genauso verfahren wie mit anderen Typen; z.B. kann man Arrays davon bilden:

```
struct point sechseck[6];
```

definiert sechseck[0], ..., sechseck[5] als 6 Variable vom Typ struct point. sechseck ist ein konstanter Pointer auf sechseck[0], sechseck+2 zeigt auf sechseck[2] usw.

Als Beispiel für die Verwendung von Strukturen betrachten wir das Programm keycount (kap6_2.c), das zählen soll, wie oft die einzelnen reservierten Worte von C in einem Text vorkommen.

Erläuterungen:
struct key enthält ein Wort und seine Häufigkeit, keytab ist das komplette Array, dessen Größe wir vom Compiler errechnen lassen.
main, binsearch: klar

Einziges **Problem**: Woher wissen wir, wie viele Elemente keytab hat? (z.B. für das Suchen oder die Ausgabe wichtig!)
Antwort: Mit sizeof. sizeof hat zwei Formen:

```
sizeof(typ);
sizeof variable;
```

liefern jeweils die Größe in Byte (genauer: Vielfachen von sizeof(char)).
Der genaue Rückgabetyp ist implementationsabhängig, auf jeden Fall wird er in <stdlib.h> als size_t (auf Convex: unsigned int) vereinbart. Damit ermitteln wir, wie viele Elemente keytab enthält:

```
#define NKEYS (sizeof keytab / sizeof keytab[0])
```

getword: holt das nächste Wort aus der Eingabe, also entweder eine Kette aus Buchstaben und Ziffern, mit einem Buchstaben beginnend, oder ein einzelnes Zeichen. lim beschränkt die Wortlänge.

6.3 typedef

typedef ist uns schon vorher begegnet, aber erst durch die Einführung von Pointern, Arrays und Strukturen wird es richtig nützlich. (Der erste Buchstabe eines neuen Typs wird meistens groß geschrieben.)
1. Beispiel:

```
typedef  char *String;
typedef  struct {
                int x;
                int y;
                } Punkt;

String   mein_string = "juhu";
Punkt    ursprung = {0, 0};
```

typedefs bieten zwei Vorteile: Sie erhöhen -- sinnvoll eingesetzt -- die Lesbarkeit des Programms, und sie tragen zur Maschinenunabhängigkeit bei:
Will man etwa mit expliziten 4 Byte-Größen rechnen (warum auch immer), benutzt man einen eigenen Typnamen (z.B. Wort) und definiert Wort mit typedef in einem Headerfile, das maschinenabhängige Teile enthält, etwa:

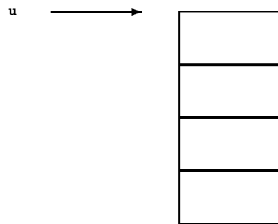
```
#ifdef   PC386
    typedef  long Wort;
#else
    typedef  int  Wort;
#endif
```

6.4 Unionen (optional)

Eine union ist eine Variable, die zu verschiedenen Zeiten Objekte mit verschiedenen Datentypen und Größen enthalten kann.
Die Syntax ist genau dieselbe wie für eine Struktur. Allerdings besitzen alle Elemente gleichsam die relative Adresse 0. Der reservierte Speicherbereich entspricht der größten Alternative.

Beispiel:

```
union    n_tag
        {
            int i_val;        /* 2 Byte */
            double f_val;     /* 4 Byte */
            char c_val;       /* 1 Byte */
        } u;
```



Welche der Alternativen gerade gilt, darum muß sich die Programmiererin selber kümmern, z.B. durch eine entsprechende Variable. Eine Union kann *initialisiert* werden mit einem Wert, der vom Typ der ersten Alternative ist.

Verwendet man eine Union in einer Struktur, erhält man ein C-Entsprechung zu varianten Records in Pascal, z.B.:

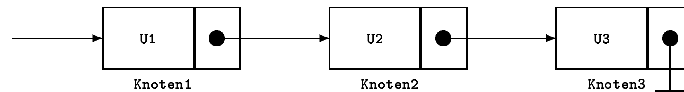
```
enum zustand_t{VORHANDEN, VERLIEHEN, FEHLT};

struct datum_t { int tag;
                 int monat;
                 int jahr; };

struct buch
  {
    char titel[80];
    char autor[80];
```

```
    int  erscheinungsjahr;
    enum zustand_t zustand;
    union {
            char signatur[20]; /* falls vorhanden */
            struct datum_t rueckgabedatum; /* falls verliehen */
            float preis;  /* falls verloren */
        } sonstiges;
}
```

6.5 Rekursive Strukturen (optional)

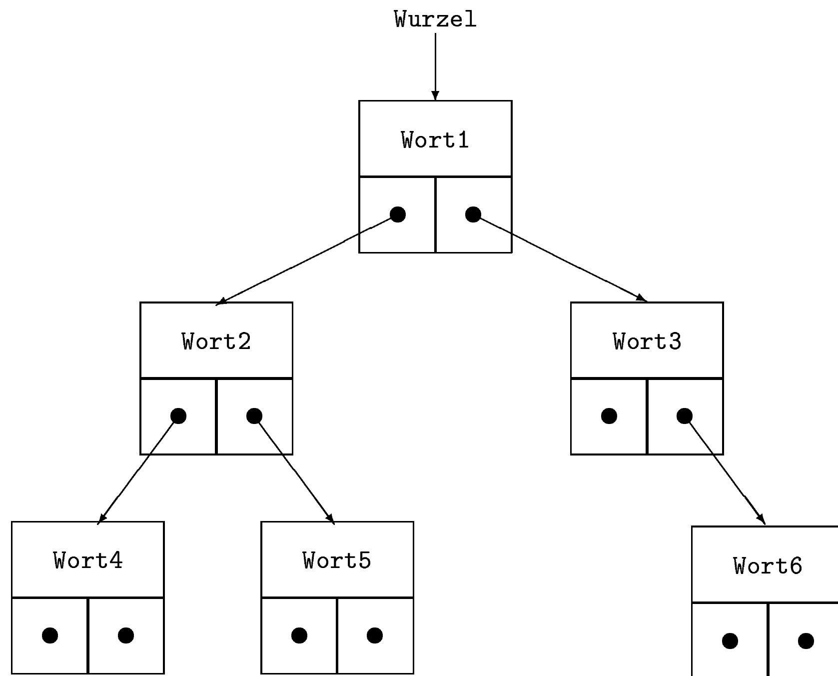
Passend zu den rekursiven Funktionen gibt es auch **rekursive Datentypen**. Sie werden durch Strukturen realisiert. *Zwar können Strukturen nicht sich selbst enthalten, aber Zeiger auf sich selbst.* Dazu zwei Beispiele:

- **verkettete Liste:**




```
struct knoten
 {
    int     datum;
    struct  knoten *naechster;
 } k1, k2, k3;

struct knoten *anfang;

k1.datum = n1;   /* und fuer 2, 3 */

anfang = &k1;
k1.naechster = &k2;
k2.naechster = &k3;
k3.naechster = NULL;
```

- **binärer Baum**




```
struct binknoten
{
    char    wort[80];
    struct   binknoten *links;
    struct   binknoten *rechts;
}
```

Als Beispiel für die Benutzung binärer Bäume betrachten wir das Programm cntwords, das die Zahl der verschiedenen Worte in einem Text ausgibt.
Das **Problem** ist die Datendarstellung: Ein Array mit allen vorkommenden Worten anzulege ist nicht sehr effizient: Entweder muß man -- bei unsortierter Liste -- jedesmal die ganze Liste absuchen, oder man muß zum Sortieren umspeichern. Eine bessere **Lösung** fügt jedes neue Wort so in einen **binären Baum** ein, daß bei jedem Knoten der linke Unterbaum Worte enthält, die vor dem des Knotens liegen, der rechte solche, die alphabet danach kommen (kap6_3.c). Auf diese Weise ist im Mittel die Suchzeit proportional zum Logarithmus der Wortzahl statt zur Wortzahl selbst. Ist die Wortfolge allerdings sortiert, entartet der Baum zur verketteten Liste. In jedem Knoten halten wir außer dem Wort auch seine Häufigkeit fest.

7 Standardbibliotheken in ANSI-C

Viele Dinge, die in anderen Sprachen Teil der Sprachdefinition sind, wie z.B. Ein-/Ausga
mathematische Funktionen etc., werden in C durch Unterroutinen erledigt, die nicht
Teil von C im engeren Sinne sind. Um eine hohe Portierbarkeit von C-Programmen zu
gewährleisten, verlangt der ANSI-Standard das Vorhandensein einer großen Zahl solche
Routinen, deren Parameter und Verhalten genau festgelegt sind. Die interne Implementati
dieser Funktionen ist natürlich frei, bei den meisten ist nicht einmal verlangt,
daß es überhaupt Funktionen sind, es können auch Preprozessor-Makros sein (putc z.B.
ist meist ein Makro).

Um bequem diese Funktionen der Standardbibliothek benutzen zu können, werden ihre
Deklarationen in 15 Headerfiles, nach Aufgabengebiet getrennt, mitgeliefert. Diese
Headerfiles enthalten außerdem i.a. Makrodefinitionen und Preprozessor-Konstanten
(z.B. EOF in <stdio.h>) sowie Typdefinitionen (z.B. struct time in <time.h>). Die
folgenden 15 Headerfiles werden vom ANSI-Standard gefordert:

ANSI-Headerfiles	
<assert.h>	Fehlersuche im Programm
<ctype.h>	Tests für Zeichenklassen
<errno.h>	Fehlercodes
<float.h>	Bereichsgrenzen für Fließkomma-Zahlen
<limits.h>	Bereichsgrenzen für ganze Zahlen
<locale.h>	Landesspez. Parameter (Währungssymbol, Datumsformat)
<math.h>	Mathematische Funktionen
<setjmp.h>	Globale Sprünge
<signal.h>	Signalbehandlung
<stdarg.h>	Variable Argumentlisten
<stddef.h>	NULL, size_t u.a. Typdefinitionen
<stdio.h>	Ein-/Ausgabe
<stdlib.h>	Verschiedenes
<string.h>	Zeichenkettenbearbeitung
<time.h>	Zeitangaben

7.1 Standard-Ein-/Ausgabe

Ein- und Ausgabeoperationen sind in der Regel sehr abhängig von der Hardware und
dem Betriebssystem.
Um die C-Programmiererin davon zu entlasten und Programme systemunabhängig zu machen,
wird durch die I/O-Bibliothek ein einfaches **Modell** für die Ein- /Ausgabe gemacht:
Ein **Zeichenstrom** besteht aus einer Folge von Zeichen, die in (verschieden langen)
Zeilen auftreten, die jeweils mit einem Zeichentrennzeichen ('\n') getrennt sind.
Aus einem Eingabestrom können die Zeichen einzeln gelesen werden (z.B. mit getchar),
ein einzelnes Zeichen kann zurückgestellt werden (mit ungetc). Das Ende des Zeichenstrom
wird von getchar mit dem Rückgabewert EOF angezeigt.

Aufgabe des Programmierers der Standardbibliothek ist es, die Routinen so zu schreiben,
daß nach außen dieses Modell implementiert wird, auch wenn intern alles anders aussieht.
(In MS-DOS z.B. werden Zeilen durch zwei Zeichen getrennt, in VM/CMS sind Zeilen

in Textfiles immer 80 Zeichen lang (Rest Blanks), in VMS beginnen Files mit Längenangabe

Die zeichenorientierten Routinen getchar und putchar kennen wir schon. Für komfortablere Ein-/Ausgabe gibt es die Routinen printf und scanf.

7.1.1 printf

```
int printf(char *format, arg1, arg2,...)
```

printf gibt die Argumente arg1, arg2,... gemäß der Spezifikation des Formatstrings aus und gibt die Zahl der ausgegebenen Zeichen zurück. Ein Formatstring enthält normale Zeichen, die genauso in die Ausgabe kopiert werden, und Formatbeschreiber. Ein Formatbeschreiber beginnt mit % und endet mit einem Umwandlungszeichen, das den Typ des Arguments spezifiziert. Folgende Zeichen sind definiert:

d, i	int, dezimal
o	int (ohne 0..)
x, X	int, hexadezimal (ohne 0x...)
u	unsigned int
c	int (char), als Zeichen
s	char *, Zeichenkette bis '\0'
f	double, festes Komma
e, E	double, mit Exponent
g, G	double, je nach \Schönheitë[E] oder f
p	void *, Pointer
%	%

Zwischen % und Umwandlungszeichen kann noch folgendes (in dieser Reihenfolge!) stehen:

- ein Flag (z.B. '−' für linksbündige Ausgabe)
- eine Zahl für die minimale Feldbreite
- ein Punkt, der Feldbreite und Genauigkeit trennt
- eine Zahl für die Genauigkeit (Nachkommastellen oder max. Zeichenzahl)
- 'h' für short/float, 'l' für long/long double

Beispiel für s="hello, world":

```
:%s:      :hello, world:
:%10s:    :hello, world:
:%.10s:   :hello, wor:
:%-15s:   :hello, world   :
:%15.10s::      hello, wor:
```

7.1.2 scanf

```
int scanf(char *format, arg1, arg2, ...)
```

scanf ist die zu printf analoge Eingabefunktion. Sie liest von der Standardeingabe der Formatangabe entsprechend und legt die Ergebnisse in den übrigen Argumenten ab. Damit das geht, **müssen diese Argumente Zeiger sein!** (Häufigster Fehler!) Beispiel:

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

holt den nächsten Integerwert aus der Eingabe und legt ihn in i ab. Ohne & wäre das Ergebnis i.a. ein *Bus Error*.

Der Formatstring ist wie bei printf aufgebaut und wird ähnlich interpretiert:

- % ...: Formatbezeichner wie bei printf, der die Interpretation der folgenden Zeichen und ihre Umwandlung angibt,
- Zwischenräume (␣, \n, \t, ...) werden ignoriert,
- sonstige Zeichen: müssen genauso in der Eingabe auftauchen, werden überlesen.

scanf liest die Eingabe so lange, wie der Formatbeschreiber darauf paßt, und gibt die Anzahl der umgewandelten Formatelemente zurück oder EOF am Eingabeende. scanf liest über Zeilengrenzen hinweg! Insgesamt ist die Logik von scanf so, daß mit einem bestimmten Formatstring über printf geschriebene Zeichen mit dem gleichen Formatstring für scanf gelesen werden können.

Beispiel:

```
    int day, year, nr;
    char month[20];

    nr=scanf("%d. %s %d", &day, month, &year); /* 1. */
    nr=scanf("%d  %s %d", &day, month, &year); /* 2. */
```

Die beiden scanf-Funktionen ergeben unterschiedliche Ergebnisse bei der Eingabe: 25. Dezember 1991:
1. Funktion:

```
    day     = 25
    month   = "Dezember"
    year    = 1991
    nr      = 3
```

2. Funktion:

```
    day     = 25
    month   = "."
    year      fehlt
    nr      = 2
```

Manchmal möchte man die *schönen* Formatierungsmöglichkeiten von printf nutzen, um eigene schöne Strings zu erzeugen, wenn man nicht auf die Ausgabe, sondern in einen String hineinschreiben will. Dazu gibt es die Funktion sprintf.

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

Sie funktioniert genau wie printf, schreibt aber in den String s (vorher Speicher holen!).
Analog dazu gibt es die Funktion sscanf, die aus einem String statt aus der Eingabe liest. Dies ist besonders praktisch, um zunächst mit getline (oder der Bibliotheksfunktic gets) eine ganze Zeile zu lesen und sie dann mit sscanf weiter zu zerlegen.

7.2 Dateibehandlung

Bisher haben wir nur auf die Standard-Ein-/Ausgabe zugegriffen und ggf. die Umlenkungsm von UNIX oder MS-DOS benutzt, um Files verwenden zu können. Natürlich enthält die Standardbibliothek auch Funktionen, um direkt auf Files zugreifen zu können. Sie werden in <stdio.h> deklariert und fangen fast alle mit f an.

Zunächst muß es eine Möglichkeit geben, vom Programm aus Verbindung zu einer durch einen Filenamen gegebenen Datei aufzunehmen. Dazu dient die Funktion fopen:

```
FILE *fp;
FILE *fopen(char *name, char *mode);
fp = fopen(...,...);
```

fopen bekommt den Filenamen und einen Mode (nämlich ''r'' für Lesen, ''w'' für Schreiben und ''a'' für Anhängen) und gibt einen FILE-Pointer zurück, den man sich für alle weiteren Zugriffe auf die Datei merken muß. Er zeigt auf eine Struktur, die alle Daten enthält, die das System für die Arbeit mit einer Datei benötigt. Ihre Definition ist abhängig von der Implementation.

Zu Beginn jedes Programms sind 3 Filepointer schon definiert: stdin, stdout, stderr.
Sie zeigen auf die Standardeingabe, -ausgabe und den Standardkanal für Fehlermeldungen (meist mit dem Bildschirm verbunden).

Kann fopen eine Datei nicht öffnen (weil sie nicht da ist oder die Zugriffsrechte es nicht erlauben), wird der NULL-Pointer zurückgegeben. Dies muß man auf jeden Fall abfragen, wenn man nicht unangenehme Überraschungen erleben will!
Ist eine Datei erst geöffnet, kann man mit

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

Zeichen daraus lesen (EOF am Fileende) und mit

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

Zeichen schreiben. putc gibt c zurück oder EOF bei Fehlern.
Für formatiertes Schreiben und Lesen gibt es

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

die genau wie printf und scanf funktionieren.

Wenn man ein File nicht mehr benötigt, macht man das mit

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

dem System kund. Daraufhin werden evtl. Buffer geleert und fp wieder freigegeben. Das ist nützlich, weil die Zahl gleichzeitig offener Dateien i.a. pro Prozeß beschränkt ist.

Bei Beendigung eines Programms werden alle noch offenen Files geschlossen.

Zunächst ein Beispiel für formatierte Ein/Ausgabe von/in Dateien (kap7_2a.c). Als nächstes betrachten wir das Programm cat, das mehrere Files hintereinander ausgibt (kap7_3.c). Es verwendet noch die Bibliotheksfunktion exit und ferror:
exit bricht die Programmausführung ab (auch aus Unterprogrammen heraus), schließt offene Dateien und gibt einen Integerwert zurück, die Deklaration lautet:

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

ferror gibt einen Wert $\neq$0 zurück, falls in fp ein Fehler aufgetreten ist.

Zur **zeilenweisen Ein-/Ausgabe** gibt es die Funktionen

```
char *fgets(char *line, int maxline, FILE *fp);
int  fputs(char *line, FILE *fp);
```

für die wir uns Implementierungen als Beispiel ansehen (kap7_4.c).

Weitere I/O-Befehle kann man in C-Handbüchern nachschlagen.

7.3 Stringfunktionen

Funktionen zur Stringmanipulation haben wir schon oft verwendet. Ich möchte nur die wichtigsten zusammenstellen:

strcat(s,t)	hängt t an s an
strcpy(s,t)	kopiert t nach s
strcmp(s,t)	liefert $<,=,>$ 0, je nachdem, ob s $<,=,>$ t ist

Will man garantieren, daß der Zielstring s groß genug ist, um t aufzunehmen, gibt es jeweils Versionen, die maximal n Zeichen kopieren oder vergleichen:

```
strncat(s,t,n);
strncpy(s,t,n);
strncmp(s,t,n);
```

Mit size_t strlen(char *s) bekommt man die Länge der Zeichenkette s (ohne '\0'), mit

```
char *strchr(char *s, char c);
```

einen Zeiger auf das erste Zeichen c in s oder NULL.

7.4 Mathematische Funktionen

Die folgende Funktionen (deklariert in math.h) haben nur double-Argumente und double-Rückgabewerte:

sin(x)	sinh(x)	sqrt(x)
cos(x)	cosh(x)	ceil(x)
tan(x)	tanh(x)	floor(x)
asin(x)	exp(x)	fabs(x)
acos(x)	log(x)	pow(x,y)
atan(x)	log10(x)	atan(y,x)

Zur Fehlerkontrolle wird <errno.h> eingebunden. Dadurch wird die globale Variable int errno definiert, die bei Fehlern einen Wert bekommt. Liegt ein Argument einer Funkton aus <math.h> außerhalb des Definitionsbereichs, bekommt errno den Wert EDOM ("domain error"), liegt der Rückgabewert außerhalb des durch double darstellbaren Bereichs, wird errno auf ERANGE gesetzt. Bei **overflow** ist der Rückgabewert der Funktion (±)HUGE_VAL, bei **underflow** 0. Ob bei underflow errno gesetzt wird, ist abhängig von der Implementierung.

7.5 Sonstige Routinen

In <stdlib.h> werden noch einige Routinen deklariert, die sonst nirgendwo hinpaßten, darunter:

malloc, free	Speicherverwaltung
system	Abschicken von Systemkommandos
rand, srand	Zufallszahlengenerator
bsearch, qsort	binäres Suchen, QuickSort

Einzelheiten darüber und über weitere Library-Funktionen findet man in der Standard-Literatur.