Programmierung in C und Fortran

Meik Hellmund

Sommersemester 2003

Ein Beispielprogramm in C und Fortran:

```c
#include <stdio.h>

int test(int);

int main(){
    int n, res;

    while (1) {
        printf("Bitte ganze Zahl eingeben, \"0\"  für Abbruch:");
        scanf("%i", &n);
        if(n==0) return 0;

        res=test(n);

        if(res==0) {
            printf("Das war leider nicht perfekt. -:(\n");
        } else printf("Treffer!\n");
    }
}





int test(int x){
    int n=0,i;




    for(i=1; i<=x/2; i++) {
        if(x%i == 0) n+=i;
    }



    if (n==x) return 1;
    else return 0;
}
/* --------------------Dateiende -----------------------------*/
```

```fortran
      program bsp1
      implicit none
      integer n
      logical res,test

 1    continue
      write(*,*) 'Bitte ganze Zahl eingeben, "0"  für Abbruch:'
      read(*,*) n

      if(n .eq. 0) then
         return
      end if

      res = test(n)

      if(res) then
         write(*,*) 'Treffer!'
      else
         write(*,*) 'Das war leider nicht perfekt. -:('
      endif
      goto 1
      end


      logical function test(x)
      implicit none
      integer n,i,x

      n=0
      test=.false.

      do 1, i=1,x/2
         if(mod(x,i) .eq. 0) then
            n=n+i
         endif
 1    continue

      if(n .eq. x) test = .true.
      end
c ------------------ Dateiende -----------------------------
```

Einführung und Literatur

Diese Vorlesung ist der Versuch, eine parallele Einführung in C und Fortran zu geben. Die Hoffnung ist, daß abstrakte Konzepte (Datentypen, Ablaufsteuerung, Parameterübergabe an Unterprogramme etc.) durch eine Gegenüberstellung zwei verschiedener konkreter Realisierungen klarer hervortreten.

Schwerpunkt ist die Anwendung der Sprachen zur Lösung von Problemen der numerischen Mathematik. Deshalb wird z.B. der Umgang mit Zeichenketten und Bitmustern nur kurz gestreift, der Umgang mit Feldern und Matrizen dagegen genauer behandelt.

Wert soll gelegt werden auf einen sauberen Programmierstil: Modularisierung, Trennung von Ein/Ausgabe-Routinen und Algorithmen, standardkonforme/betriebssytemunabhängige Programmierung, wiederverwendbaren Code, Verwendung existierender Bibliotheken u.a.

Die Bedeutung von C bedarf wohl keiner Begründung. Fortran ist die Programmiersprache, die sich unmittelbar an den Bedürfnissen von Numerikern orientiert und auf diesem Gebiet hat sie klare Vorteile gegenüber anderen Sprachen, die sich auch in kürzeren Entwicklungszeiten und schnellerem Code niederschlagen.

Leider existiert zur Zeit noch kein freier Fortran90/95-Compiler. Dies ist der Hauptgrund, weshalb diese Vorlesung sich auf Fortran77 beschränkt. Dies wird sich hoffentlich in naher Zukunft ändern.

Es gibt zahlreiche Bücher über C und Fortran. Herausgestellt sei hier nur der C-Klassiker (es exisitieren auch deutsche Übersetzungen)

- B.W. Kernighan und D. Ritchie, *"The C Programming Language"*

Eine umfassende Einführung in numerische Programmierung ist

- Ch. Überhuber, *Computernumerik*, 2 Bände, Springer 1995

Man findet im WWW ebenfalls leicht einführende Texte, Tutorials und Informationen. Hier einige Empfehlungen:

- C.G. Page, *Professional Programmer's Guide to Fortran77*,
  http://www.glue.umd.edu/~nsw/fortran/tutorial/prof77.ps
- T. Love, *ANSI C for Programmers on UNIX Systems*,
  http://www-h.eng.cam.ac.uk/help/documentation/docsource/teaching_C.pdf
- Die C-FAQ (Frequently Asked Questions List), auf Deutsch
  http://www2.informatik.uni-wuerzburg.de/dclc-faq/inhalt.html
  und (ausführlicher) English http://www.eskimo.com/~scs/C-faq/faq.html
- Die Fortran-FAQ http://www.faqs.org/faqs/fortran-faq/
- Alles über Fließkommazahlen:
  W. Kahan, *IEEE Standard 754 for Binary Floating-Point Arithmetic*,
  http://http.cs.berkeley.edu/~wkahan/ieee754status/ieee754.ps
- LAPACK - Das Softwarepaket zur Linearen Algebra http://www.netlib.org/lapack/
- Netlib - Die Sammlung numerischer Software http://www.netlib.org/

Eine Fülle von Informationen bieten auch die man- und info-pages under Unix/Linux. "man gcc" und "man g77" informieren über die Aufrufsyntax und Optionen des C- und Fortran-Compilers, Informationen zu den Funktionen der C-Bibliotheken findet man mit "man 3 <name>", z.B. "man 3 sprintf", "man 3 malloc", "man 3 strncmp". Ausführliche Informationen zur C-Systembibliothek liefert "info libc", zu den Compilern "info gcc" und "info g77".

> Info-Dateien sind Hypertexte: Eine Zeile, die mit einem * beginnt, ist ein Link. Man kann ihm folgen, indem man den Cursor in der Zeile positioniert und enter drückt. Zurück kommt man mit u (up), p und n (previous, next) blättern weiter, /string sucht nach dem Text string. Man kann die info-pages auch mit nichts als der Leertaste von vorne bis hinten durchblättern. Unter KDE kann man auch info:libc oder man:gcc in der URL-Zeile von Konqueror eingeben und danach genauso wie in HTML-Dokumenten navigieren.

1 Geschichte von C und Fortran

1.1 Fortran

1957 John Backus bei IBM entwickelt eine Sprache zur "**For**mula **tran**slation"

1966 ANSI (American National Standards Institute, vgl. DIN in Deutschland) veröffentlicht den ersten Standard für eine Programmiersprache überhaupt: Fortran 66

1978 ANSI Standard X3.9-1978 definiert Fortran 77

1980 Die International Standards Organization ISO adoptiert diesen Standard im Dokument ISO 1539-1980

80er Jahre Eine Expertengruppe arbeitet an einer durchgehenden Modernisierung und Erweiterung der Sprache unter dem Arbeitstitel Fortran 8X.

1991 Fortran 90 wird von ANSI und ISO als Standard veröffentlicht. Neuheiten gegenüber Fortran 77 umfassen die Einführung von Zeigern und dynamischer Speicherverwaltung, Modulen und Interfaces, intrinsische Funktionen für Felder und vieles mehr.

1995 Eine leichte Erweiterung und Überarbeitung, Fortran 95, wird veröffentlicht und schließlich als Standard ISO/IEC 1539-1:1997 verabschiedet.

ca. 2005 geplante nächste Revision des Standards mit Einführung von Objektorientierung, parametrisierten Typen (vgl. "templates" in C++), definierter Schnittstelle zu C-Programmen und mehr.

1.2 C

1970–73 Dennis Ritchie bei AT&T entwickelt C.

1978 "The C Programming Language" von B.W. Kernighan und D. M. Ritchie erscheint. Da die in der 1. Auflage beschriebene Sprache in einigen kleinen Details vom späteren Standard abweicht, wird sie heute oft "K&R C" genannt.

80er Jahre C findet weite Verbreitung. Bjarne Stroustrup nimmt C als Grundlage für objektorientierte Erweiterungen und schafft C++.

1989 Der ANSI C Standard X3.159-1989 wird verabschiedet.

1990 Die International Standards Organization ISO adoptiert diesen Standard im Dokument ISO/IEC 9899:1990

1999 Eine überarbeitete und erweiterte Version, C99 genannt, wird von ISO vorgeschlagen.

1.3 Gegenwärtige Situation

- ANSI C (1989) ist weit verbreiteter Standard. Die Erweiterungen von C99 werden noch wenig genutzt und nicht von allen Compilern unterstützt.
- Fortran77 und Fortran90/95 sind beide weit verbreitet. Die meisten Bibliotheken liegen in Fortran77 vor.
- Diese Vorlesung behandelt, wenn nichts anderes gesagt wird, ANSI C und Fortran77.

2 Grundlagen

Ein Programm besteht aus einer Menge von Funktionen, die in einer oder mehreren Quelltextdateien (source code files) untergebracht sind. Zusätzliche Funktionen können durch Programmbibliotheken (libraries) bereitgestellt werden.

Funktionen bestehen aus Definitionen, Deklarationen und Anweisungen.

Es gibt eine ausgezeichnete Funktion, die den Namen `main` hat.
Diese wird beim Programmstart vom Betriebssystem aufgerufen.

Es gibt eine ausgezeichnete Funktion, die mit der Anweisung `program <name>` eingeleitet wird.
Diese wird beim Programmstart vom Betriebssystem aufgerufen.

Außer den Funktionsdefinitionen kann eine Quelldatei noch

- Präprozessoranweisungen
- globale Definitionen und Deklarationen
- Prototypen (Deklarationen von Funktionen)
- `typedef`-Anweisungen

- `block data`-Anweisungen

enthalten.

Definition und Deklaration

Eine Variable/Funktion ist definiert, wenn sie erzeugt wird und Speicherplatz für sie zur Verfügung gestellt wird. Eine Variable/Funktion wird deklariert, wenn man dem Compiler nur ihren Typ mitteilt und sich ihre Definition an anderer Stelle (z.B. in einer anderen Quelltextdatei oder Bibliothek) befindet.

Namen von Variablen und Funktionen

Namen von Variablen und Funktionen können aus den lateinischen Buchstaben `a-zA-Z`, dem Unterstrich `_` und Ziffern bestehen. Das erste Zeichen darf keine Ziffer sein.

- In Fortran77 ist der Unterstrich nicht erlaubt.
- Groß- und Kleinbuchstaben werden nicht unterschieden (`Nmax` und `NMAX` ist dieselbe Variable).
- Sehr alte Compiler erlauben nur Großbuchstaben.
- Leerzeichen werden ignoriert (`Nmax = 1` und `N ma x = 1` bedeuten dasselbe).

Syntax

Anweisungen enden mit Semikolon. Zeilenenden haben keine syntaktische Bedeutung. Genauer: Jeglicher leerer Raum (Leerzeichen, Tabulatoren, Zeilenumbrüche) zwischen Namen, Operatoren etc. wird ignoriert. In einem Namen oder Operator dürfen aber keine Leerzeichen eingefügt werden.

Klassische Fortran77-Programme haben eine sehr starre Struktur. Jede Anweisung steht in einer Zeile, beginnend ab Spalte 7. Die Zeile darf maximal 72 Spalten lang sein. In Spalte 1-5 können Labels (Anweisungsnummern) stehen. Wenn eine Anweisung nicht in eine Zeile paßt, muß in der nächsten Zeile ein Zeichen in Spalte 6 stehen (Fortsetzungszeichen). Dieses Zeichen ist nicht Bestandteil der Anweisung.
Viele Fortran-Compiler erlauben Abweichungen von dieser starren Form.

Sprungmarken (labels)

Label sind Namen, die durch Doppelpunkt getrennt vor einer Anweisung stehen:

```
    if( y>0 ) goto M1;
    ...
M1: x = sin(y);
```

Label sind Nummern, die in Spalte 1-5 stehen:

```
    if( y .gt. 0) goto 23
    ...
 23 x = sin(y)
```

Kommentare

Kommentare beginnen mit `/*` und enden (eventuell nach mehreren Zeilen) mit `*/`. Sie können nicht verschachtelt sein (`/*` und `*/` haben keine "Klammersyntax"), das erste `*/` in einem Kommentar beendet ihn.
Viele Compiler akzeptieren auch C++-Kommentare, die sich von `//` bis zum Zeilenende erstrecken.

```
/* das ist ein Kommentar */
/* das
   /* auch */
 das nicht mehr */
x=x+1; // C++-Kommentar
```

Kommentare sind Zeilen, die mit einem `*` oder `c` in Spalte 1 beginnen.
Viele Compiler akzeptieren auch Fortran90-Kommentare, die sich von einem `!` bis zum Zeilenende erstrecken.

```
c  Das ist eine Kommentarzeile
     ...
 23 x = sin(y)  ! F90-Kommentar
```

Funktionen und Blöcke

Geschweifte Klammern {} fassen Definitionen und Anweisungen zu einem Block zusammen. An jeder Stelle, an der eine Anweisung stehen darf, kann auch ein Block stehen. Die Definition/Deklaration von Variablen ist nur am Blockanfang möglich.
Alle zu einer Funktion gehörenden Anweisungen bilden einen Block:

```
int test(...) {
...
}
```

Funktionen beginnen mit
`<typ> function <name> (arg1, arg2,...)`
oder
`subroutine <name> (arg, arg2,...)`
und werden mit
`end`
beendet. Definitionen und Deklarationen müssen vor der ersten ausführbaren Anweisung stehen.

3 Vom Quelltext zum ausführbaren Programm

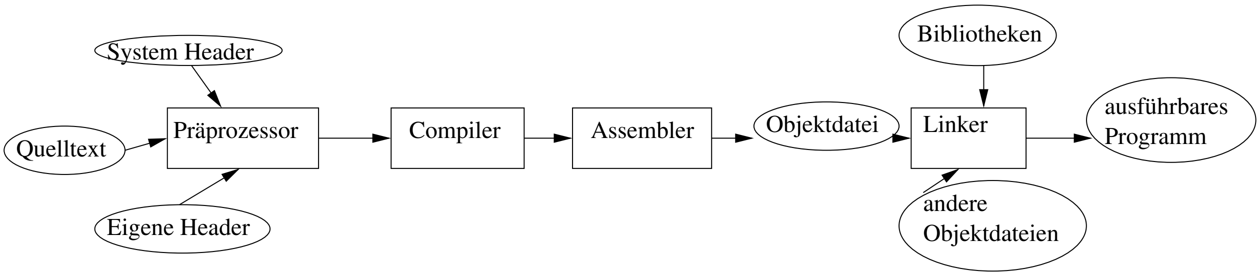

Abbildung 1: Kompilierungsschritte in C

Klassischerweise unterscheidet man zwischen Compiler- und Interpretersprachen. Ein Interpreter liest Anweisung für Anweisung eines Programmtextes ein, übersetzt sie in Maschinensprache und **führt sie sofort aus**. Ein Compiler hingegen transformiert einen Quelltext als Ganzes in eine ausführbare Binärdatei.

Diese Unterscheidung ist heute nicht mehr so scharf möglich: Für viele Sprachen existieren Compiler und Interpreter; die Arbeitsweise vieler Interpreter ist wesentlich komplexer geworden und ähnelt der eines Compilers. Trotzdem kann man wohl sagen, daß C und Fortran typische Compilersprachen sind[1].

Damit unterscheidet man zwischen Dingen (Fehler, etc.), die während des Compilerlaufs (at compile time) oder erst bei der Programmausführung (at run time = zur Laufzeit) passieren. Triviale Ausdrücke (wie z.B `i=1+1;`) werden – spätestens, wenn man mit eingeschalteter Optimierung kompiliert – bereits vom Compiler ausgewertet.

In C (und in vielen Fortran-Inplementationen) wird als erstes ein Präprozessor-Lauf durchgeführt. Dieser macht reine Textsubstitutionen (Kommentare entfernen, den Inhalt anderer Dateien einfügen etc.), d.h. sein Output ist wieder ein C-Quelltext. Anschließend erzeugt der Compiler daraus eine Objektdatei (Dateiendung .o) und der Linker fügt die Objektdatei mit Bibliotheken (Sammlungen von Objektdateien) und evtl. anderen Objektdateien zu einem ausführbaren Programm zusammen.

Alle diese Schritte können durch ein Kommando erledigt werden[2]:

```
gcc beispiel1.c
```

führt alle diese Schritte aus und erzeugt (falls keine Fehler auftreten) eine Datei mit dem ausführbaren Programm namens `a.out`.

[1] Ein interessantes Projekt eines Interpreters für C++ ist das am CERN entwickelte `root`-System, `http://root.cern.ch/`

[2] Die freien GNU-Compiler für C und Fortran heißen `gcc` und `g77`. Kommerzielle Varianten heißen oft `cc` und `f77`.

Ein etwas komplexeres Beispiel ist

```
gcc -c -O2 -Wall -I ../includes beispiel2.c
```

-c läßt den Linker-Schritt weg (d.h., eine Objektdatei beispiel2.o wird erzeugt)
-O2 optimiert den erzeugten Code
-Wall erzeugt Warnungen bei Anweisungen, die zwar gültiges C sind, aber oft Fehlerquellen darstellen (z.B. if(a=b)...)
-I <Pfad> zusätzlicher Suchpfad für #include-Präprozessoranweisungen

Noch ein Beispiel: Hier wird die oben erzeugte Objektdatei beispiel2.o weiterverwendet.

```
gcc -o myprog -Wall -O2  beispiel3.c beispiel2.o  -lm -L/usr/local/lib -lgl
```

-o <name> erzeugt ein Programm namens myprog
<Dateiliste> Liste von Quelldateien, die übersetzt, und Objektdateien, die dazugelinkt werden sollen
-l<name> beim Linken ist die Bibliothek lib<name>, hier also libm (die Bibliothek der Mathematik-Funktionen) und libgl, zu verwenden.
-L <Pfad> zusätzlicher Suchpfad für Bibliotheken

Alle hier angegebenen Optionen existieren analog auch für den Fortran-Compiler g77.

Statisches vs. dynamisches Linken Ohne die Option -static wird dynamisch gelinkt: es wird überprüft, ob alle erforderlichen Bibliotheksfunktionen zur Verfügung stehen, aber sie werden nicht in die Binärdatei kopiert, sondern erst zur Laufzeit geladen – Binärdateien werden wesentlich kleiner und man kann eine Bibliothek durch eine neuere Version ersetzen, ohne Programme neu linken zu müssen.

Der Präprozessor

#include <math.h>	Einfügen der Datei math.h. Sie wird in vordefinierten Systemverzeichnissen (oft /usr/include/) gesucht.
#include "defs/mydef.h"	Einfügen der Datei defs/mydef.h. Sie wird relativ zum aktuellen Verzeichnis gesucht.
#define NMAX 1000	Definition einer Präprozessorkonstanten mit Wert. Die Zeichenfolge NMAX wird von jetzt ab im gesamten Quelltext durch die Zeichenfolge 1000 ersetzt. Präprozessorkonstanten können in 3 Zuständen sein: undefiniert, definiert ohne Wert oder definiert mit einem Wert.
#define DEBUG	Definition einer Präprozessorkonstanten ohne Wert.

#ifdef DEBUG
printf(...);
#endif
	Bedingte Compilierung. #ifdef ... ist wahr, wenn die Konstante mit oder ohne Wert definiert ist. Es gibt auch #ifndef NAME, #else und #undef NAME.

Präprozessorkonstanten können auch beim Compileraufruf mit der -D Option gesetzt werden:

```
gcc -DDEBUG -DNMAX=1000 ...
```

4 Datentypen

Ein Datentyp (kurz Typ) ist mathematisch gesprochen eine Menge von Werten und darauf definierten Operationen. Z.B. umfaßt der Typ `int` (in C) bzw. `integer` (Fortran) auf einer 32bit-CPU die Werte $-2147483648\ldots+2147483647$ ($-2^{31}\ldots 2^{31}-1$) und als Operationen die übliche Arithmetik ganzer Zahlen.

Etwas weniger abstrakt gesehen ist der Typ eines Objektes die Vereinbarung, wieviel Speicherplatz es benötigt und wie die Bitmuster, die den Inhalt dieses Speicherbereichs darstellen, zu interpretieren sind.

Neben den fundamentalen Typen kann eine Programmiersprache die Definition eigener, abgeleiteter Typen erlauben.

C und Fortran sind beides Sprachen mit einem strengen und statischen Typensystem: prinzipiell hat jede Variable und jede Funktion einen Typ und dieser kann während des Programmablaufs nicht verändert werden.

Die beiden Sprachen sind allerdings unterschiedlich konsequent, was die "Strenge" der Typüberprüfung (insbesondere von Funktionen) betrifft und haben jede ihre eigenen Hintertürchen zum Austricksen des Typsystems.

Ganze Zahlen

(und deren Implementierung auf 32Bit-Architekturen)

C		Fortran	
`short int`	2 Byte	`integer`	4 Byte
`int`	4 Byte		
`long int`	4 Byte		
`unsigned short int`			
`unsigned int`			
`unsigned long int`			

Fließkommazahlen

C		Fortran	
`float`	4 Byte	`real` (auch `real*4`)	4 Byte
`double`	8 Byte	`double precision` (auch `real*8`)	8 Byte
		`complex` (2 reals, oft auch `complex*16`, 2 doubles)	

Zeichen und Strings

```
char x='a'
char name[5]="Hans"
char Name[]="Hans"
```
(Compiler ermittelt Länge selbst)

```
character*4 name, name2, name3*6
name='Hans'
```

Boolesche Variablen

`logical` mit den 2 Werten `.true.` und `.false.`

Implizite Deklarationen in Fortran
Wenn eine Variable nicht deklariert ist, wird ihr vom Fortran-Compiler ein Standardtyp zugeordnet: Sie wird als `integer` angesehen, wenn ihr Name mit `i,j,k,l,m,n` anfängt, sonst als `real`. Dieses Feature führt z.B. dazu, daß der Compiler keine Fehlermeldung ausgibt, wenn man sich bei einem Variablennamen verschreibt. Man hat dann implizit einen neue Variable eingeführt. Es läßt sich durch die Anweisung
`implicit none`
am Anfang jeder Funktion/Subroutine abschalten.

Konstanten

Ganzzahlige Konstanten

Integerkonstanten sind vom Typ `int`. Wenn ein `L` oder `l` nachgestellt wird, sind sie vom Typ `long int`. Mit Präfix `0` (Null) werden sie als Oktalzahlen interpretiert, mit einem Präfix `0x` als Hexadezimalzahlen:
`100, 100L, 020 (=16), 0xFF (=255)`

Laut Standard können Integerkonstanten nur im Dezimalsystem angegeben werden. Viele Compiler verstehen aber die Fortran90-Erweiterung
`b'101'` (=5) für Binärzahlen, `o'20'` (=16) für Oktal- und `z'FF'` (=255) für Hexadezimalzahlen.

Gleitkommakonstanten

Beispiele in C und Fortran: `1673.0333e-7, 1e+10, 1., -1.e12, .78`

Die Konstanten sind prinzipiell vom Typ `double`.

Die Konstanten sind vom Typ `real`. Konstanten vom Type `double precision` erhält man, indem man den Exponenten mit `d` statt `e` abtrennt: `1.2d+3`.
Komplexe Konstanten werden als Paare angegeben: `( 3.14, -1.23e-3)`

Zeichenkonstanten

Einzelne Zeichen werden in Apostrophe eingeschlossen und sind vom Typ `int`[a]: `'0', 'A', '"', '\0'` (NUL), `'\n'` (Newline-Zeichen), `'\\'` (Backslash), `'\nnn'` (Beliebiges ASCII-Zeichen durch 3 Ziffern im Oktalsystem dargestellt)
Zeichenketten (strings) werden in Anführungszeichen eingeschlossen: `"bla bla"`. Intern wird ein Feld von Zeichen erzeugt, das ein Byte länger ist und mit `\0` als Endezeichen abgeschlossen wird.

[a] Ja: `int`, nicht `char`. Näheres später unter "impliziter Typumwandlung".

Zeichen und Zeichenketten werden in Apostrophe eingeschlossen. Es gibt keine spezielle Kennung am Ende.

5 Zusammengesetzte Typen

5.1 Homogene Typen (Felder = arrays)

Homogen: Alle Komponenten sind vom selben Typ.
Ein Feld von 10 ganzen Zahlen:

```
int a[10];
```

Die zehn Elemente sind als
`a[0], a[1], ... , a[9]` ansprechbar.

```
integer a(10)
```

Die Elemente sind `a(1), a(2), ... a(10)`.
Andere Indexbereiche können definiert werden:

```
integer a(-2:5)
```

hat 8 Elemente `a(-2), a(-1), a(0),...,a(5)`
Eine alternative Form der Deklaration ist zweistufig:

```
integer a
dimension a(10)
```

Mehrdimensionale Felder:

```
int a[2][3];
```

Die 6 Elemente sind zeilenweise gespeichert, d.h. in der Reihenfolge `a[0][0], a[0][1], a[0][2], a[1][0], a[1][1], a[1][2]`

```
integer a(2,3)
```

Die Elemente sind spaltenweise gespeichert, der erste Index variiert am schnellsten: `a(1,1), a(2,1), a(1,2), a(2,2), a(1,3), a(2,3)`.
Andere Indexbereiche können definiert werden:

```
integer a(-2:5, 2:4)
```

5.2 Inhomogene Typen

```
struct Datum {
      short int Jahr;
      char Monat[3];
      short int Tag;
};                        /* Definition eines Typs "Datum" */

struct Datum Geburtstag; /* Definition einer Variable diesen Typs */

Geburtstag.Monat = "Jan"; /* Komponenten haben Namen statt Indizes */
if(1980 < Geburtstag.Jahr ) {...
```

Verschiedene Daten am selben Speicherplatz

```
union {
      char ByteFeld[4];
      int GanzZahl;
} word;

Verwendung analog zu struct:
word.GanzZahl=1 etc.
```

```
c Feld aus 4 Zeichenketten der Länge 1
    character ByteFeld(4)*1
    integer GanzZahl
    equivalence (ByteFeld, GanzZahl)
```

Die 4 Bytes `ByteFeld[0]...[3]` und die 4 Bytes von GanzZahl belegen denselben Speicherplatz.

6 Ablaufsteuerung

6.1 Verzweigungen

```
if ( <Ausdruck> ) <Anweisung oder Block>
```

```
if ( <Ausdruck> ) <Anweisung oder Block>
else <Anweisung oder Block>
```

```
lhs = <Ausdruck1> ? <Ausdruck2> : <Ausdruck3>
```

Wenn Ausdruck1 wahr ist, wird Ausdruck2 zugewiesen, sonst Ausdruck3. Beispiel:

```
maxXY = x > y ? x : y
```

```
if(<logischer Ausdruck>) Anweisung
```

```
if (<logischer Ausdruck>) then
      <Anweisungen>
else if (<logischer Ausdruck>) then
      <Anweisungen>
else
      <Anweisungen>
end if
```

```
if(<num. Ausdr.>) <label1>,<label2>,<label3>
```

Das Program verzweigt zu <label1>, wenn der Ausdruck negativ ist, zu <label2>, wenn er Null ist und zu <label3>, wenn er positiv ist.

6.2 Mehrfachverzweigungen

```
switch (<Ausdruck>) {
      case <konst_Ausdruck1>:
           <Anweisungen...>
           break;
      case <konst_Ausdruck1>:
           <Anweisungen...>
           break;
      default:
           <Anweisungen...>
}
```

Der Ausdruck muß ganzzahlig oder ein Zeichen (Byte) sein.

```
goto (<Label1>,<Label2>,...<LabelN>) Ausdruck
```

Der Ausdruck muß ganzzahlig sein. Wenn er gleich 1,2,..N ist, wird zum ersten, zweiten,..Nten Label verzweigt.

6.3 Schleifen

```
while ( <Ausdruck> ) <Anweisung oder Block>

do <Anweisung oder Ausdruck> while (<Ausdruck>)

for ( <Initialisierung>;<Test>;<Update> )
<Anweisung oder Block>
```

```
do <label>,<var>=<Start>,<Ende>
      <Anweisungen>
<label> continue
```

In C kann es bei den Varianten `while` und `for` sein, daß der Schleifenkörper nie abgearbeitet wird, bei der `do...while`-Version wird der Schleifenkörper auf jeden Fall einmal durchlaufen.

Eine `for`-Schleife `for(i=0;i<10;i++){...}` wird abgearbeitet als:

- Initialisierung i=0
- Test i<10? Wenn ja, Abarbeitung des Schleifenkörpers `{...}`, sonst Ende der `for`-Schleife.

- Update i=i+1
- Test, Schleifenkörper, Update, Test, Schleifenkörper,...,Test

Natürlich kann man das Update auch als letzten Befehl in den Schleifenkörper schreiben:
`for(i=0;i<10;){...; i++;}`.
Man kann die Initialisierung vorher machen: `i=0; for(;i<10;){...; i++;}`
und das ist äquivalent zu `i=0; while(i<10){...; i++;}`

Laut Fortran77-Standard wird eine do-Schleife nicht durchlaufen, wenn der Startwert größer als der Endwert ist. Es gibt jedoch sehr alte Compiler, die den Test erst am Ende der Schleife durchführen, so daß die Schleife immer mindestens einmal durchlaufen wird (und sehr alten Fortran-Code, der dieses Verhalten erwartet). Es gibt auch die Möglichkeit, eine Schrittweite ungleich 1 anzugeben:
`do <label>, <var>=<Start>,<Ende>,<Schrittweite>`
Diese kann auch negativ sein.

Im Unterschied zu C wird die Anzahl der Schleifendurchläufe am Anfang des Loops aus den Angaben `<Start>,<Ende>` und evtl. `<Schrittweite>` berechnet. Zuweisungen an den Schleifenzähler innerhalb des Do-Blocks haben keine Auswirkung auf die Anzahl der Durchläufe.

Viele Fortran-Compiler erlauben die Fortran90-Syntax

```
do i=1,100
 ...
end do
```

6.4 Sprunganweisungen

Sowohl C als auch Fortran kennen die unbedingte Verzweigung `goto <label>`. Wie bei allen Verzweigungen muß das Ziel innerhalb derselben Funktion liegen.

Innerhalb von Schleifen kennt C noch `break` und `continue`. Ersteres führt zum Verlassen der innersten Schleife, letzteres zum sofortigen Start der nächsten Iteration:

```
 for(i=0; i<100; i++) {
     y = func(2*i);
     if(y > 2) break;     /* Schleife wird vorzeitig abgebrochen */
     if(a[i]<0) continue; /* Start der nächsten Schleifeniteration */
     a[i]=sqrt(a[i]);
}
```

7 Arithmetische Operatoren

Wenn die beiden Operanden der binären arithmetischen Operatoren (`+,-,*,/`) nicht vom gleichen Typ sind, wird ein Operand umgewandelt, wobei die Umwandlung immer in Richtung zum längeren Datentyp erfolgt (z.B. `char -> int, int -> float, float -> double`). Das Ergebnis hat denselben Typ wie die beiden Operanden. Bei einer Zuweisung erfolgt anschließend wenn nötig die Anpassung an den Datentyp der linken Seite. Dies kann also auch eine Abwärtskonversion sein.

Multiplikation/Division bindet natürlich stärker als Addition/Subtraktion, ansonsten sind die Operationen linksassoziativ: `a / b / c = (a / b) / c`. Ganzzahlige Division liefert eine ganze Zahl als Ergebnis, der gebrochene Anteil wird abgeschnitten. Damit ist
`3 / 4 * 5.0 = 0 * 5.0 = 0.0` und `5.0 * 3 / 4 = 15.0 / 4 = 3.75`.

Rest der ganzzahligen Division:
`a % b`

Potenz: `pow(a,b)`
ist äquivalent zu `exp(b*log(a))`

Die mathematischen Funktionen `sqrt, log, log10, exp, sin, cos, tan, asin, sinh, ...` sind alle vom Typ `double f(double)`. Ihre Deklarationen sind in `math.h` zu finden.

Die Betragsfunktion `abs` ist vom Typ `int abs(int)`! Für Fließkommazahlen muß man `fabs` verwenden.

Rest der ganzzahligen Division:
`mod(a, b)`

Potenz: `a**b`
für kleine ganzzahlige b wird dies vom Compiler in Multiplikationen umgewandelt, ansonsten ist es equivalent zu `exp(b*log(a))`.

Die mathematischen Funktionen `abs, sqrt, log, log10, exp, sin, cos, tan, asin, sinh, ...` sind sogenannte generische Funktionen: der Typ des Ergebnisses ist gleich dem Typ des Arguments. Manchmal findet man auch die spezifischen Formen, z.B. `dsqrt, csqrt` für die `double precision` oder die `complex`-Version der Quadratwurzel.

8 Logische Operatoren und Vergleichsoperatoren

In C kann der Wert jedes Ausdrucks logisch interpretiert werden: `0 (int), 0.0 (float)` oder `\0 (char)` bedeuten FALSE, alles andere wird als TRUE interpretiert.
Umgekehrt: jeder Vergleichsausdruck hat den numerischen Wert 0 oder 1, damit sind Konstruktionen wie `x + (y >= 1 )` möglich.
Vergleiche: `>, <, >=, >=, ==, !=`
Logische Operatoren: `||, &&, !`

Vergleiche erzeugen Werte vom Typ `logical`, Argumente von `if()` müssen von diesem Typ sein:

```
logical isodd
isodd = mod(x,2) .eq. 1
if(isodd) then ...
```

Vergleiche: `.gt., .lt., .ge., .le., .eq., .ne.`
Logische Operatoren: `.or., .and., .not., .eqv.` (logische Gleichheit), `.neqv.` (logisch ungleich, exklusives oder)

Logische Operatoren werden von links nach rechts ausgewertet, allerdings nur solange, bis ihr Wahrheitswert feststeht. Das spart Rechenzeit – und ermöglicht in C Konstrukte wie
`(x >= 0) && (y = sqrt(x))`

9 Einige Besonderheiten in C

Eine Zuweisung `var = Ausdruck` ist in C ebenfalls ein Ausdruck, sein Wert und Typ ist der Wert/Typ der linken Seite. Damit sind Mehrfachzuweisungen möglich, wobei = rechtsassoziativ ist:
`a = b = c = d` wird ausgewertet als `a = ( b = ( c = d ) )`
(Achtung! `int a; float b,c; b=a=c=3.14;` setzt b=3.0!)

Ausdrücke der Form `x = x • y` können als `x •= y` geschrieben werden.

Die Zuweisungen `i=i+1` und `i=i-1` können auch als `++i` oder `--i` geschrieben werden. Die Postfix-Form `i++` oder `i--` hat die gleiche Wirkung auf `i`, ihr **Wert** als Ausdruck ist jedoch `i` und nicht `i+/-1`. Anders gesagt, `y=++x;` ist äquivalent zu `x=x+1; y=x;` Die Variante `y=x++;` jedoch ist äquivalent zu `y=x; x=x+1;`.

Zwei durch Komma getrennte Ausdrücke werden von links nach rechts ausgewertet. Typ und Wert des **rechten** Ausdrucks bestimmen Typ und Wert des Gesamtausdrucks.

Diese Regeln erlauben in C sehr kompakte Schreibweisen. Hier ein Beispiel aus dem Code des PostScript-Interpreters `ghostscript`:

```
ccase = (skew > 0 ? copy_right : ((bptr += chunk_bytes), copy_left)) + function;
```

"Auseinandergenommen" kann man das auch so schreiben:

```
if(skew > 0) {
      ccase = copy_right + function;
} else {
      bptr = bptr + chunk_bytes;
      ccase = copy_left + function;
}
```

9.1 Zeiger

Zeiger enthalten die Adresse eines Objekts (sie "zeigen" darauf).

```
char * cx;         /* Zeiger auf ein Objekt vom Typ char */
int  * ip;         /* Zeiger auf eine ganze Zahl */
double * fx[10];   /* Feld von 10 Zeigern auf Double-Zahlen */
char * p="Hans";   /* Zeiger auf char mit Initialisierung */
double ** fz;      /* Zeiger auf einen Zeiger auf eine Double-Zahl */
void * p;          /* generischer Zeiger */
```

Der Typ `void` dient a) zur Deklaration generischer Zeiger und b) zur Deklaration von Funktionen ohne Rückgabewert: `void swap(int a, int b){ ...}`

Der `&`-Operator liefert die Adresse eines Objekts; der `*`-Operator, angewandt auf einen Zeiger, liefert das Objekt, auf das der Zeiger zeigt:

```
int i, *ip, vec[9]; /* ip ist ein Zeiger auf eine ganze Zahl */

ip   = &i;         /* Jetzt hat ip einen Wert: er zeigt auf i */
*ip += 3;          /* Damit wird i (das, worauf ip gerade zeigt) um 3 erhöht. */
j = 2*(*ip);       /* äquivalent zu j=2*i */
ip   = &vec[3];    /* Jetzt zeigt ip auf das 4. Element von vec. */
ip--;              /* Jetzt zeigt ip auf das 3. Element von vec. */
*ip  = 10;         /* vec[2] hat nun den Wert 10. */
```

Zeiger können miteinander verglichen oder voneinander subtrahiert werden. Weiterhin können ganze Zahlen addiert oder subtrahiert werden. Zeigerarithmetik geschieht dabei immer in Speichereinheiten des Types, auf den der Zeiger zeigt: `ipn = ip + 1` zeigt auf das nächste Feldelement, d.h. es wird z.B. um 4 Bytes weitergezählt.

9.2 Dynamische Speicherverwaltung

Die Existenz von Zeigern erlaubt es, Speicherplatz dynamisch zur Laufzeit des Programms (z.B. in Abhängigkeit von den Eingabedaten) anzufordern. Die Funktion `malloc()` nimmt als Argument die Anzahl der geforderten Bytes und liefert einen `void`-Zeiger auf einen entsprechenden Speicherbereich zurück. Durch Aufruf der Funktion `free()` mit diesem Zeiger als Argument wird der Speicherbereich dem Betriebssytem zurückgegeben. Die entsprechenden Prototypen sind in `stdlib.h` definiert. Der `sizeof()`-Operator liefert die Größe eines Objekts in Bytes.

```
#include <stdlib.h>
     double * ff, *g;              /* Zeiger auf double-Objekte */
     int n, i;
     printf("Bitte Anzahl der Daten eingeben\n");
     scanf("%i", &n);              /* n enthält jetzt die Anzahl der Daten */
```

```
ff = malloc(n*sizeof(double)); /* Ein Feld der entsprechenden Größe wird
                       angefordert und seine Adresse in ff gespeichert */
g = ff;
for(i=0; i<n; i++) {
    scanf("%lf", g);    /* Lese Zahl ein und speichere sie an Position g */
    g++;                /* g zeigt auf nächsten freien Platz */
}
...
free(ff);
```

9.3 Das Hauptprogramm

- `main` ist vom Typ `int`. Daher muß es mit einer Anweisung enden, die eine ganze Zahl zurückgibt. Dieser Rückgabewert ist der "exit code" für das Betriebssystems und normalerweise wird 0 verwendet, um ein fehlerfreies Ende des Programms zu signalisieren:

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

Das Hauptprogramm bekommmt auch Argumente übergeben, die bei Bedarf ausgewertet werden können. Das erste Argument von `main` ist eine ganze Zahl, die Anzahl der Zeichenketten in der Kommandozeile. Das zweite Argument ist ein Feld von Zeigern auf Zeichenketten: die einzelnen "Worte" der Kommandozeile. Der letzte Zeiger des Feldes ist der Nullzeiger. Üblicherweise wird `main` so deklariert:

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

Wenn z.B. das Programm "my_solve" heißt und mit der Kommandozeile

```
my_solve datei1 -n 100
```

aufgerufen wird, dann ist

```
argc = 4
argv[0] = "my_solve"
argv[1] = "datei1"
argv[2] = "-p"
argv[3] = "100"
argv[4] = 0
```

10 Ein- und Ausgabe

Dateien werden geöffnet, gelesen oder/und geschrieben und wieder geschlossen.

Öffnen einer Datei:

```
#include <stdio.h>
/* Deklarationen der Standard-
   Input/Output Funktionen,
   des Types FILE usw. */
...
FILE * fp1, * fp2, * fp3;
/* fp1,... sind Zeiger auf
   spezielle FILE-Objekte */

fp1 = fopen("beispiel.dat", "r");
/* Die Datei beispiel.dat
   wird zum Lesen geöffnet */

fp2 = fopen("beispiel2.dat", "w");
/* Die Datei wird zum Schreiben
   (durch Neuanlegen oder
   Überschreiben) geöffnet */

fp3 = fopen("beispiel3.dat", "a");
/* Die Datei wird zum anhängendem
   Schreiben geöffnet */
```

```
   open(unit=4, file='bsp.dat', status='old')
c Kanal 4 wird mit der Datei bsp.dat verbunden,
c welche bereits existieren muss.

  open(17, file='bsp2.dat', status='new', err=12)
c Kanal 17 wird mit bsp2.dat verbunden, sie darf
c nicht existieren. Wenn dabei ein Fehler
c auftritt, wird "goto 12" ausgeführt.

  open(19,file='b3',status='unknown',access='append')
c Wenn b3 schon existiert, wird beim Schreiben
c angehängt
```

Der Dateiname ist eine Zeichenkette, kann also auch eine Variable vom Typ Zeichenkette sein. In C gibt `fopen()` einen Nullzeiger zurück, falls ein Fehler beim Öffnen auftritt.

Lesen aus einer offenen Datei und Schließen der Datei:

```
fscanf(fp1, <Format>, <Zeiger>...);
fclose(fp1);
```

```
read(4,*) <var1>,<var2>,...
close(4)
```

Ein Spezialfall sind die Standardeingabe, Standardausgabe und Fehlerausgabe-Kanäle (`stdin, stdout, stderr` im C-Jargon): diese stellt das Betriebssystem als geöffnete Kanäle zur Verfügung und sie dienen zur interaktiven Ein- und Ausgabe[3].

Verwendung von `scanf()` und `printf()` (analog zu `fscanf() und fprintf()`, nur ohne das erste Argument) liest von der Standardeingabe und schreibt auf die Standardausgabe.
`fprintf(stderr,...)` schreibt auf die Fehlerausgabe.

Verwendung eines `*` anstelle der Kanalnummer:

```
read(*, *) a,b,c
write(*, 23) a,b
```

verwendet die Standardeingabe/ausgabe-Kanäle.

[3]Unter der Unix-Shell sind alle drei Kanäle mit dem Terminal verbunden, in dem man das Programm startet. Die Umleitungsoperatoren können zur Trennung von `stdout` und `stderr` genutzt werden:
`meinprogramm < ein.dat > aus.dat 2> err.dat` liest als Standardeingabe Daten aus `ein.dat`, Ausgaben auf den Standardausgabe-Kanal werden nach `aus.dat` geschrieben und Ausgaben auf den Fehlerausgabe-Kanal in die Datei `err.dat`.

Formatierung von Ein- und Ausgabe

- Die Funktionen `(f)scanf, (f)printf` haben als 1.(2.) Argument eine Zeichenkette, die Formatspezifikationen enthält.
- Weitere Argumente von `(f)printf` sind die auszugebenden Variablen, bei `(f)scanf` Zeiger auf die einzulesenden Variablen:

```
double a;
float z[10][10];
int n;
fscanf(fp1,"%lf %i %f",
            &a, &n, &z[2][3]);
printf("a= %12.5g n= %8i\n", a, n);
printf(" z[4,5]=%g\n", z[4][5]);
```

- Die Funktionen `read, write` haben (neben der Kanalnummer als erstem Argument) als zweites Argument das Label einer `Format`-Anweisung.
- Nach den beiden Argumenten in Klammern folgt eine Liste der zu lesenden/zu schreibenden Variablen.
- Als 2. Argument kann auch ein `*` verwendet werden. Damit werden beim Einlesen die Variablen entsprechend ihres Typs gelesen, zum Ausgeben wird ein Standardformat verwendet. Insbesondere zum Einlesen dürfte das für die meisten Zwecke ausreichen.

```
      write(3,*) 'a=', a,' b_10= ', b(10)
      read(4,*) x,y
      write(3,21) i,x,y
   21 format(' i= ',i8,/,f12.5,g16.6)
```

Im Folgenden werden nur die einfachsten Formatierungsregeln erwähnt. Für die umfangreichen Details sei auf die Dokumentation der C-Funktionen (`"man 3 fprintf"`, `"man 3 fscanf"`) und z.B. auf Kapitel 10.6–10.11 im *Professional Programmer's Guide to Fortran77* verwiesen.

- Der Formatstring kann Zeichen enthalten, die direkt ausgegeben werden.
- Wichtige Formatangaben sind `%i,%s,%e,%f,%g,%le,%lf,%lg` für ganze Zahlen, Zeichenketten, `floats` mit oder ohne abgetrenntem Exponenten (das `%g`-Format wählt je nach Größe der Zahl das `%e` oder `%f`-Format) und `doubles`.
- Die Formate können auch Längenangaben enthalten: `%10f, %10.4f` Die erste Zahl ist die Formatbreite, die zweite die Anzahl der Nachkommastellen.
- Beim Lesen unterdrückt ein `*` die Zuweisung: `scanf("%i %*s %i",&m,&n)` kann Zeilen der Form `"23 blabla 45"` einlesen.

- Formatangaben haben die Form `Iw, Ew.d, Fw.d, Gw.d, A, Aw` oder `Lw`. Dabei ist `w` die Formatbreite, `d` die Anzahl der Nachkommastellen. `I` gibt eine ganze Zahl aus, `F` eine Fließkommazahl, `E` eine Fließkommazahl mit abgetrenntem Exponenten und `G` wählt, je nach Größe der Zahl, `E` oder `F`. Das `A`-Format ist für Zeichen und Zeichenketten, das `L`-Format für boolesche Variablen.
- Ein Bruchstrich `/` erzeugt einen Zeilenvorschub analog zu `\n` in C.
- Formate sollten immer mit einem Leerzeichen beginnen, da manche Fortran-Implementationen ein nichtleeres Zeichen am Anfang einer Zeile speziell interpretieren.

Fehlerbehandlung:

`(f)scanf` meldet die Zahl der gelesenen Variablen zurück. Dies können weniger als erwartet sein, wenn beim Lesen/Konvertieren ein Fehler aufgetreten ist.

```
n = scanf("%i %i %i",i,j,k);
if(n<3) {
  /* Beim Lesen ist ein
    Fehler aufgetreten */
...
}
```

`read` kann als weiteres Argument ein Label haben, zu dem im Falle eines Lesefehlers verzweigt wird:

```
      read(*,*,err=33) i,j,k
      ...
   33 write(*,*) ' Lesefehler!'
```

11 Sichtbarkeit (scope), Lebensdauer und Initialisierung von Variablen

- **Lokale Variablen** sind Variablen, die am Anfang von Funktionen (C: oder Blöcken) deklariert werden. Sie sind nur in dieser Funktion/Block sichtbar. Sie behalten ihren Wert **nicht** zwischen zwei Funktionsaufrufen. Ihr Anfangswert bei jedem Funktionsaufruf ist (falls sie nicht explizit initialisiert werden) undefiniert.

Variablen können schon bei ihrer Deklaration mit Konstanten initialisiert werden:

```
int f1(int y){
    int x=1;
    double pi = 3.14159;
    int feld1[5] = {15,2,4,4,4};
    int * ip = &x;
    char str[]="Haha";  /* Compiler ermittelt Länge selbst */
    ...
}
```

Diese Zuweisungen erfolgen bei jedem Aufruf von `f1()` neu.

Deklaration und Initialisierung lassen sich nicht kombinieren:

```
integer i
real*8 m
...
i=1
m=2.3
```

- **Statische Variablen** behalten ihren Wert zwischen zwei Funktionsaufrufen:

```
int f2(int x){
   static int j;
   static float g;
   ...
   j++;
   printf("f2 called %i times",j);
}
```

Statische Variablen werden beim Programmstart mit 0 initialisiert. Wird ein expliziter Initialisierungswert angegeben:

```
static double g=3.3;
```

so wird auch diese Initialisierung nur einmal beim Programmstart durchgeführt.

```
integer function f1(y)
implicit none
real pi, g
integer feld1(5)
character str*4
data pi/3.14159/, feld1/15,2,3*4/
data str/'Haha'/
save g
...
```

Eine Variable wird statisch durch die `save`-Deklaration oder indem sie mit einer `data`-Deklaration initialisiert wird. Auch diese Initialisierung erfolgt einmalig beim Programmstart. Die `data`-Anweisung erlaubt einige praktische Tricks zur Initialisierung von Feldern:

```
real feld(1000)
data feld/500*1.0, 500*2.0/
```

setzt die ersten 500 Elemente des Felds gleich 1, die anderen gleich 2.

- **Globale Variablen** sind in mehreren Funktionen sichtbar. Damit sind sie neben der Parameterübergabe durch Funktionsargumente eine Methode des Datenaustausches oder gemeinsamen Datenzugriffs verschiedener Funktionen.

Eine Deklaration/Definition außerhalb von Funktionen (typischerweise am Dateianfang nach `#include`-Anweisungen) ist global. Globale Variablen sind immer statisch.
Eine globale Definition (d.h., eine Definition außerhalb von Funktionsblöcken) kann ebenfalls das Attribut `static` enthalten, welches bei globalen Variablen (die sowieso statisch sind) allerdings eine andere Bedeutung hat: es schränkt die Sichtbarkeit auf die Quelltextdatei ein.

```
int k=1;
/* eventuell auch in anderen
Quelltextdateien verwendbar */
static int m=2;
/* nur in dieser Datei sichtbar */
```

Um von Funktionen in verschiedenen Quelltextdateien aus auf eine globale Variable zugreifen zu können, muß sie in jeder dieser Dateien deklariert werden: `int k;` Es empfiehlt sich, solche Deklarationen in eine .h-Datei zu schreiben, die dann mittels `#include` überall eingebunden wird. Genau eine Datei kann dann eine zusätzliche Initialisierung enthalten: `int k=1;`
Wenn in einem Block eine lokale Variable gleichen Namens definiert wird, verdeckt sie die globale Variable. Auf letztere kann dann in diesem Block nicht mehr zugegriffen werden.

Globale Variablen werden durch `common`-Blöcke vereinbart.
Ein `common`-Block besteht aus einem Namen und einer Liste von Variablen. Die entsprechende Anweisung (und die Deklaration der Variablen) ist in jede Funktion aufzunehmen, die auf diese globalen Variablen zugreifen soll.

```
common /block1/ a,b,feld(10)
integer a,b
double precision feld
```

Der entsprechende Speicherbereich ist durch den Namen des `common`-Blocks charakterisiert, die Namen der Variablen können in einer anderen Funktion andere sein, z.B. kann in `function f1` stehen

```
common /bl/a,b,c
```

und in `function f2`:

```
common /bl/a1,a2,a3
```

Dann sind `a` und `a1`, `b` und `a2` etc. nur verschiedene lokale Namen für dieselben drei globalen Variablen.
Seltsamerweise sind `common`-Blöcke lt. Fortran-Standard nicht unbedingt statisch. Der einfachste Weg, sie statisch zu machen, ist, ihre Deklaration mit ins Hauptprogramm (`program ...`) aufzunehmen.

- **Konstanten:** Variablen können als konstant deklariert werden. Jeder Versuch, diesen Variablen nach der Initialisierung einen anderen Wert zuzuweisen, führt zu einer Compiler-Fehlermeldung.

```
const int N=100;
const double Pi=3.14159;
const char * p;
/* Zeiger zu einem konstanten Zeichen */
char * const p2;
/* konstanter Zeiger */
```

Konstanten werden durch eine `parameter`-Anweisung vereinbart und initialisiert:

```
integer N
double precision Pi
parameter (N=100, Pi=3.14159)
```

12 Funktionen

12.1 Funktionen mit Rückgabewert

Die Funktion muß mindestens eine Anweisung `return <Ausdruck>` enthalten:
Datei 1:

```
/* Prototyp */
int timesec(int, int, int);
...
int main(){
...
   seconds = timesec(hours, mins, secs);
...
}
```

Datei 2:

```
int timesec(int h, int m, int s){
    int x;
    if(s<0) return -1;
    x = 60*((60*h)+m)+s;
    return x;
}
```

In der Funktion wird einer Variablen gleichen Namens der Rückgabewert zugewiesen. `return` ohne Argument kann zur Ablaufsteuerung verwendet werden.

```
program main2
implicit none
integer timesec
...
seconds = timesec(hours, mins, secs)
...
end

integer function timesec(h, m, s)
implicit none
integer h,m,s
if(s.lt.0) then
   timesec=-1
   return
endif
timesec=60*((60*h)+m)+s
end
```

12.2 Funktionen ohne Rückgabewert

```
/* Prototyp */
void timesec(int*, int, int, int);
...
int main(){
...
   timesec(&seconds, hours, mins, secs);
...
}

void timesec(int *x, int h, int m, int s){
    if(s<0) {*x=-1; return;}
    *x = 60*((60*h)+m)+s;
}
```

Funktionen ohne Rückgabewert heißen `subroutinen`. Sie müssen mit `call` aufgerufen werden.

```
program main2
implicit none
...
call timesec(seconds, hours, mins, secs)
...
end

subroutine timesec(x, h, m, s)
implicit none
integer h,m,s,x
if(s.lt.0) then
   x=-1
   return
endif
x=60*((60*h)+m)+s
end
```

12.3 Argumentübergabe

Das letzte Beispiel illustriert einen fundamentalen Unterschied zwischen Fortran und C: In Fortran erfolgt die Parameterübergabe durch "call by reference". Wenn eine gerufene Funktion ihre Argumente verändert (wie die Variable `x` im letzten Beispiel), so ist diese Änderung anschließend auch im rufenden Programm (Variable `seconds`) wirksam.

In C erfolgt Parameterübergabe prinzipiell durch "call by value", die gerufene Funktion bekommt nur eine Kopie des Wertes mitgeteilt. Eine Zuweisung an ein Argument einer Funktion hat daher keinerlei Auswirkung auf die Variablen im rufenden Programm. "call by reference" muß mit Zeigern simuliert werden: Im C-Beispiel bekommt `void timesec(...)` einen Zeiger auf die Variable `seconds` übergeben. Eine Änderung oder Neuzuweisung dieses Zeigers hätte ebenfalls wieder keine Auswirkungen auf das rufende Programm. Aber der Zeiger wird genutzt, um die Speicherzelle zu ändern, auf die er zeigt (`*x=...`) – und dies ist gerade die Variable `seconds` des rufenden Programms.

Dieses Verhalten ist auch der Grund dafür, daß die Funktionen `scanf(), fscanf(),...` Zeiger auf die einzulesenden Variablen als Argumente übermittelt bekommen müssen – sie sollen ja den Wert dieser Variablen setzen.

Die Tatsache, daß C von jedem Funktionsargument erst eine lokale Kopie für die gerufene Funktion herstellt, sollte beachtet werden, wenn man z.B. umfangreiche `struct`s an Funktionen übergeben will. Hier kann die Verwendung von Zeigern wesentlich effizienter sein.

Eine Besonderheit stellen Felder als Argumente dar: hier übergibt C automatisch nur einen Zeiger (und keine Kopie des ganzen Feldes), s. übernächster Abschnitt.

13 Funktionen als Argumente von Funktionen

Manchmal möchte man einer Funktion den Namen einer anderen Funktion übergeben, die von dieser aufgerufen werden soll. Beispiele sind ein Sortieralgorithmus, dem man neben den zu sortierenden Daten auch eine Vergleichsfunktion übergibt oder eine Routine zur numerischen Integration.

C kennt neben Zeigern auf andere Objekte auch Zeiger auf Funktionen. Diese kann man an Unterprogramme übergeben. Im Unterprogramm kann man diesen Zeiger wie eine Funktion verwenden, man muß ihn nicht mit `*` dereferenzieren. Etwas Übung benötigt die Syntax der Typdeklarationen:

- `fp1` sei Zeiger auf eine Funktion vom Typ int f(int,int):
  `int (*fp1)(int, int);`
- `fun` sei eine Funktion, deren erstes Argument ein solcher Zeiger und das zweite Argument ein `int` ist:
  `int fun( int (*p)(int, int), int);`
- `fp1` wird ein Wert zugewiesen: die Adresse einer Funktion func2 vom entsprechenden Typ:
  `fp1 = &func2;`

Sei als Beispiel `intgr` eine primitive numerische Integrationsroutine (Trapezregel, 100 Stützstellen):

```
/* Prototypen */
float bsp(float);
float intgr(float (*f)(float), float, float);

int main(){
   ...
   x=intgr(bsp, 0, 1);
   ...
}

float bsp(float x){
    return 2*x*x;
}

float intgr(float (*fp)(float),
            float x0, float x1){
     ...

     y = fp(x0+ (x1-x0)*i/100.);
     ...
}
```

Funktions- und Subroutine-Namen können einfach als Argumente übergeben werden. Wenn in einem Programmteil nicht erkennbar ist, daß es sich um Funktionen handelt (weil sie nicht aufgerufen, nur weitergegeben werden), sind sie als `external` zu deklarieren, zum Sprachumfang von Fortran gehörende Funktionen als `intrinsic`.

```
  program trapezregel
  implicit none
  intrinsic sin, tan
  external bsp
  ...
  x1 = intgr(sin, 0, 3.14)
  x2 = intgr(tan, 0, 3.14)
  x3 = intgr(bsp, 0, 2)
  ...
  end

  real function bsp(x)
  implicit none
  real x
  bsp = cos(x) + sqrt(x)
  end

  real function intgr(myfunc, x0, x1)
  implicit none
  real myfunc, x0, x1
  do 7, i=1,100
     ...
     y = myfunc( x0 + (x1-x0)*i/100. )
     ...
7 continue
  ...
  end
```

14 Felder als Argumente von Funktionen

Eine Standardsituation in der Numerik ist, daß man einer Routine (die z.B. Eigenwerte berechnet) eine Matrix beliebiger Größe übergeben möchte. Dies wird von Fortran direkt unterstützt, von C leider nicht.

14.1 Felder in Fortran

- Eine Funktion/Subroutine kann ein Feld beliebiger Größe gemeinsam mit den Feldgrößen als Argument übergeben bekommen:

```
      double precision function trace(mat, n)
      double precision mat(n,n)
c Deklaration mit erst zur Laufzeit bekannter Größe
      implicit none
      integer n,i
      trace=0
      do 1 i=1,n
      trace = trace + mat(i,i)
 1    continue
      end
```

  Die Feldgröße kann auch mit Hilfe eines `common`-Blocks übergeben werden. Die Definition eines lokalen Feldes variabler Größe

```
      subroutine bsp(n)
      float mat(n)
      ...
```

  ist eine oft (z.B. von `g77`) unterstützte Fortran90-Erweiterung ("automatic arrays").
- Ein- und mehrdimensionale Felder sind immer in einem zusammenhängenden Speicherbereich abgespeichert. Das Element `a(i,j,k)` einer $M \times N \times L$-Matrix steht an Position `(k-1)*M*N+(j-1)*M+i` (Zählung beginnt mit 1. Die Formel ist entsprechend zu modifizieren, wenn auch untere Indexgrenzen – `a(M0:M1,N0:N1,K0:K1)` – vereinbart sind.) Wie man sieht, muß das Unterprogramm die letzte Dimension `L` der übergebenen Matrix gar nicht wissen, um ihre Elemente korrekt adressieren zu können:

```
      subroutine bsp(mat, vec, n)
      implicit none
      integer n
      double precision mat(n,*), vec(*)
      ...
      mat(i,j) = x * vec(i+2)
      ...
```

  Man ist dabei natürlich selbst dafür verantwortlich, daß man in `do`-Schleifen etc. nicht über die Feldgrenzen hinausläuft.
- Man kann die Idee "ein Feld ist nichts als ein zusammenhängender Speicherblock" in Fortran noch weiter ausnutzen: Felder können in rufender und gerufener Funktion verschieden deklariert sein und man kann auch ein beliebiges Feldelement anstelle des Feldes übergeben. Dies wird dann vom Unterprogramm als der Anfang des Speicherbereiches angesehen:

```
      program confus
      implicit none
      double x(-3:50), y(1000), z(100,20)
      call bss(x)
      call bss(y(7))
```

```
      call bss(z(1,15))
      end

      subroutine bss(x)
      double x(50)
      do 1,i=1,50
      x(i)=2.*i
    1 continue
      end
```

Beim ersten Aufruf setzt `bss` die Elemente `x(-3)...x(46)` des rufenden Programms, beim zweiten Aufruf `y(7)...y(56)` und beim 3. Aufruf `z(1,15)...z(50,15)`.

14.2 Felder in C

- Ein- und mehrdimensionale Felder sind immer in einem zusammenhängenden, lückenlosen Speicherbereich abgespeichert.
- Die Indexsyntax `a[i][j]` ist in C sowohl für Zeiger als auch für Felder anwendbar:
  - Wenn `a` ein Feld ist, deklariert z.B. als `double a[5][9]`, dann übersetzt der Compiler einen Zugriff auf `a[i][j]` zu: addiere zur (Anfangs)adresse von `a` neunmal `i` (das ist dann der Anfang der i-ten Zeile), anschließend addiere `j`, liefere den Wert, der an dieser Adresse steht.
  - Sei `a` deklariert als `double **a` (Zeiger auf Zeiger auf Zahlen). Der Zugriff auf `a[i][j]` bedeutet nun: Addiere `i` zur Adresse von `a`; lese die Adresse, die an dieser Adresse steht; addiere `j` zu dieser Adressse, liefere den Wert, der dort steht.
- In C gehört die Feldgröße zu seiner Typdeklaration und muß zur Compile-Zeit bekannt sein. Etwas analoges zu Fortran

  ```
  double trace(double mat[n][n], int n){
      /* Illegale Syntax! Feldgrößen müssen zur Compile-Zeit auswertbare
         konstante Ausdrücke sein. */
  ...
  }
  ```

  ist **nicht** möglich.
- Anstelle von Feldern wird "in Wirklichkeit" nur ein Zeiger auf das erste Element übergeben.
- Felder definierter Größe können an Funktionen übergeben werden:

  ```
  void bsp2(double mat[2][2]){ ... }
  ```

- Ein Unterprogramm muß die erste Dimension eines übergebenen Feldes gar nicht wissen, um seine Elemente korrekt adressieren zu können:

  ```
  void  bsp3(double mat[][10], vec[]){ ... }
  ```

- Dies löst allerdings nicht eine Reihe von Problemen. Oft ist es in einem Algorithmus nötig, für Zwischenergebnisse weitere Matrizen von der Größe der Input-Matrix bereitzustellen. Und schon bei einer quadratischen Matrix unbekannter Größe weiß man weder Zeilen- noch Spaltenzahl. Es ist daher oft sinnvoller, auf Felder zu verzichten und ausschließlich Zeiger zu verwenden.
- Wenn man einen Zeiger definiert, wird nur Speicherplatz für den Zeiger selbst zur Verfügung gestellt. Den Speicherbereich für die Daten muß man selbst mit `malloc()` oder `calloc()` bereitstellen.
- Verwendung von Zeigern statt eindimensionaler Felder:

```
#include <stdlib.h>
double skalarprodukt(double *, double *, int);
```

```
int main(){
    double *a, *b, res;
    int i,n;

    ...  /* n wird gesetzt, eigelesen, o.ä */

    /* a und b sollen auf Vektoren der Länge n zeigen */
    a = malloc(n*sizeof(double));
    b = malloc(n*sizeof(double));

    for(i=0; i<n; i++) {
       a[i] = ...
       b[i] = ...
    }

    res = skalarprodukt(a, b, n);
    ...
    free(a); free(b);
    return 0;
}

double skalarprodukt(double *x, double *y, int size){
    double res=0;
    int i;
    for(i=0;i<size;i++) res += x[i]*y[i];
    return res;
}
```

- Zweidimensionale (und analog höherdimensionale) Felder kann man zweckmäßigerweise als Zeiger auf Zeiger auf Zahlen implementieren: `double **a` ist die Adresse eines Feldes von Zeigern. Jeder Zeiger in diesem Feld zeigt auf eine Zeile der Matrix. Die Anforderung und Freigabe des benötigten Speicherplatzes erledigt man am Besten durch zwei kleine Hilfsfunktionen:

```
double ** alloc_matrix(int n, int m){
/* Stelle Speicherplatz für eine nxm-Matrix bereit */
    int i;
    double **mat;
    /* Speicherplatz für die Zeiger auf Zeilen */
    mat = malloc(n*sizeof(double*));
    /* zusamenhängender Speicherplatz für alle Zeilen */
    mat[0] = malloc(n*m*sizeof(double));
    /* Eintrag der Zeiger auf die Zeilen */
    for(i=1; i<n; i++) mat[i] = mat[i-1] + m;
    return mat;
}

void free_matrix(double ** mat){
    free(mat[0]); /* Speicherplatz für die Daten */
    free(mat);    /* Speicherplatz für die Zeiger */
}
```

Eine kleine Anwendung zur Illustration:
`void matmult(double **a, double **b, double **c, int n, int m)` multipliziert die $n \times m$-Matrix a mit der $m \times n$-Matrix b, Ergebnis ist die $n \times n$-Matrix c. Der Platz für c muß vom rufenden Programm bereitgestellt werden.

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

void matmult(double**, double**, double**, int , int );
double ** alloc_matrix(int, int);
void free_matrix(double **);

int main(){
    double **x,**y,**z;
    int i,j;
    x=alloc_matrix(2,3);      /* Allokiere Speicherplatz */
    y=alloc_matrix(3,2);
    z=alloc_matrix(2,2);

    for(i=0;i<2;i++)         /* Fülle Matrizen mit Werten */
        for(j=0;j<3;j++) {
            x[i][j] = i+j;
            y[j][i] = i*j;
        }

    matmult(x,y,z,2,3);

    for(i=0;i<2;i++)         /* Drucke Ergebnis */
        for(j=0;j<2;j++)
            printf(" z[%i][%i]=%e\n",i,j,z[i][j]);

    free_matrix(x); free_matrix(y); free_matrix(z);
    return 0;
}

void matmult(double **a, double **b, double **c, int n, int m){
    int i,j,k;
    double sum;

    for(i=0; i<n; i++)
        for(j=0; j<n; j++) {
            sum=0;
            for(k=0; k<m; k++) sum += a[i][k]*b[k][j];
            c[i][j] = sum;
        }
}
```

15 Was fehlt

Zeichenketten-Funktionen, Bitmanipulationen,
C: typedef, enum, register, variable-length argument lists
Fortran: statement functions, block data, entry, unformatted & direct access I/O
(Und einiges fehlt auch hier...)

16 Guter Programmierstil

- Die Implementationen eines Algorithmus einerseits und der erforderlichen Ein/Ausgabe- und Steuerroutinen andererseits sollen klar getrennt werden.

- Der eigentliche Algorithmus soll in wiederverwendbarer Form implementiert werden: gekapselt in eine Funktion (mit Unterfunktionen), die alle Parameter und Daten übergeben bekommt und selbst keinerlei Ein-/Ausgabe-Operationen durchführt. Nur so ist garantiert, daß sie auch unter anderen Bedingungen verwendbar ist.

- Fehlerhafte Daten o.ä. sollen weder zu einem Abbruch des Algorithmus noch zu einer Fehlermeldung führen: sie werden in geeigneter Form an das rufende Programm zurückgemeldet. Üblicherweise dient dazu eine ganzzahlige Variable `nerror` o.ä., deren Wert fehlerfreien Ablauf (`nerror=0`) oder einen bestimmten Fehlercode signalisiert.

- Nur in Ausnahmefällen sollen die Ein-/Ausgaberoutinen interaktiv arbeiten. Besser ist es, wenn die benötigten Daten aus einer Datei eingelesen und die Ergebnisse in eine Datei geschrieben werden.

- Es soll standardkonform und betriebssytemunabhängig programmiert werden. Die Notwendigkeit, auf einen anderen Rechner zu wechseln, kann sich unerwartet ergeben.

- Der Quelltext soll auf mehrere Dateien aufgeteilt werden - dies ermöglicht schnelleres Rekompilieren nach Änderungen.

- In C sollen Funktions-Prototypen und globale Deklarationen in Header-Dateien zusammengefaßt werden.

17 C99

Der GNU-Compiler `gcc` unterstützt in der aktuellen Version 3.2 bereits die meisten C99-Features. Dazu muß er mit der Option -std=gnu99 oder -std=c99 gerufen werden. Hier seien die wichtigsten Erweiterungen erwähnt:

17.1 C++-Elemente, die in C99 aufgenommen wurden

- `//`-Kommentare

- Definitionen/Deklarationen können in einem Block beliebig mit ausführbaren Anweisungen gemischt werden, sie müssen nicht am Blockanfang stehen. (Natürlich muß trotzdem eine Variable vor ihrer Anwendung deklariert sein.)

- Es gibt einen booleschen Typ, `bool` mit den Werten `true` und `false`. Im Unterschied zu C++ muß zu seiner Nutzung `<stdbool.h>` `include`d werden.

- Funktionen können das Attribut `inline` erhalten:

```
inline double hypot(double a, double b){
        return sqrt(a*a+b*b);
}
```

  Der Compiler ist aufgefordert, den Funktionsaufruf wegzuoptimieren, d.h. `x=y*hypot(f1,f2)` durch `x=y*sqrt(f1*f1+f2*f2)` zu ersetzen. `inline`-Funktionen sind nur innerhalb einer Quelltextdatei sichtbar, d.h. wenn man sie universell verwenden will, sind sie gute Kandidaten für die Aufnahme in eine Header-Datei.

- Deklarationen in `for`-Schleifeninitialisierung:
  `for(int k=0; k<k_end; k++){...}`
  Die Variable `k` wird erst am Anfang der `for`-Schleife eingeführt und ist sichtbar bis zum Ende des `for`-Blocks `{...}`.

17.2 Weitere C99-Erweiterungen

- Support für komplexe Zahlen:

```
#include <complex.h>

complex double a,b;
double xabs;

a = 10 - 17*I;
b = csqrt(a + 2*conj(a) + csin(a));
xabs = cabs( 3 + 10.2*I) + creal(a) + cimag(a);
```

- Einen Datentyp `long long int` für 64-Bit Integers. Typischerweise wird auf 32Bit-CPUs (Pentium-Familie) `int=long=4 Bytes, long long=8 Bytes` und auf 64Bit-CPUs (Itanium, UltraSparc) `int=4 Bytes, long=long long=8 Bytes` gelten.

- Eine Erweiterung der Initialisierungs-Syntax für Felder, Strukturen und Unions, "designated initializers" genannt. Für Felder sieht das so aus:

```
  int a[6] = { [4] = 29, [2] = 15 };
/* Dies ist äquivalent zu
  int a[6] = { 0, 0, 15, 0, 29, 0 }; */
```

- Deklarationen, die dem Compiler eine bessere Optimierung ermöglichen: `restrict`-Zeiger.
  Die Verwendung von Zeigern, insbesondere auch im Umgang mit Feldern/Matrizen führt in C zum "Aliasing-Problem": Der Compiler kann in der Regel nicht feststellen, ob zwei verschiedene Zeiger tatsächlich verschiedene Daten referenzieren, oder ob die Zeiger auf dieselben (oder überlappende) Daten zeigen. Damit sind viele Optimierungsstrategien, wie das Halten von Zwischenergebnissen in Registern oder das Umordnen von load/store-Operationen nicht möglich. Wenn nun ein Zeiger das `restrict`-Attribut trägt, wird dem Compiler signalisiert, daß auf den entsprechenden Datenbereich nur und ausschließlich mit diesem Zeiger zugegriffen wird:

```
/* Vektoraddition a1 = a1 + a2 */
void f1(int n, double * restrict a1, double * restrict a2){
     for(int i=0; i<n; i++) a1[i]+=a2[i];
}
```

  Hier sind Code-Umordnungen, Parallelisierungen u.ä. möglich, die offensichtlich nicht möglich wären (es gäbe falsche Ergebnisse), wenn `a1` und `a2` auf den gleichen oder einen überlappenden Speicherbereich weisen.

- Felder variabler Länge
  Endlich geht, was in Fortran schon immer geht: Man kann einer Funktion ein Feld beliebiger Größe zusammen mit der Größe übergeben.

```
void matmult(int n, int m, double a[n][m], double b[m][n], double c[n][n]){...}
```

Wichtig: die Argumente `n` und `m` müssen vor den Felder stehen, damit sie bereits deklariert sind. Der Protoyp für diese Funktion sieht so aus:
`void matmult(int, int, double [*][*],double [*][*],double [*][*]);`
Wie in Fortran90 können innerhalb einer Funktion ebenfalls Felder variabler Länge deklariert werden: `void func(int n){ double a[n]; ...}`

18 Fortran90/95

- Genaueres kommt in der nächsten "Auflage", wenn `g95` zur Verfügung steht.
- Das Standardwerk ist *"Fortran 90/95 explained"* von M. Metcalf und J. Reid.
- Eine knappe, gute Einführung sind Kapitel 21 und 22 von *"Numerical Recipes in Fortran 90"*, die man auf `www.nr.com` findet.

Anhang I: Zahlensysteme

Zahlen werden im Computer durch Bitmuster einer festen Länge N dargestellt.

Ganze Zahlen

Positive ganze Zahlen werden in Binärdarstellung dargestellt:

$$\begin{aligned}
0000\cdots0000 &= 0 \\
0000\cdots0001 &= 1 \\
0000\cdots0010 &= 2 \\
&\cdots \\
0111\cdots1111 &= 2^{N-1}-1
\end{aligned}$$

Für negative ganze Zahlen verwenden zahlreiche Prozessoren, darunter die Intel-Familie, die Zweierkomplement-Darstellung: $-x$ wird dargestellt, indem das Bitmuster von x invertiert und anschließend 1 addiert wird:

$$\begin{aligned}
1111\cdots1111 &= -1 \\
1111\cdots1110 &= -2 \\
&\cdots \\
1000\cdots0001 &= -2^{N-1}-1 \\
1000\cdots0000 &= -2^{N-1}
\end{aligned}$$

Damit ist das höchstwertige Bit Vorzeichenbit in dem Sinne, daß seine Anwesenheit negative Zahlen charakterisiert. Sein Setzen oder Löschen entspricht allerdings nicht der Transformation $x \to -x$.

Fließkommazahlen

erlauben, einen viel größeren Zahlenbereich darzustellen, allerdings mit einer begrenzten Genauigkeit ("Anzahl der gültigen Ziffern"). Die Zahl wird dargestellt durch einen Exponenten E, eine Mantisse M ($0 \leq M < 1$) und ein Vorzeichenbit V:

$$x = (-1)^V \times 2^E \times M$$

Die genauen Regeln für die Darstellung, Rundungsregeln, Fehlerbehandlung etc. in der Fließkommaarithmetik sind im amerikanischen Standard IEEE 754 aus dem Jahr 1985 festgelegt, der als IEC 559 im Jahr 1989 auch internationaler Standard wurde.

Die Intel-32Bit-Architektur kennt drei Formate:

Einfach langes Format: 32 Bit

1	8 Bit	23 Bit
V	E	M

Doppelt langes Format: 64 Bit

1	11 Bit	52 Bit
V	E	M

Erweitertes Format: 80 Bit

1	15 Bit	64 Bit
V	E	M

Zahlen im erweiterten Format werden von GNU C als Typ `long double` unterstützt. Für sie werden 96 Bit Speicherplatz (drei 32Bit-Wörter) reserviert, von denen 16 ungenutzt bleiben.

Intern arbeitet die Intel-CPU anstelle des doppelt langen immer mit dem erweiterten Format. Dies kann z.B. zu leicht unterschiedlichen Verhalten zwischen mit und ohne Optimierung kompilierten Programmversionen führen: Ohne Optimierung wird ein Zwischenresultat eventuell im RAM zwischengespeichert und dabei auf `double`-Format gekürzt, während es in der optimierten Version in einem (80 Bit langen) Prozessorregister gehalten wird und damit genauer ist.

	einfach	doppelt	erweitert
Exponentenlänge E (Bits)	8	11	15
kleinster Exponent e_{min}	-125	-1021	-16382
größter Exponent e_{max}	128	1024	16384
Mantissenlänge	24	53	64
betragsmäßig kleinste normalisierte Zahl größer Null	1.18×10^{-38}	2.23×10^{-308}	3.36×10^{-4932}
größte darstellbare Zahl	3.40×10^{38}	1.79×10^{308}	1.19×10^{4932}
kleinste Zahl ϵ, für die $1 + \epsilon \neq 1$	1.19×10^{-7}	2.22×10^{-16}	1.08×10^{-19}

- Die Größe ϵ begründet die Sprechweise "Einfach genaue Zahlen haben eine Genauigkeit von etwa 7 Dezimalstellen, doppelt genaue Zahlen etwa 16 gültige Dezimalstellen."
- Der Exponent wird nicht im Zweierkomplement o.ä. kodiert, sondern als vorzeichenlose Zahl, von der eine feste Zahl (bias) abgezogen wird.
- Die Exponenten überdecken nicht den vollen Wertebereich von E Bits. Das Bitmuster "111. . . 11" (e_{max}+1) im Exponentenfeld ist reserviert, um die speziellen Werte `NaN` ("Not a Number", Mantisse ungleich 0) und ±`Unendlich` (Mantisse=0, V=0 oder 1) zu kodieren. Bei einfach- und doppeltgenauen Zahlen ist das Bitmuster "000. . . 00" ($e_{min} - 1$) reserviert für denormalisierte Zahlen.
- Normalisierte und denormalisierte Zahlen: Die Aufteilung in Exponent und Mantisse ist nicht eindeutig. So kann (der Einfachheit halber im Dezimalsystem, die Mantisse sei dreistellig) 0.1 geschrieben werden als $.100 \times 10^0$, $.010 \times 10^1$ und $.001 \times 10^2$. Die erste Form, bei der die erste Stelle der Mantisse ungleich Null ist, ist die normalisierte Form. Nun ist die Mantisse binär kodiert, also ist die erste Stelle bei normalisierten Zahlen immer 1. Daher kann man auf die Speicherung der ersten Stelle der Mantisse verzichten (implizites erstes Bit) und dies wird bei einfach- und doppeltgenauen Zahlen auch so gemacht. Dies erklärt die Differenz zwischen den Angaben für die Mantissenlänge in der Tabelle (24 bzw. 53) und der vorhergehenden Angabe über die Zahl der Bits, die zum Speichern der Mantisse zur Verfügung stehen (23 bzw. 52). Im erweiterten Format wird das erste Bit mit kodiert.
- Denormalisierte Zahlen werden nur dann verwendet, wenn sie betragsmäßig so klein sind, daß eine normalisierte Darstellung nicht mehr möglich ist. Die kleinste normalisierte einfach genaue Zahl

hat die Mantisse .1000...0 binär ($= 2^{-1} = 0.5$) und den Exponenten $e_{min} = -125$, ist also gleich $2^{-125} \times 2^{-1} = 2^{-126} \approx 1.18 \times 10^{-38}$. Wird diese Zahl nun z.B. durch 8 geteilt, so ist das Ergebnis nur noch denormalisiert mit der Mantisse .00010...0 binär darstellbar. Aufgrund des impliziten ersten Bits muß man denormalisierte Zahlen durch einen speziellen Wert im Exponentenfeld kennzeichnen. (Interpretiert werden sie natürlich mit dem Exponenten $e = e_{min}$.)

- Das Erreichen denormalisierter Zahlen ist natürlich mit einem Genauigkeitsverlust begleitet. Trotzdem hat es sich in der Praxis als vorteilhaft herausgestellt, daß dadurch die ”Lücke um die Null” kleiner und ein ”gradual underflow” möglich ist. Es gibt auch Computersysteme (z.B. Cray, IBM/390), die keine denormalisierten Zahlen zulassen.

Anhang II: Zeichensätze

ASCII

Der American Standard Code for Information Interchange (ASCII) ist ein Zeichensatz, der nur 7 von den 8 Bits eines Bytes benutzt und demzufolge 128 Zeichen umfaßt.

Die ersten 32 Zeichen 0x00...0x1F sind nicht druckbare Zeichen und enthalten z.B. das Nullbyte 0x00 (in C als `\0` darstellbar und als Endmarkierung von Zeichenketten verwendet), den Zeilentrenner 0x0A (`\n`) und das Tabulatorzeichen 0x09 (`\t`).

0x20 kodiert das Leerzeichen. Auf Position 0x30...0x39 stehen die Ziffern 0...9, auf Position 0x41 bis 0x5A die Großbuchstaben A...Z und auf Position 0x61...0x7A die Kleinbuchstaben.

Weitere druckbare Zeichen (neben 0-9, A-Z, a-z und Leerzeichen) sind

```
! " # $ % & ' ( ) * + , - . / : ;  < = > ? @ [ ] \  ^ _ ` { } |  ~
```

ISO 8859

standardisiert eine Reihe weiterer 8Bit-Zeichensätze durch Auffüllung der freien Hälfte des ASCII-Zeichensatzes.

ISO 8859-1 (Latin1) enthält die Zusatzzeichen der meisten westeuropäischen Sprachen, wie ä (0xE4), é (0xE9), ñ (0xF1) oder æ (0xE6).

ISO 8859-15 (Latin9) ist eine kleine Modifikation von Latin1, bei der u.a. das Eurosymbol € auf Position 0xA4 eingeführt wurde.

Unicode

wird als einheitliche Kodierung aller Schriften der Welt immer wichtiger. Es können bis zu ca. 2 Millionen Zeichen kodiert werden, davon wird jedoch erst ein kleiner Teil, ca. 100 000 Zeichen genutzt.[4]

Unicode umfaßt bereits alle wichtigen Schriftsprachen der Gegenwart (z.B. über 40 000 chinesisch/japanisch/koreanische Schriftzeichen, Arabisch, Hebräisch, Tamil,...) und zahlreiche tote Sprachen (z.B. die um 1300 v.Chr. auf Kreta gebräuchliche ”Linearschrift B”) und wird zur Zeit noch ständig weiterentwickelt.

Unicode selbst ist keine Kodierung vergleichbar mit ISO 8859, sondern es gibt verschiedene ”Unicode transformation formats” (UTF), die einem Unicode-Zeichen eine eindeutige Sequenz von Bytes zuordnen. Die wichtigsten sind UTF-8, UTF-16 und UTF-32. UTF-8 enthält den ASCII-Zeichensatz als Untermenge. Zeichen, die nicht zu ASCII gehören, werden durch 2 bis 4 Bytes kodiert, wobei im ersten Byte a) das höchste Bit gesetzt ist (dadurch als nicht-ASCII erkennbar) und b) kodiert ist, wieviele der folgenden Bytes zu diesem Zeichen gehören.[5]

---

[4] `http://www.unicode.org`
[5] `http://www.cl.cam.ac.uk/~mgk25/unicode.html`