Programmieren in C

Burkhard Bunk

6.3.2013

1 C

Die Programmiersprache C wurde Anfang der 70er Jahre von Brian Kernighan und Dennis Ritchie bei Bell Labs (später AT&T) entwickelt und 1989 (in etwas weiter entwickelter Form) standardisiert (ANSI C). Es ist *die* Sprache für Probleme auf der Ebene des Betriebssystems, insbesondere Unix, und der darauf aufbauenden Anwendungssoftware. Inzwischen werden aber auch viele Programme für numerische Probleme in C geschrieben.

2 Beispielprogramm

Das folgende Programm steht in einer Datei `wurzeln.c`, die von

`http://poolinfo.physik.hu-berlin.de`

heruntergeladen werden kann. Es berechnet Quadrat– und Kubikwurzeln mit Hilfe der Iterationen

$$x_{n+1} = \frac{1}{2}\left(x_n + \frac{z}{x_n}\right) \to \sqrt{z}$$
$$y_{n+1} = \frac{1}{3}\left(2y_n + \frac{z}{y_n^2}\right) \to z^{1/3}$$

2.1 Listing

```
 1 /*                                                    B Bunk 4/2001
 2   Tabelle von Quadrat- und Kubikwurzeln berechnen     rev    3/2012
 3       und in File schreiben
 4 */
 5 #include <stdlib.h>                /* fuer exit() */
 6 #include <stdio.h>                 /* fuer printf, scanf, fopen etc */
 7 #include <math.h>                  /* nicht noetig */
 8
 9 #define NMAX 100
10 static float tol = 1.e-6;
```

```
11
12 float w2(float z);                  /* Prototypen der weiter unten */
13 void sub3(float z, float *w3);      /* definierten Unterprogramme */
14
15 int main(void){
16
17   float tabelle[NMAX][3];       /* Index-Bereiche 0:(NMAX-1), 0:2 */
18   float x0, dx, x, w3;
19   int   n, i;
20   FILE  *tabfile;
21
22   printf("x0 dx n? ");               /* Parameter abfragen */
23   scanf("%f%f%i", &x0, &dx, &n);     /* und einlesen */
24
25   if (n > NMAX){
26     printf("Fehler: Anzahl zu gross fuer Tabelle\n");
27     exit(1);
28   }
29
30   for (i = 0; i < n; i++){       /* Schleife ueber Tabellenzeilen */
31     x = x0 + i*dx;
32
33     tabelle[i][0] = x;           /* Tabellenzeile berechnen */
34     tabelle[i][1] = w2(x);       /* Funktionsaufruf */
35
36     sub3(x, &w3);                /* Aufruf von Unterprogramm */
37                                  /* Adresse von w3 uebergeben */
38                                  /* fuer Rueckgabe des Ergebnisses */
39     tabelle[i][2] = w3;
40
41     /* Kontrollausdruck */
42     printf("%10f %10f %10f\n",
43            tabelle[i][0], tabelle[i][1], tabelle[i][2]);
44   }
45
46   tabfile = fopen("wurzeln.tab","w"); /* File zum Schreiben oeffnen */
47   for (i = 0; i < n; i++){
48     /* Zeile schreiben */
49     fprintf(tabfile,"%10f %10f %10f\n",
50             tabelle[i][0], tabelle[i][1], tabelle[i][2]);
51   }
52   fclose(tabfile);                   /* File schliessen */
53   exit(0);
54 }
55
56 /* !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!! */
57
```

```
58 float w2(float z){
59
60   /* w2 <- sqrt(z) berechnen */
61
62   float x, xalt;
63
64   if (z < 0.){                       /* Argument pruefen */
65     printf("Fehler in w2: negatives Argument\n");
66     exit(2);
67   }
68
69   xalt = 0.;
70   x = 1.;                            /* Startwert */
71   while (fabs(x-xalt) > tol){        /* Konvergenztest */
72     xalt = x;
73     x = .5 * (x + z/x);              /* Iteration */
74   }
75   return x;
76 }
77
78 /* !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!! */
79
80 void sub3(float z, float *w3){
81
82   /* w3 <- z**(1/3) berechnen */
83
84   float w, walt;
85
86   if (z < 0.){                       /* Argument pruefen */
87     printf("Fehler in sub3: negatives Argument\n");
88     exit(3);
89   }
90
91   w = 1.;                            /* Startwert */
92   do    {
93     walt = w;
94     w = (2*w + z/(w*w))/3;           /* Iteration */
95   }
96   while (fabs(w-walt) > tol);        /* Konvergenztest */
97   *w3 = w;
98   return;
99 }
```

3 Zeilenkommentar

3.1 Struktur

5-13: Vorspann
15-54: Hauptprogramm `main`
58-76: Funktion `w2` für Quadratwurzel
80-99: Prozedur `sub3` für Kubikwurzel

3.2 Vorspann

1-4: `/* ... */` ist ein Kommentarblock (über mehrere Zeilen)
5-7: Präprozessor bindet Systemdateien (Header) ein
9: Präprozessor–Definition einer Konstanten (max. Tabellengröße)
10: globale Variable, gültig in der ganzen Quelldatei
12-13: Compiler wird informiert über Unterprogramme, die erst weiter unten definiert werden (Prototypen)

3.3 Hauptprogramm

15: Hauptprogramm heißt (immer) `main`, gibt eine `int` ans Betriebssystem zurück (return value) – Argumentliste ist leer (`void`), d.h. keine Übernahme aus der Kommandozeile – "{" leitet den Definitionsblock des Hauptprogramms ein (endet in Zeile 54)
17: Deklaration einer Matrix (für die Tabellierung der Werte) mit `NMAX` Zeilen und drei Spalten – Indizes $0..(NMAX-1)$ bzw. $0..2$
18-19: Deklaration der Variablen vom Typ Gleitkommazahl (`float`) und Ganzzahl (`int`)
20: Deklaration eines Zeigers auf eine Datei (file pointer)
22: Bildschirmausgabe der Frage nach Parametern (ohne Zeilenvorschub)
23: Einlesen der Parameter von der Tastatureingabe. Es sollen n Werte tabelliert werden, angefangen mit x_0 und in Schritten von dx. Die Formatangaben `%f` (für eine Variable vom typ `float`) und `%i` (für `int`) werden der Reihe nach den Variablen zugeordnet, die folgen. Hier müssen die **Adressen** der Variablen stehen, z.B. `&x0`, nicht der **Wert** `x0`.
25-28: Programm wird (mit Fehlermeldung) gestoppt, wenn Tabelle zu klein (Rückgabewert = 1)
30-44: Schleife über x-Werte, d.h. Tabellenzeilen
31-33: aktuellen x-Wert berechnen und in die erste Tabellenspalte schreiben
34: Wurzelfunktion (s.u.) aufrufen, Resultat in die zweite Spalte
36-39: Prozedur für Kubikwurzel aufrufen, Resultat `w3` in die dritte Spalte
42-43: aktuelle Tabellenzeile (Spalten `0..2`) auf den Schirm schreiben
46: Datei zum Schreiben öffnen, Zeiger `tabfile` definieren
47-51: Tabelle zeilenweise in Datei schreiben
52: Datei schließen

53: normales Ende des Programms (Rückgabewert = 0)
54: Ende des Hauptprogrammblocks

3.4 Funktion `w2`

58: Kopfzeile der Funktion `w2` mit Dummy–Argument `z`; der Funktionsname dient als Variable, um das Resultat zurückzugeben; Typen werden gleich hier deklariert
62: Deklaration lokaler Hilfsvariablen
64-67: Fehlerbehandlung (ggf. Rückgabewert = 2)
69-70: Iteration initialisieren
71-74: Iterationsschleife
71: `while` – Schleife läuft, solange Genauigkeit noch nicht erreicht ist
75: `x` als Wert der Funktion zurückgeben, Unterprogramm beenden
76: Ende des Unterprogrammblocks

3.5 Prozedur `sub3`

80: Kopfzeile der Prozedur mit Dummy–Argumenten `z` zur Eingabe und `w3` zur Rückgabe des Resultats (**Zeiger**); der Name der Prozedur dient nur zum Aufruf (Typ `void`)
84: Deklaration lokaler Hilfsvariablen
86-89: Fehlerbehandlung (ggf. Rückgabewert = 3)
91: Iteration initialisieren
92-96: Iterationsschleife
96: `do-while` – Schleife: läuft mindestens einmal und dann weiter, solange Genauigkeit noch nicht erreicht ist
97: `w` als **Wert** von `w3` zurückgeben
98: Unterprogramm beenden
99: Ende des Unterprogrammblocks

4 Übersetzen und Aufrufen

Das Quellprogramm muss mit einem C–Compiler übersetzt werden. In erster Linie bietet sich dafür der GNU–C–Compiler `gcc` an, aber man kann auch die Compiler von Intel oder Portland benutzen:

```
      unix> gcc wurzeln.c -lm
oder
      unix> icc wurzeln.c
oder
      unix> pgcc wurzeln.c
```

Der Schalter `-lm` bindet die (zu `math.h` gehörige) Mathematik–Bibliothek `libm` ein. Er ist nur bei `gcc` nötig, die beiden anderen Compiler machen das automatisch.

In allen Fällen entsteht eine ausführbare Binärdatei mit dem Standardnamen `a.out`, deren Aufruf das Programm ablaufen lässt:

```
unix> a.out
```

Meist gibt man der Binärdatei gleich beim Übersetzen einen besseren Namen, z.B. einfach den Basisnamen `wurzeln` ohne Erweiterung:

```
unix> gcc -o wurzeln wurzeln.c -lm
unix> wurzeln
```

5 Einige Punkte im Überblick

5.1 Quellformat

Präprozessor–Anweisungen sind an einem "#" in der ersten Spalte zu erkennen.
Im eigentlichen C–Programm ist die Formatierung frei, Anweisungen müssen mit ';' (Semikolon) abgeschlossen werden. Sie können sich über den Zeilenumbruch fortsetzten, auch mehrere Anweisungen pro Zeile sind möglich.
Namen für Objekte und Schlüsselwörter bestehen aus Buchstaben (groß/klein, signifikant), Ziffern (nicht am Anfang) und '_' (Unterstrich).
Zeichenketten werden mit `"..."` begrenzt.
Benutzung der Klammersymbole:

```
( )   in Ausdruecken und fuer Funktionsargumente,
[ ]   fuer Indizes von Feldern,
{ }   fuer Anweisungsbloecke.
```

Kommentare werden durch `/* ... */` geklammert und können sich über mehrere Zeilen erstrecken, dürfen aber nicht geschachtelt werden.

5.2 Datentypen

Die wichtigsten sind

```
int         ganze Zahl, 4 Bytes, Bereich ca +/- 2x10^9
long        ganze Zahl, 4 oder 8 Bytes

float       Kommazahl, 4 Bytes, 7 Stellen, aus +/-10^(+/-38)
double      Kommazahl, 8 Bytes, 16 Stellen, aus +/-10^(+/-308)

char        Zeichen(kette)
```

Die Bedeutung von `long` hängt von der Architektur des Systems ab (32/64 bit).
Deklarationen stehen entweder oberhalb der Programmteile (im Dateikopf), dann gelten sie bis ans Ende der Datei, oder am Anfang eines Programm– oder Anweisungsblocks (vor den ausführbaren Anweisungen), dann gelten sie bis zum Ende des Blocks.

Alle Namen muss man deklarieren, bevor man sie benutzt. Das gilt auch für aufzurufende Funktionen und Prozeduren, für die man einen *Prototyp* vorausschickt (meist im Dateikopf). Für die Systemfunktionen geschieht das über "Headerdateien", die der Präprozessor im System sucht und über Direktiven wie `#include <stdlib.h>` einfügt.

5.3 Zeiger und Werte

In C hat man nicht nur auf gespeicherte Werte Zugriff, sondern auch auf die Speicheradressen. Das sind eigentlich uninteressante "Hausnummern" im RAM, die auch noch bei jedem Programmstart anders ausfallen können. Trotzdem sind sie wichtige Objekte, z.B. für

- Zugriff auf Elemente von zusammengesetzten Datensätzen wie Listen (Vektoren), Matrizen, Zeichenketten etc;
- Rückgabe von Ergebnissen aus Unterprogrammen.

Ein *Name* kann dabei entweder für den Wert oder für die Adresse stehen, je nach Deklaration:

```
double x;               /* x bezeichnet einen Double-Wert */
double *y;              /* y bezeichnet die Adresse */

&x                      /* Zugriff auf die Adresse von x */
*y                      /* Zugriff auf den Double-Wert von y */
```

Eine Adressgröße wie `y` wird als Zeiger (*pointer*) bezeichnet.

5.4 Kontrollstrukturen

```
while ( Bedingung )           /* Test vor dem Durchlauf */
      Anweisungen

do
      Anweisungen
while ( Bedingung );          /* Test nach dem Durchlauf */

for ( Initialisierung; Bedingung; Inkrement/Dekrement )
      Anweisungen

if ( Bedingung )              /* einfachste Form */
      Anweisungen

if ( Bedingung )
      Anweisungen
else if ( Bedingung )         /* optional */
```

```
      Anweisungen
else if ( Bedingung )          /* optional */
      Anweisungen
...
else                           /* optional */
      Anweisungen
```

Nicht vergessen: jede Anweisung mit ';' abschließen, Blöcke aus mehreren Anweisungen in `{..}` setzen.
Vergleichsoperatoren:

```
==  !=  >  <  >=  <=
```

Hier noch eine Übersicht über Befehle, die etwas beenden:

Befehl..	beendet..
continue	Schleifendurchlauf
break	Schleife
return	Unterprogramm
exit()	Programm

In einer Funktion setzt man mit `return Ausdruck` auch gleich den Rückgabewert. Mit dem Argument von `exit()` kann man den Rückgabewert des Programms ans Betriebssystem übergeben.

5.5 Mathematische Funktionen

Hier eine Auswahl von Funktionen, die über `#include <math.h>` deklariert und (wenn nötig) mit der Compileroption `-lm` eingebunden werden:

```
pow  fabs  sqrt  exp  log  sin  ...
```

Diese Funktionen sind in `double` implementiert, aber auch mit `float` aufrufbar.
Es gibt kein Rechenzeichen für die Potenz, man muss dafür die Funktion `pow` verwenden!
Den Absolutwert für `int` und `long` bekommt man mit `abs` bzw. `labs` (aus `stdlib.h`).
Hilfe zu den eingebauten Funktionen kann man über die Unix–Manpages aufrufen, z.B.

```
unix> man sqrt
```

5.6 Formatangaben

Beim Lesen/Schreiben von formatierten Daten mit `scanf, printf, ..` muss man einen "Formatstring" angeben, in dem jeder Zahl (oder auch Zeichenkette) in der Argumentliste eine von % eingeleitete Formatspezifikationen zugeordnet ist. Hier die wichtigsten:

Spezifikation..	für Datentyp..	
%i (oder %d)	int	
%li	long	wichtig bei `scanf`
%f	float	
%e	float/double	Exponentialform
%g	float/double	%f oder %e je nach Größe
%lf (nicht %d !)	double	wichtig bei `scanf`
%s	char	Zeichenketten

Hinter dem % kann noch eine Angabe über Feldweite und Dezimalstellen eingefügt werden, z.B.

```
%10f        fuer float, Feldweite (mindestens) 10 Zeichen
%20.10f     fuer float, Feldweite 20, 10 Nachkommastellen
```

Wenn der Platz nicht reicht, wird das Feld verbreitert.