Modellierung und Programmierung

Dr. Martin Riplinger

31.10.2012

Eingabe durch den Benutzer: `scanf`

- dient u.a. zum Einlesen von Zahlen
- erfordert Präprozessordirektive: `#include<stdio.h>`
- Syntax:

```
scanf(Formatstring, Parameter);
```

- `Formatstring`: `%[modifizierer]typ`
- `Parameter`: `&` vor Variablennamen!

Beispiel:

```
#include <stdio.h>

main()
{
  setvbuf(stdout, NULL, _IONBF, 0);
  /*   Die Standard-Ausgabe wird nicht gepuffert */
  double zahl;
  printf("Bitte geben Sie eine Zahl ein: ");
  scanf("%lf", &zahl);
  printf("%lf zum Quadrat ist %lf\n", zahl, zahl*zahl);
}
```

Zeichen

- Zeichen (engl. *character*) werden intern wie (positive) ganzzahlige Werte behandelt, der Datentyp ist `char`. Variablen vom Typ `char` belegen im Speicher 1 Byte.
- Die standardisierte Zuordnung *Zeichen* $\longleftrightarrow$ *Zahl* erfolgt gemäß der ASCII-Tabelle. ASCII = American Standard Code for Information Interchange. Die Kodierung definiert 128 Zeichen, bestehend aus 33 nicht-druckbaren sowie 95 druckbaren:

0	Nullzeichen
1–32	Steuerzeichen
33–126	Ziffern, Buchstaben, Symbole, usw.

Auszug:

37	%	65	A	97	a	124	}
38	&	66	B	98	b	167	$
48	0	67	C	99	c	181	µ
49	1	92	\	123	{	223	ß

- Es sind äquivalent

```
char c = 'A';
char c = 65;
char c = 0101; // oktal:       65 = 1*64 + 0*8 + 1*1
char c = 0x41; // hexadezimal: 65 = 4*16 + 1*1
```

Zeichen

- Ausgabe mittels `printf`:

```
char c = 88;
printf("88 interpretiert als Zeichen: %c\n",c);
printf("88 interpretiert als integer: %d\n",c);
```

- Zeichenarithmetik

```
char c = 'A';
int diff = 'C' - c;   // diff = 2
c = 'B' + diff;       // c = 68
printf("Das zu c korrespondierende Zeichen ist %c\n",c);
```

- Einlesen von Zeichen via der Funktion `getchar`

```
int getchar(void)
```

- Beispiel

```
char c;

c = getchar();
```

Anweisungsblöcke

- Folge von Anweisungen, die von geschweiften Klammern eingeschlossen sind
- Anweisungsblock ist syntaktisch äquivalent zu einer einzelnen Anweisung
- eine einzelne Anweisung bedarf keiner Klammer
- Anweisungsblöcke können geschachtelt sein

Beispiele:

```
printf("Hallo Welt!\n");
```

```
{
  printf("Hallo Welt!\n");
  printf("Ich gehoere zum inneren Anweisungsblock.\n");
}
```

```
{
  printf("Ich bin im uebergeordneten Anweisungsblock!\n");
  {
    printf("Hallo Welt!\n");
    printf("Ich gehoere zum inneren Anweisungsblock.\n");
  }
}
```

if-Anweisung

```
if (Bedingung)
  Anweisungsblock
```

Wenn-Dann

oder

```
if (Bedingung)
  Anweisungsblock_1
else
  Anweisungsblock_2
```

Wenn-Dann-Andernfalls

Beispiel: Vorzeichenfunktion sign

Mathematische Definition:

$$sign(a) = \begin{cases} +1, & \text{falls } a > 0 \\ 0, & \text{falls } a = 0 \\ -1, & \text{falls } a < 0 \end{cases}$$

```
int a;
if (a>0)
  printf("sign(a) = +1\n");
else
{
  if (a==0)
    printf("sign(a) = 0\n");
  else
    printf("sign(a) = -1\n");
}
```

Mehrfache Alternativen: `else if`

Syntax

```
if (Bedingung_1)
  Anweisungsblock_1
else if (Bedingung_2)
  Anweisungsblock_2
      ⋮
else if (Bedingung_N)
  Anweisungsblock_N
else                  // optional
  Anweisungsblock     // optional
```

Beispiel

```
if (a > 0)
  printf("sign(a) = +1\n");
else if (a == 0)
  printf("sign(a) = 0\n");
else // oder else if (a < 0)
  printf("sign(a) = -1\n");
```

Viele Alternativen: `switch`

Syntax

```
switch (Variable)
{
  case Wert_1:
    Anweisungsblock_1
    break;            // optional
  case Wert_2:
    Anweisungsblock_2
      ⋮
  case Wert_N:
    Anweisungsblock_N
  default:            // optional
    Anweisungsblock   // optional
}
```

- Bei der Ausführung wird zu dem case label gesprungen, an der zum ersten Mal Variable und Wert übereinstimmt.
- Der **gesamte** folgende Code bis zum ersten `break` oder zum Ende des Blocks wird ausgeführt. Das schließt auch Code **außerhalb des angesprungenen case labels** ein.
- Als Wert im case label sind nur **Konstanten** zulässig.

Viele Alternativen: `switch`

Beispiel

```
unsigned char eingabe;

printf("Bitte Befehl eingeben: ");
scanf("%c", &eingabe);

switch (eingabe)
{
  case 'q':
    printf("Programm wird beendet!\n");
    break;
  case 'p':
  case 'P':
    printf("Drucken...\n");
    break;
  case 'h':
    printf("Hilfe wird aufgerufen.\n");
    break;
  default:
    printf("Eingabe nicht erkannt!\n");
    break;
}
```

Schleifen

- Schleifen wiederholen einen Anweisungsblock so lange bis ein bestimmtes Abbruchkriterium erfüllt ist
- Die wichtigsten Schleifen in C sind `for` und `while`

`while`-Schleife

- oft verwendet, wenn die Anzahl der Wiederholungen nicht vorherbestimmt ist
- Syntax:

```
while (Bedingung)
  Anweisungsblock
```

- wiederholt Anweisungsblock bis `Bedingung` "falsch", d.h. gleich `0` ist
- Autor ist selbst verantwortlich, dass das Abbruchkriterium irgendwann erfüllt ist → Gefahr einer Endlosschleife!
- Variablen in `Bedingung` müssen deklariert und ggf. initialisiert werden
- Anweisungsblock = "Rumpf" der Schleife

Beispiel:
Programm soll eine natürliche Zahl N einlesen und die Summe $1 + 2 + 3 + \ldots + N$ ausgeben.

Beispiel: `while`-Schleife

```c
#include<stdio.h>

main()
{
  int i = 1, sum = 0, N;

  printf("Geben Sie eine natuerliche Zahl ein: N= ");
  scanf("%d", &N);

  while (i <= N)
  {
    sum += i;
    i++;
  }

  printf("Die Summe der ersten %d natuerlichen Zahlen ", N);
  printf("betraegt %d\n", sum);
}
```

Beispiel: `while`-Schleife

Es sind äquivalent:

```c
int i = 1;

while (i <= N)
{
  sum += i;
  i++;
}
```

```c
int i = 0;

while (i < N)
{
  i++;
  sum += i;
}
```

```c
int i = 0;

while (i++ < N)
  sum += i;
```

```c
int i = 0;

while (++i <= N)
  sum += i;
```

$\rightsquigarrow$ Kein guter Stil, da fehleranfällig!

do-while-Schleife

- Anweisungsblock wird mindestens ein Mal ausgeführt
- beachte Semikolon am Ende

```
do
  Anweisungsblock
while (Bedingung);
```

- äquivalent zu:

```
Anweisungsblock
while (Bedingung)
  Anweisungsblock
```

Beispiel

```
int N;
do {
  printf("Bitte geben Sie eine ganze Zahl zwischen 5 und 15 ein:");
  scanf("%i", &N);
} while(N<5 || N>15);
```

for-Schleife

- vermutlich die am häufigsten verwendete Schleifenvariante
- kommt zum Einsatz wenn das Update immer gleich ist
- Anzahl der Wiederholungen ist a priori bekannt
- Syntax:

```
for (Initialisierung; Bedingung; Update)
  Anweisungsblock
```

- äquivalent zu:

```
Initialisierung;
while (Bedingung)
{
  Anweisungsblock
  Update;
}
```

- Beispiel:

```
int i;

for (i=1; i<=N; i++)
  sum += i;
```

Stolperfallen

- Abbruchkriterium fehlerhaft: Zuweisung statt Vergleich

```
for(i=1; i=N; i++)
  sum += i;
```

→ Endlosschleife!

- Leerer Anweisungsblock: Falsch platziertes Semikolon

```
for(i=1; i<=N; i++);
  sum += i;
```

- Unter- oder Überlauf

```
unsigned i;
for(i=N; i>=0; i--)
  sum += i;
```

```
unsigned char i;
for(i=1; i<=N; i++)
  sum += i;
```

→ Endlosschleife für N>255!

Schachtelung

Beispiel:

```
1  #include <stdio.h>
2
3  main()
4  {
5    int i, j;
6
7    for(i=1; i<=5; i++)       // aeussere Schleife
8    {
9      for(j=1; j<=5; j++)     // innere Schleife
10       printf("%2d ", i*j);  // Feldbreite 2 -> Zahlen rechtsbuendig
11
12     printf("\n");           // wieder aussen
13   }
14 }
```

Ausgabe:

```
 1   2   3   4   5
 2   4   6   8  10
 3   6   9  12  15
 4   8  12  16  20
 5  10  15  20  25
```

Steuerung von Wiederholungen

`break` beendet **aktuelle** Wiederholungsansweisung

`continue` Rest der Schleife wird übersprungen und der nächste Schleifendurchlauf gestartet

Merke: `break` und `continue` sollten sparsam eingesetzt werden, da sonst das Programm unübersichtlich wird.

`return` beendet aktuelle Funktion (später mehr!)

Absolut verpönt:

`goto` bewirkt einen Sprung im Programm an eine zurvor definierte Stelle

Merke: Anwendung von `goto` ist verboten und wird mit Fehlersuchen nicht unter 10 Jahren bestraft!

Steuerung von Wiederholungen

Beispiel:

```
1  #include <stdio.h>
2
3  main()
4  {
5    unsigned eingabe;
6
7    while(1)    // ohne break eine Endlosschleife!
8    {
9      printf("Bitte eine natuerliche Zahl kleiner als 100");
10     printf(" eingeben: ");
11     scanf("%u", &eingabe); // Vorsicht: Unterlauf moeglich!
12
13     if (eingabe < 100)
14     break;
15   }
16
17   printf("Die Zahl war %u.\n", eingabe);
18 }
```

Zufallszahlen

Definition

Ein **Zufallsexperiment** ist ein Experiment, dessen Ausgang nicht aus den vorherigen Ergebnissen vorausgesagt werden kann. Eine **Zufallszahl** ist eine Zahl, die sich aus dem Ergebnis eines Zufallsexperiments ableitet.

- „Echte“ Zufallszahlen sind prinzipiell nur solche, die von *wirklich zufälligen* (physikalischen) Prozessen abgeleitet werden (Beispiel: radioaktiver Zerfall).
- Viele physikalische Prozesse sind zwar deterministisch (vorhersagbar), jedoch nicht praktisch berechenbar (Beispiel: Temperatur am 9. Dezember des Folgejahres) $\rightarrow$ Pseudo-Zufall

Definition

Ein Experiment gilt als **pseudo-zufällig**, wenn sich sein Ergebnis zwar theoretisch voraussagen lässt, ohne Kenntnis der genauen Berechnungsvorschrift jedoch eine Prognose unmöglich ist.

Zufallszahlen

- Computer kann nur Pseudo-Zufallszahlen erzeugen
- Ziel (zunächst): Gleichverteilte Pseudo-Zufallszahlen auf $[0, 1[$ oder auf $[0, M[$ erzeugt
- Gute Generatoren müssen eine Reihe von statistischen Tests bestehen

Am weitesten verbreitet ist die **lineare Kongruenzmethode**:

1. Gib einen *seed*-Wert n_0 vor (Benutzereingabe / anderweitige Berechnung)
2. Berechne für feste natürliche Zahlen a, b, M und $k = 0, 1, \ldots$:

$$n_{k+1} = (a \cdot n_k + b) \mod M$$

   Soll das Ergebnis eine Gleitkommazahl in $[0, 1[$ sein, berechne

$$x_{k+1} = n_{k+1}/M.$$

Die Qualität des Generators hängt entscheidend von den Parametern a, b und M ab!

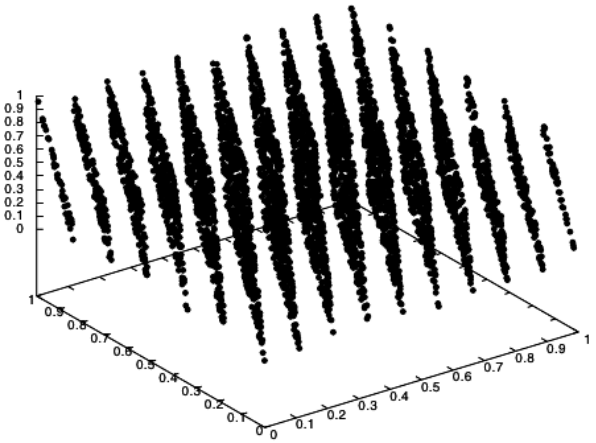
Zufallszahlen

Beispiel für einen „schlechten“ Generator (aus Kirsch/Schmitt, Kap. 12):

Wähle $a = 2^{16} + 3,\ b = 0,\ M = 2^{31}$

Visualisierung: Fasse je 3 aufeinanderfolgende Zahlen zu Vektoren in $\mathbb{R}^3$ zusammen.

Ergebnis: Jeder Vektor liegt in einer von 15 festen Ebenen!



Erzeugung von Zufallszahlen in C

- Präprozessordirektive `#include <stdlib.h>` notwendig
- Für `int`-Zufallszahlen auf $[0, \texttt{INT\_MAX}]$: Funktion `rand()`
- Seed-Wert für den `rand`-Generator wird mit `srand(seed)` festgelegt. Dabei wird `seed` als `unsigned int` interpretiert.
- Für `double`-Zufallszahlen auf $[0, 1]$: Funktion `drand48()`
- Seed-Wert für den `drand48`-Generator wird mit `srand48(seed)` festgelegt. Dabei wird `seed` als `long int` interpretiert.

Verwendung:

```
unsigned seed = 884722;
int zz;

srand(seed);
zz = rand();

printf("zz = %d\n", zz);
```

```
long seed = 88472250439203531568;
double zz;

srand(seed);
zz = drand48();

printf("zz = %f\n", zz);
```

Beispiel: (primitive) Simulation eines Aktienkurses

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

main()
{
  long seed;
  int i, periode = 20;        // Anzahl Tage
  double wert,                // aktueller Wert
         max_schwankung = 10; // Tagesschwankung maximal 10 Euro

  printf("Bitte Seed-Wert eingeben: ");
  scanf("%ld", &seed);
  srand48(seed);

  printf("Wert der Aktie zu Beginn (Euro): ");
  scanf("%lf", &wert);

  for(i=0; i<periode; i++)
    wert += 2 * (drand48() - 0.49) * max_schwankung;
         // Skalierung auf [-0.99,1.01]

  printf("Wert nach 20 Tagen: %.2f Euro\n", wert);
}
```