Einführung in die Programmierung für Physiker

Die Programmiersprache C – Kontrollstrukturen

Marc Wagner

Institut für theoretische Physik
Johann Wolfgang Goethe-Universität Frankfurt am Main

WS 2013/14

Anweisungen und Blöcke

Anweisungen

- Ausdrücke wie z.B. `x = 1.23` oder `printf("abc\n")` werden zu **Anweisungen** durch anhängen von `;`.

Blöcke

- Mehrere Anweisungen können mit `{` und `}` in einem **Block** zusammengefgasst werden.
- Ein Block ist syntaktisch äquivalent zu einer einzelnen Anweisung.
- Blöcke enden nicht mit `;`.
- Blöcke werden z.B. häufig bei `if`-Anweisungen oder `while`- und `for`-Schleifen verwendet.

```
1. #include<stdio.h>
2.
3. int main(void)
4. {
5.   int i = 2;
6.
7.   if(i == 2)
8.     {  // Anfang des if-Blocks ...
9.       printf("**************\n");
```

```
10.         printf("i ist gleich 2\n");
11.         printf("**************\n");
12.     }  // ... Ende des if-Blocks.
13. }
```

```
**************
i ist gleich 2
**************
```

Fallunterscheidungen: if ... else ...

- Syntax:
  - **if(***expr***)** *statement*.
  - **if(***expr***)** *statement1* **else** *statement2*.
  - *expr* bezeichnet einen logischen Ausdruck.
  - *statement*, *statement1* und *statement2* bezeichnen entweder einzelne Anweisungen oder Blöcke.
- Die **if**-Anweisung/Anweisungen *statement* bzw. *statement1* wird/werden ausgeführt, falls der logische Ausdruck *expr* "ungleich 0" (also "true") ist.
- Die **else**-Anweisung/Anweisungen *statement2* wird/werden ausgeführt, falls der logische Ausdruck *expr* "gleich 0" (also "false") ist.

```
1. #include<stdio.h>
2.
3. int main(void)
4. {
5.   double d = -3.0;
6.
7.   if(d < 0.0)
```

```
 8.     printf("d ist negativ\n");
 9.
10.   if(d < 0.0)
11.     {
12.       printf("d ist negativ\n");
13.     }
14.
15.   if(d >= 0.0)
16.     printf("d ist nicht negativ\n");
17.   else
18.     printf("d ist negativ\n");
19.
20.   if(d >= 0.0)
21.     {
22.       printf("d ist nicht negativ\n");
23.     }
24.   else
25.     {
26.       printf("d ist negativ\n");
27.     }
28. }
```

```
d ist negativ
d ist negativ
d ist negativ
d ist negativ
```

- Mehrdeutigkeit z.B. bei

if(*expr1*) if(*expr2*) ... else ....

- Auf welches **if** bezieht sich der **else**-Zweig?
  → Der **else**-Zweig ist mit dem letzten **if** verbunden.
- Derartig unübersichtliche Konstrukte sollten durch die Verwendung von Blöcken vermieden werden,
  if(*expr1*) { if(*expr2*) ... else ... }.

```
1.  #include<stdio.h>
2.
3.  int main(void)
4.  {
5.    double d = 0.0;
6.
7.    if(d >= 0.0)
8.      if(d != 0.0)
9.        printf("d ist positiv\n");
10.     else
11.       printf("d ist null\n");
12.   // --> Ausgabe "d ist null" ... o.k.
13.
14.   printf("**********\n");
15.
16.   if(d >= 0.0)
17.     if(d != 0.0)
18.       printf("d ist positiv\n");
```

```
19.   else
20.     printf("d ist negativ\n");
21.   // --> Ausgabe "d ist negativ" ... !!! nicht o.k. !!!
22.
23.   printf("**********\n");
24.
25.   if(d >= 0.0)
26.     {
27.       if(d != 0.0)
28.         printf("d ist positiv\n");
29.     }
30.   else
31.     printf("d ist negativ\n");
32.   // --> Keine Ausgabe ... o.k.
33. }
```

```
d ist null
**********
d ist negativ
**********
```

- Häufig verwendet man auch

  ```
  if(expr1) ...
  else if(expr2) ...
  else if(expr3) ...
  ...
  else ....
  ```

```
1. #include<stdio.h>
2. #include<stdlib.h>
3.
4. int main(void)
5. {
6.   enum wochentag { MO, DI, MI, DO, FR, SA, SO };
7.
8.   enum wochentag wt = MI;
9.
10.   if(wt == MO)
11.     printf("Montag\n");
12.   else if(wt == DI)
13.     printf("Dienstag\n");
14.   else if(wt == MI)
15.     printf("Mittwoch\n");
16.   else if(wt == DO)
17.     printf("Donnerstag\n");
18.   else if(wt == FR)
19.     printf("Freitag\n");
20.   else if(wt == SA)
21.     printf("Samstag\n");
22.   else if(wt == SO)
23.     printf("Sonntag\n");
24.   else
25.     {
26.       printf("Fehler: Unzulaessiger Wert von wt!\n");
```

```
27.         exit(0);
28.     }
29. }
```

Mittwoch

Fallunterscheidungen: switch ...

- Syntax:
  - ```
    switch(expr) {
    case const_expr1: statements
    case const_expr2: statements
    ...
    default: statements
    }.
    ```
  - *expr* bezeichnet einen Integer-Ausdruck.
  - *const_expr1*, *const_expr2*, ... bezeichnen paarweise verschiedene Integer-Konstanten.
  - *statements* bezeichnet eine Folge von Anweisungen (auch keine Anweisung ist erlaubt).
- Falls der Wert von *expr* gleich der Konstante *const_expr1* ist, wird mit den Anweisungen nach der **case-Marke**
  `case const_expr1:`
  fortgefahren; alle weiteren Anweisungen (auch solche nach weiteren **case**-Marken) werden ausgeführt.
    ```

- Falls der Wert von *expr* gleich der Konstante *const_expr2* ist, ... (analog).
- Falls der Wert von *expr* keiner **`case`**-Marke entspricht, wird mit den Anweisungen nach
  `default:`
  fortgefahren; **`default`** kann auch weggelassen werden.
- Die Reihenfolge von **`case`**-Marken und **`default`** ist beliebig.
- Der **`switch`**-Block kann mit der Anweisung
  `break;`
  vorzeitig verlassen werden.

```
1. #include<stdio.h>
2.
3. int main(void)
4. {
5.   enum wochentag { MO, DI, MI, DO, FR, SA, SO };
6.
7.   enum wochentag wt = MI;
8.
9.   printf("Uni-Termine heute:\n");
10.
11.   switch(wt)
12.     {
13.     case SA:
14.     case SO:
15.       printf("Keine Termine (Wochenende).\n");
```

```
16.       break;
17.
18.     case DO:
19.       printf("08:15-09:45 PPROG-Uebung bei Alessandro.\n");
20.       printf("14:15-15:45 PPROG-Vorlesung.\n");
21.
22.     default:
23.       printf("16:00       Kaffee im Cafe-Physik.\n");
24.       break;
25.
26.     case FR:
27.       printf("15:00       Kaffee im Cafe-Physik.\n");
28.     }
29. }
```

```
Uni-Termine heute:
16:00       Kaffee im Cafe-Physik.
```

```
6. ...
7.   enum wochentag wt = DO;
8. ...
```

```
Uni-Termine heute:
08:15-09:45 PPROG-Uebung bei Alessandro.
14:15-15:45 PPROG-Vorlesung.
16:00       Kaffee im Cafe-Physik.
```

```
6. ...
7.   enum wochentag wt = FR;
8. ...
```

```
Uni-Termine heute:
15:00       Kaffee im Cafe-Physik.
```

Schleifen: while ...

- Syntax:
  - **while(***expr***)** *statement*.
  - *expr* bezeichnet einen logischen Ausdruck.
  - *statement* bezeichnet entweder eine einzelne Anweisung oder einen Block.
- Die Anweisungen innerhalb der **while**-Schleife werden wiederholt, so lange der logische Ausdruck *expr* "ungleich 0" (also "true") ist.

```
1. #include<stdio.h>
2.
3. int main(void)
4. {
5.   int ctr = 0;
6.
7.   while(ctr < 5)
8.     printf("ctr = %d\n", ctr++);
9.
10.   while(ctr < 10)
11.     {
12.       printf("ctr = %d\n", ctr);
13.       ctr++;
```

```
14.     }
15. }
```

```
ctr = 0
ctr = 1
ctr = 2
ctr = 3
ctr = 4
ctr = 5
ctr = 6
ctr = 7
ctr = 8
ctr = 9
```

- Eine **`while`**-Schleife kann mit der Anweisung
  `break;`
  vorzeitig verlassen werden.

```
1. #include<stdio.h>
2. 
3. int main(void)
4. {
5.   int ctr = 0;
6. 
7.   while(1)  // while(1) --> unendliche Schleife.
8.     {
9.       // Abbruch der Schleife "von Hand", sobald ctr >= 10.
10.      if(ctr >= 10)
11.        break;
12. 
13.      printf("ctr = %d\n", ctr);
```

```
14.         ctr++;
15.     }
16. }
```

```
ctr = 0
ctr = 1
ctr = 2
ctr = 3
ctr = 4
ctr = 5
ctr = 6
ctr = 7
ctr = 8
ctr = 9
```

- Mit der Anweisung
  continue;
  können innerhalb eines **while**-Blocks die noch folgenden Anweisungen übersprungen werden; es wird direkt mit der Überprüfung des logischen Ausdrucks *expr* fortgefahren und bei "true" der **while**-Block erneut ausgeführt.

```
1. #include<stdio.h>
2. 
3. int main(void)
4. {
5.   int ctr = 0;
6. 
7.   while(ctr < 10)
8.     {
9.       printf("ctr = %d\n", ctr);
10.       ctr++;
```

```
11.
12.       // Ueberspringe die folgende print-Anweisung, falls ctr nicht durch 3 teilbar ist.
13.       if(ctr%3 != 0)
14.         continue;
15.
16.       printf("**********\n");
17.     }
18. }
```

```
ctr = 0
ctr = 1
ctr = 2
**********
ctr = 3
ctr = 4
ctr = 5
**********
ctr = 6
ctr = 7
ctr = 8
**********
ctr = 9
```

Schleifen: do ... while ...

- Syntax:
  - **do** *statement* **while(***expr***);**.
  - *expr* bezeichnet einen logischen Ausdruck.
  - *statement* bezeichnet entweder eine einzelne Anweisung oder einen Block.
- Die Anweisungen innerhalb der **do-while**-Schleife werden ausgeführt und dann wiederholt, so lange der logische Ausdruck *expr* "ungleich 0" (also "true") ist.

```
1. #include<stdio.h>
2.
3. int main(void)
4. {
5.   int ctr = 0;
6.
7.   do
8.     {
9.       printf("ctr = %d\n", ctr);
10.      ctr++;
11.    }
12.  while(ctr < 10);
13. }
```

```
ctr = 0
ctr = 1
ctr = 2
ctr = 3
ctr = 4
ctr = 5
ctr = 6
ctr = 7
ctr = 8
ctr = 9
```

- Eine **do-while**-Schleife kann mit der Anweisung

  break;

  vorzeitig verlassen werden.

- Mit der Anweisung

  continue;

  können innerhalb eines **do-while**-Blocks die noch folgenden Anweisungen übersprungen werden; es wird direkt mit der Überprüfung des logischen Ausdrucks *expr* fortgefahren und bei "true" der **do-while**-Block erneut ausgeführt.

Schleifen: for ...

- Syntax:
  - **for(***expr1***;** *expr2***;** *expr3***)** *statement*.
  - *expr1* bezeichnet einen Ausdruck, in der Regel die Initialisierung einer Zählvariable.
  - *expr2* bezeichnet einen logischen Ausdruck.
  - *expr3* bezeichnet einen Ausdruck, in der Regel eine nach jedem Schleifendurchlauf stattfindende Veränderung einer Zählvariable.
  - *statement* bezeichnet entweder eine einzelne Anweisung oder einen Block.
- Zu Beginn der **for**-Schleife wird einmalig *expr1* ausgewertet.
- Die Anweisungen innerhalb der **for**-Schleife werden wiederholt, so lange der logische Ausdruck *expr2* "ungleich 0" (also "true") ist.
- Nach jedem Schleifendurchlauf wird *expr3* ausgewertet.

```
1. #include<stdio.h>
2.
3. int main(void)
4. {
5.   int ctr;
```

```
6.
7.   for(ctr = 0; ctr < 10; ctr++)
8.     printf("ctr = %d\n", ctr);
9. }
```

```
ctr = 0
ctr = 1
ctr = 2
ctr = 3
ctr = 4
ctr = 5
ctr = 6
ctr = 7
ctr = 8
ctr = 9
```

- Eine **`for`**-Schleife kann mit der Anweisung
  `break;`
  vorzeitig verlassen werden.

```
1. #include<stdio.h>
2.
3. int main(void)
4. {
5.   int ctr;
6.
7.   for(ctr = 0; ctr < 10; ctr++)
8.     {
9.       // Abbruch der Schleife "von Hand", sobald ctr == 5.
10.      if(ctr == 5)
11.        break;
```

```
12.
13.       printf("ctr = %d\n", ctr);
14.     }
15. }
```

```
ctr = 0
ctr = 1
ctr = 2
ctr = 3
ctr = 4
```

- Mit der Anweisung
  `continue;`
  können innerhalb eines **`for`**-Blocks die noch folgenden Anweisungen übersprungen werden; es wird direkt mit der Auswertung von *expr3* und der Überprüfung des logischen Ausdrucks *expr2* fortgefahren und bei "true" der **`for`**-Block erneut ausgeführt.

```
1. #include<stdio.h>
2.
3. int main(void)
4. {
5.   int ctr;
6.
7.   for(ctr = 0; ctr < 10; ctr++)
8.     {
9.       // Ueberspringe die folgende print-Anweisung, falls ctr nicht gerade ist.
10.      if(ctr%2 != 0)
11.        continue;
12.
```

```
13.         printf("ctr = %d\n", ctr);
14.     }
15. }
```

```
ctr = 0
ctr = 2
ctr = 4
ctr = 6
ctr = 8
```

goto ...

- Syntax:
  - **goto** *label*;.
  - *label*:.
- **goto** *label*;
  bewirkt, dass zur **Marke**
  label:
  gesprungen und mit den nach der Marke folgenden Anweisungen fortgefahren wird.

```
1. #include<stdio.h>
2.
3. int main(void)
4. {
5.   int ctr = 0;
6.
7.  loop_start:
8.
9.   printf("ctr = %d\n", ctr);
10.   ctr++;
11.
12.   if(ctr < 10)
```

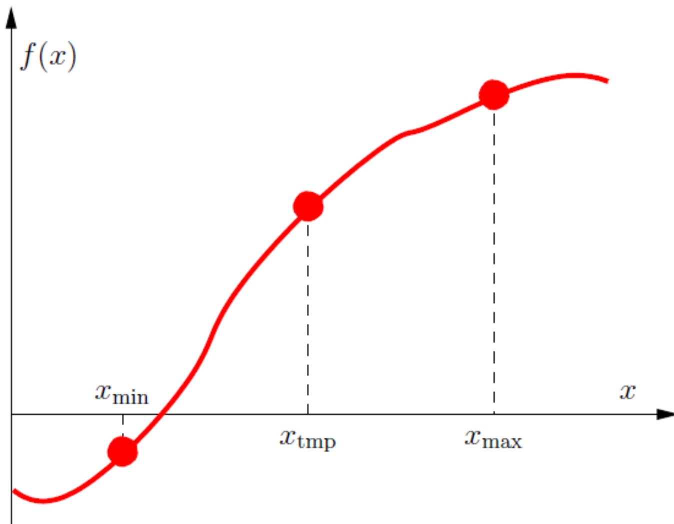
```
13.     goto loop_start;
14. }
```

```
ctr = 0
ctr = 1
ctr = 2
ctr = 3
ctr = 4
ctr = 5
ctr = 6
ctr = 7
ctr = 8
ctr = 9
```

- Die Verwendung von **goto** gilt als schlechter Programmierstil (Programmcode ist unübersichtlich und damit fehleranfällig) und sollte nur in Ausnahmefällen eingesetzt werden.

Anwendung: Nullstellensuche mit Bisektion

- Häufig lassen sich die Nullstellen einer Funktion $f(x)$ nicht analytisch bestimmen (Nebenbemerkung: Die Lösung einer beliebigen [z.B. nicht-linearen] Gleichung $g(x) = h(x)$ ist äquivalent zur Nullstellensuche $f(x) = 0$ mit $f(x) \equiv g(x) - h(x)$).
- Ein einfaches numerisches Verfahren zur Nullstellensuche ist die **Bisektion**:
  - Startpunkt: x_{min} und x_{max} mit $f(x_{\mathrm{min}})f(x_{\mathrm{max}}) \leq 0$ (damit ist mindestens eine Nullstelle im Intervall $[x_{\mathrm{min}}, x_{\mathrm{max}}]$ garantiert).
  - **Algorithmus**:
    1. Falls $x_{\mathrm{max}} - x_{\mathrm{min}} < \epsilon$, beende den Algorithmus; die Nullstelle ist $x_0 = (x_{\mathrm{min}} + x_{\mathrm{max}})/2$ mit einer numerischen Genauigkeit von mindestens $\epsilon/2$.
    2. Teile das Intervall in der Mitte, d.h. bei $x_{\mathrm{tmp}} = (x_{\mathrm{min}} + x_{\mathrm{max}})/2$.
    3. Falls eine Nullstelle im linken Intervall garantiert ist (falls $f(x_{\mathrm{min}})f(x_{\mathrm{tmp}}) \leq 0$), ersetze $x_{\mathrm{max}} = x_{\mathrm{tmp}}$; gehe zu 1.
    4. Eine Nullstelle ist im rechten Intervall garantiert; ersetze $x_{\mathrm{min}} = x_{\mathrm{tmp}}$; gehe zu 1.



- Das folgende Programm bestimmt die im Intervall $[3.0, 3.5]$ liegende Nullstelle der Funktion $\sin(x)$ mit der numerischen Genauigkeit von $\epsilon = 10^{-6}$ mit Hilfe von Bisektion (das Ergebnis ist natürlich bekannt, $x_0 = \pi$).

```
1. #include<math.h>
2. #include<stdio.h>
3. #include<stdlib.h>
4.
5. // ********************
6.
7. // Die Funktion, von der eine Nullstelle gesucht wird (sin(x)).
8. double f(double x)
9. {
```

```
10.   return sin(x);
11. }
12.
13. // ********************
14.
15. int main(void)
16. {
17.   double eps = pow(10.0, -6.0);  // Genauigkeit der numerisch zu bestimmenden Nullstelle.
18.
19.   double x_min = 3.0;  // Minimaler Wert des Startintervalls.
20.   double x_max = 3.5;  // Maximaler Wert des Startintervalls.
21.
22.   // **********
23.
24.   // Fehlerabfrage der Input-Daten.
25.
26.   if(eps <= 0.0)
27.     {
28.       printf("Fehler: eps <= 0.0.\n");
29.       exit(0);
30.     }
31.
32.   if(x_max < x_min)
33.     {
34.       printf("Fehler: x_max < x_min.\n");
35.       exit(0);
36.     }
```

```
37.
38.  double f_min = f(x_min);
39.  double f_max = f(x_max);
40.
41.  if((f_min <= 0.0 && f_max <= 0.0) ||
42.     (f_min >= 0.0 && f_max >= 0.0))
43.    {
44.      printf("Fehler: Nullstelle im Intervall [x_min,x_max] ist nicht garantiert.\n");
45.      exit(0);
46.    }
47.
48.  // **********
49.
50.  printf("x_min = %+f   f(x_min) = %+f   x_max = %+f   f(x_max) = %+f\n", x_min, f_min, x_max, f_max);
51.
52.  // Nullstellensuche mit Bisektion.
53.
54.  while(x_max-x_min > eps)
55.    {
56.      double x_tmp = 0.5 * (x_min+x_max);
57.      double f_tmp = f(x_tmp);
58.
59.      if((f_min <= 0.0 && f_tmp >= 0.0) ||
60.         (f_min >= 0.0 && f_tmp <= 0.0))
61.        // Nullstelle im linken Teilintervall [x_min,x_tmp].
62.        {
63.          x_max = x_tmp;
```

```
64.             f_max = f_tmp;
65.           }
66.         else
67.           // Nullstelle im rechten Teilintervall [x_tmp,x_max].
68.           {
69.             x_min = x_tmp;
70.             f_min = f_tmp;
71.           }
72.
73.         printf("x_min = %+f   f(x_min) = %+f   x_max = %+f   f(x_max) = %+f\n", x_min, f_min, x_max, f_max);
74.       }
75.
76.   double x_0 = 0.5 * (x_min+x_max);  // Die gesuchte Nullstelle.
77.   printf("Nullstelle gefunden: f(%+f) = %+e\n", x_0, f(x_0));
78. }
```

```
x_min = +3.000000   f(x_min) = +0.141120   x_max = +3.500000   f(x_max) = -0.350783
x_min = +3.000000   f(x_min) = +0.141120   x_max = +3.250000   f(x_max) = -0.108195
x_min = +3.125000   f(x_min) = +0.016592   x_max = +3.250000   f(x_max) = -0.108195
x_min = +3.125000   f(x_min) = +0.016592   x_max = +3.187500   f(x_max) = -0.045891
x_min = +3.125000   f(x_min) = +0.016592   x_max = +3.156250   f(x_max) = -0.014657
x_min = +3.140625   f(x_min) = +0.000968   x_max = +3.156250   f(x_max) = -0.014657
x_min = +3.140625   f(x_min) = +0.000968   x_max = +3.148438   f(x_max) = -0.006845
x_min = +3.140625   f(x_min) = +0.000968   x_max = +3.144531   f(x_max) = -0.002939
x_min = +3.140625   f(x_min) = +0.000968   x_max = +3.142578   f(x_max) = -0.000985
x_min = +3.140625   f(x_min) = +0.000968   x_max = +3.141602   f(x_max) = -0.000009
x_min = +3.141113   f(x_min) = +0.000479   x_max = +3.141602   f(x_max) = -0.000009
x_min = +3.141357   f(x_min) = +0.000235   x_max = +3.141602   f(x_max) = -0.000009
x_min = +3.141479   f(x_min) = +0.000113   x_max = +3.141602   f(x_max) = -0.000009
x_min = +3.141541   f(x_min) = +0.000052   x_max = +3.141602   f(x_max) = -0.000009
x_min = +3.141571   f(x_min) = +0.000022   x_max = +3.141602   f(x_max) = -0.000009
x_min = +3.141586   f(x_min) = +0.000006   x_max = +3.141602   f(x_max) = -0.000009
x_min = +3.141586   f(x_min) = +0.000006   x_max = +3.141594   f(x_max) = -0.000001
x_min = +3.141590   f(x_min) = +0.000003   x_max = +3.141594   f(x_max) = -0.000001
```

```
x_min = +3.141592   f(x_min) = +0.000001   x_max = +3.141594   f(x_max) = -0.000001
x_min = +3.141592   f(x_min) = +0.000001   x_max = +3.141593   f(x_max) = -0.000000
Nullstelle gefunden: f(+3.141593) = +1.509958e-07
```