Kapitel 7

Programmflusskontrolle

7.1 Bedingte Ausführung

Bei der Bedingten Ausführung werden Ausdrücke auf ihren Wahrheitswert hin überprüft und der weitere Ablauf des Programms davon abhängig gemacht. C sieht hierfür die Anweisungen `if` und `switch` vor.

7.1.1 Die if()–Anweisung

Die allgemeine Form der Verzweigung (Alternative) ist

```
if (logischer Ausdruck)
{
      Anweisungen A;
}
else
{
      Anweisungen B;
}
```

und zählt ihrerseits wiederum als Anweisung. Der `else`–Zweig kann weggelassen werden (einfache Alternative). Folgt nach dem `if`– bzw. `else`–Befehl nur eine Anweisung, so muss diese nicht in einen Block (geschweifte Klammern) geschrieben werden.

Beispiel 7.1 Signum-Funktion. Die Signum-Funktion gibt das Vorzeichen an:

$$y(x) = \begin{cases} 1 & x > 0 \\ 0 & x = 0 \\ -1 & x < 0 \end{cases}$$

```
int main() /* Signum Funktion */
{
       float x,y;

       if (x>0.0)
       {
              y=1.0;
       }
       else
       {
              if (x == 0.0)
```

```
                {
                        y=0.0;
                }
                else
                {
                        y=-1.0;
                }
        }
        return 0;
}
```

□

7.1.2 Die switch()–Anweisung

Zur Unterscheidung von mehreren Fällen ist die Verwendung von `switch`–`case`–Kombinationen bequemer. Mit dem Schlüsselwort `switch` wird ein zu überprüfender Ausdruck benannt. Es folgt ein Block mit `case`–Anweisungen, die für die einzelnen möglichen Fälle Anweisungsblöcke vorsehen. Mit dem Schlüsselwort `default` wird ein Anweisungsblock eingeleitet, der dann auszuführen ist, wenn keiner der anderen Fälle eingetreten ist (optional).

```
 switch (Ausdruck)
 {
       case Fall 1:
       {
              Anweisungen f\"ur Fall 1
              break;
       }
       ...
       case Fall n:
       {
              Anweisungen f\"ur Fall n
              break;
       }
       default:
       {
              Anweisungen f\"ur alle anderen F\"alle
              break;
       }
 }
```

Achtung!!! Man beachte, dass der Anweisungsblock jedes `case`–Falles mit `break` abgeschlossen werden muss! Ansonsten wird in C der nächste `case`–Block abgearbeitet. Das ist anders als in MATLAB!

Beispiel 7.2 switch–Anweisung.

```
#include <stdio.h>

int main()
{
       int nummer;
       printf("Geben Sie eine ganze Zahl an: ");
       scanf("%i",&nummer);

       printf("Namen der Zahlen aus {1,2,3} \n");
```

```
        switch (nummer)
        {
                case 1:
                {
                        printf("Eins = %i \n", nummer);
                        break;
                }
                case 2:
                {
                        printf("Zwei = %i \n", nummer);
                        break;
                }
                case 3:
                {
                        printf("Drei = %i \n", nummer);
                        break;
                }
                default:
                {
                        printf("Die Zahl liegt nicht in der Menge {1,2,3} \n");
                        break;
                }
        }
        return 0;
}
```

□

7.2 Schleifen

Schleifen dienen dazu, die Ausführung von Anweisungsblöcken zu wiederholen. Die Anzahl der Wiederholungen ist dabei an eine Bedingung geknüpft. Zur Untersuchung, ob eine Bedingung erfüllt ist, werden Vergleichs- und Logikoperatoren aus Kapitel 6.4 benutzt.

7.2.1 Der Zählzyklus (for–Schleife)

Beim Zählzyklus steht die Anzahl der Zyklendurchläufe a–priori fest, der Abbruchtest erfolgt vor dem Durchlauf eines Zyklus. Die allgemeine Form ist

```
        for (ausdruck1; ausdruck2; ausdruck3)
        {
                Anweisungen;
        }
```

Beispiel 7.3 Summe der natürlichen Zahlen von 1 bis n. Vergleich dazu auch Algorithmus 3.5.

```
#include <stdio.h>

int main()
{
        int i,summe,n;
        char tempstring[80];
        /* Einlesen der oberen Schranke n
         * von der Tastatur */
```

```
        printf("Obere Schranke der Summe : ");
        fgets(tempstring, sizeof(tempstring), stdin);
        sscanf(tempstring, "%i", &n);

        summe=0;        /* Setze summe auf 0                */
        for (i=1; i<=n; i=i+1)
        {
                summe=summe+i;
        }
        printf("Summe der Zahlen von 1 bis %i ist %i \n",n,summe);
        return 0;
}
```

Im obigen Programmbeispiel ist `i` die Laufvariable des Zählzyklus, welche mit `i=1` (ausdruck1) initialisiert wird, mit `i=i+1` (ausdruck3) weitergezählt und in `i <= n` (ausdruck2) bzgl. der oberen Grenze der Schleifendurchläufe getestet wird. Im Schleifeninneren `summe=summe+i;` (anweisung) erfolgen die eigentlichen Berechnungsschritte des Zyklus. Die Summationsvariable muss vor dem Eintritt in den Zyklus initialisiert werden. □

Beispiel 7.4 Kompakte Programmierung der Summe der natürlichen Zahlen von 1 bis n. Eine kompakte Version dieser Summationsschleife (korrekt, aber sehr schlecht lesbar) wäre:

```
for (summe=0, i=1; i <=n; summe+=i, i++);
```

Man unterscheidet dabei zwischen den Abschluss einer Anweisung ";" und dem Trennzeichen "," in einer Liste von Ausdrücken. Diese Listen werden von links nach rechts abgearbeitet. □

Der ausdruck2 ist stets ein logischer Ausdruck und ausdruck3 ist ein arithmetischer Ausdruck zur Manipulation der Laufvariablen. Die Laufvariable kann eine einfache Variable vom Typ int, float oder double sein.

Achtung!!! Vorsicht bei der Verwendung von Gleitkommazahlen (float,double) als Laufvariable. Dort ist der korrekte Abbruchtest wegen der internen Zahlendarstellung unter Umständen nicht einfach zu realisieren.

Die folgenden Beipiele 7.5, 7.6 verdeutlichen die Problematik der begrenzten Genauigkeit von Gleitkommazahlen in Verbindung mit Zyklen und einige Tipps zu deren Umgehung.

Beispiel 7.5 Ausgabe der diskreten Knoten x_i des Intervalls [a,b], welches in n gleichgroße Teilintervalle zerlegt wird, d.h.

$$x_i = a + ih,\ i = 0, \ldots, n \ \text{ mit } \ h = \frac{b-a}{n}$$

```
#include <stdio.h>

int main()
{
        float a,b,xi,h;
        int n;
        char tempstring[80];

        a=0.0;   /* Intervall [a,b] wird initialisiert */
        b=1.0;   /* mit [0,1] */
```

```
        printf("Geben Sie die Anzahl der Teilintervalle an: ");
        fgets(tempstring, sizeof(tempstring), stdin);
        sscanf(tempstring, "%i", &n);
        h=(b-a)/n;

        n=1;    /* n wird nun als Hilfsvariable verwendet */
        for (xi=a; xi<=b; xi=xi+h)
        {
                printf("%i.te Knoten : %f \n",n,xi);
                n=n+1;
        }
        return 0;
}
```

Da Gleitkommazahlen nur eine limitierte Anzahl gültiger Ziffern besitzt, wird es oft passieren, dass der letzte Knoten $x_n = b$ nicht ausgegeben wird. Auswege sind:

1.) Änderung des Abbruchtests in `xi <= b + h/2.0` (jedoch ist x_n immer noch fehlerbehaftet).

2.) Keine forlaufende Addition, sondern Berechnung der Knoten immer vom ersten Knoten ausgehend, verwende dazu Zyklus mit int-Variable:

```
    for (i=0; i<=n; i++)
    {
      xi=a+i*h;
      printf("'\%i.te Knoten : \%f \bs n"',n,xi);
    }
```

□

Die gemeinsame Summation kleinerer und größerer Zahlen kann ebenfalls zu Ungenauigkeiten führen, vergleiche auch Algorithmus 3.2 zur Lösung der quadratischen Gleichung.

Beispiel 7.6 Im diesem Beispiel wird die Summe $\sum_{i=1}^{n} 1/i^2$ auf zwei verschiedene Arten berechnet:

$$\sum_{i=1}^{n} \frac{1}{i^2} = \sum_{i=1}^{n} \frac{1}{(i*i)} = \sum_{i=1}^{n} \frac{\left(\frac{1}{i}\right)}{i}.$$

Der Reihenwert ist $\pi^2/6 = 1.644934068....$

```
#include <stdio.h>
#include <math.h>
#include <limits.h> /* enth\"alt die Konstante INT_MAX */

int main()
{
        float summe1 = 0.0,summe2 = 0.0;
        int i,n;
        char tempstring[80];

        printf("Der erste Algorithmus wird ungenau f\"ur n bei ca. %f \n",
                ceil(sqrt(1.0/1.0e-6)) );
        /* siehe Kommentar 1.Schranke */
```

```
        printf("Weitere Fehler ergeben sich f\"ur n >= %f, \n",
               ceil(sqrt(INT_MAX)) );
        /* siehe Kommentar 2.Schranke */

        printf("Geben Sie die obere Summationsschranke n an : ");
        fgets(tempstring, sizeof(tempstring), stdin);
        sscanf(tempstring, "%i", &n);

        for (i=1; i<=n; i++)
        {
               /* 1. Schranke f\"ur i */
               /* Der Summand 1.0/(i*i) wird bei der Addition */
               /* nich mehr ber\"ucksichtigt, falls 1.0/(i*i) < 1.0e-6 */

               /* 2. Schranke f\"ur i */
               /* Das Produkt i*i ist als int-Variable nicht */
               /* mehr darstellbar, falls i*i > INT_MAX */

               summe1=summe1+1.0/(i*i);
        }

        for (i=n; i>=1; i--)
        {
               summe2=summe2+1.0/i/i;
        }

        printf("Der erste Algorithmus liefert das Ergebnis : %f \n",summe1);
        printf("Der zweite Algorithmus liefert das Ergebnis : %f \n",summe2);
        return 0;
}
```

Das numerische Resultat in summe2 ist genauer, da dort zuerst alle kleinen Zahlen addiert werden, welche bei summe1 wegen der beschränkten Anzahl gültiger Ziffern keinen Beitrag zur Summation mehr liefern können. Gleichzeitig ist zu beachten, dass die Berechnung von i*i nicht mehr in int-Zahlen darstellbar ist für i*i > INT_MAX. Dagegen erfolgt die Berechnung 1.0/i/i vollständig im Bereich von Gleitkommazahlen. □

7.2.2 Abweisender Zyklus (while–Schleife)

Beim abweisenden Zyklus steht die Anzahl der Durchläufe nicht a–priori fest. Der Abbruchtest erfolgt vor dem Durchlauf eines Zyklus.

Die allgemeine Form ist

```
        while(logischer Ausdruck)
        {
               Anweisungen
        }
```

Beispiel 7.7 while–Schleife. Für eine beliebige Anzahl von Zahlen soll das Quadrat berechnet werden. Die Eingabeserie wird durch die Eingabe von 0 beendet.

```
#include <stdio.h>

int main()
{
```

```
        float zahl;
        char tempstring[80];

        printf("Geben Sie eine Zahl ein ('0' f\"ur Ende) : ");
        fgets(tempstring, sizeof(tempstring), stdin);
        sscanf(tempstring, "%f", &zahl);

        while (zahl != 0.0)
        {
                printf("%f hoch 2 = %f \n",zahl,zahl*zahl);
                printf("Geben Sie eine Zahl ein ('0' f\"ur Ende) : ");
                fgets(tempstring, sizeof(tempstring), stdin);
                sscanf(tempstring, "%f", &zahl);
        }
        return 0;
}
```

□

7.2.3 Nichtabweisender Zyklus (do–while–Schleife)

Beim nichtabweisenden Zyklus steht die Anzahl der Durchläufe nicht a-priori fest. Der Abbruchtest erfolgt nach dem Durchlauf eines Zyklus. Somit durchläuft der nichtabweisende Zyklus mindestens einmal die Anweisungen im Zykleninneren.

Die allgemeine Form ist

```
        do
        {
                Anweisungen;
        }
        while(logischer Ausdruck);
```

Beispiel 7.8 do–while–Schleife. Es wird solange eine Zeichenkette von der Tastatur eingelesen, bis die Eingabe eine nichtnegative ganze Zahl ist.

```
#include <stdio.h>
#include <math.h>   /* F\"ur pow */

int main()
{
        int  i, n, zahl=0;
        char text[100];

        do
        {
                printf("Geben Sie eine nichtnegative ganze Zahl ein : ");
                fgets(text, sizeof(text), stdin);
                sscanf(text, "%s",text);

                /* Es wird nacheinander gepr\"uft ob text[i] */
                /* eine Ziffer ist. Besteht die Eingabe */
                /* nur aus Ziffern, dann wird abgebrochen */
                i=0;
                while ('0' <= text[i] && text[i] <= '9')
                {       /* ASCII */
                        i=i+1;
                }
```

```
                if (text[i] != '\0')
                {
                        printf("%c ist keine Ziffer \n",text[i]);
                }
        }
        while (text[i] != '\0');

        /* Umwandlung von String zu Integer */
        n=i;  /* Die L\"ange des Strings == i */
        for (i=0;i<=n-1;i++)
        {
                zahl=zahl+ (text[i]-'0')*pow(10,n-1-i);
        }
        printf("Die Eingabe %s die nichtnegativen ganze Zahl %i\n",text,zahl);
        return 0;
}
```

Intern behandelt der Computer Zeichenkonstanten wie int-Variablen. Die Zuweisung erfolgt über die ASCII-Tabelle. So entsprechen z.B. die Zeichenkonstanten '0',...,'9' den Werten 48,..,57. □

7.3 Anweisungen zur unbedingten Steuerungsübergabe

- **break** Es erfolgt der sofortige Abbruch der nächstäußeren switch–, while–, do–while– oder for–Anweisung.
- **continue** Abbruch des aktuellen und Start des nächsten Zyklus einer while–, do–while– oder for–Schleife.
- **goto** *marke* Fortsetzung des Programms an der mit *marke* markierten Anweisung

Achtung!!! Die goto–Anweisung sollte sparsam (besser gar nicht) verwendet werden, da sie dem strukturierten Programmieren zuwider läuft und den gefürchteten Spaghetticode erzeugt. In den Übungen (und in der Klausur) ist die goto–Anweisung zur Lösung der Aufgaben **nicht erlaubt**.