ModProg 15-16, Vorl. 4

Richard Grzibovski

Nov. 11, 2015

Übersicht

Übersicht

1. Beispiele von 4.11.
2. Deklarationen und Typen (Teil 2)
3. Operationen (Teil 2)
4. Funktionen

```c
#include <stdio.h>
int main(void)
{
    int a,b,c;
    double x;
    a=1;              /* a=1 */
    a=9/8;            /* a=1, Integerdivision */
    a=3.12;           /* a=3, abrunden wegen int-Variable */
    a=-3.12;          /* a=-3 oder -4, Compiler abhaengig */
    b=6;              /* b=6 */
    c=10;             /* c=10 */
    x=b/c;            /* x=0 */
    x=(double) b/c; /* x=0.6  */
    x=(1+1)/2;        /* x=1 */
    x=0.5+1.0/2; /* x=1 */
    x=0.5+1/2;        /* x=0.5 */
    x=4.2e12;         /* x=4.2*10^{12} wiss. Notation */
    return 0;
}
```

Schleifen

Schleifen dienen dazu, die wiederholte Ausführung bestimmter Anweisungsblöcke zu steuern. Die betreffenden Anweisungsblöcke werden im Zusammenhang mit Schleifen auch *Schleifenrumpf* genannt.

`while`- und `do`-Schleifen

```
1  while( Ausdruck )
2       Schleifenrumpf
```

```
1  do
2       Schleifenrumpf
3  while( Ausdruck );
```

Beispiel

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

Beispiel

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

```c
#include <stdio.h> // <-- hier ist printf deklariert

int main(void)
{
    int a=12, r, j;

    printf("%d |_10 = ",a);
    j=1024*64; // j ist 2^16
    do {
        r=a/j;
        printf("%d",r);
        a = a - r*j;
        j = j/2;
    } while(j>=1);
    printf( " |_2\n");

    return 0;
}
```

```
1   #include <stdio.h> // <-- hier ist printf deklariert
2
3   int main(void)
4   {
5       int a=12, r, j;
6
7       printf("%d |_10 = ",a);
8       j=1024*64; // j ist 2^16
9       do {
10          r = a/j;
11          printf("%d",r);
12          a = a - r*j;
13          j = j/2;
14      } while(j>=1);
15      printf(" |_2\n");
16
17      return 0;
18  }
```

5 Man darf die Variable **deklarieren** und gleichzeitig **initialisieren**.

7,11 printf("%d",r); drückt die ganze Zahl `r` (Format "%d")

```c
#include <stdio.h> // <-- hier ist printf deklariert

int main(void)
{
    int a, r, j;

    printf("a= ");
    // einlesen a von der Tastatur
    scanf("%d",&a);
    printf("%d |_10 = ",a);
    j=1024*64; // j ist 2^16
    do {
        r=a/j;
        printf("%d",r);
        a = a - r*j;
        j = j/2;
    } while(j>=1);
    printf( " |_2\n");

    return 0;
}
```

```c
#include <stdio.h> // <-- hier ist printf deklariert

int main(void)
{
    float eps, meps; // Deklaration

    eps = 1.0f;
    // ausdrcken Namen fuer die Spalten:
    printf("eps                1 + eps\n");
    do {
        meps=eps;              // der Antwort ist eps
        eps = eps/2.0f;    // Reduzierung der eps
        // ausdruecken die Zwischenergebnisse:
        printf("%e  %.15e\n", eps, 1.0f + eps);
        // wenn 1+eps ist nicht gleich 1, widerholen.
    } while(1.0f + eps != 1.0f);

    // ausdruecken das Endnergebnis
    printf("meps= %.10e\n", meps);
    return 0; // die Funktion main ist abgeschlossen
}
```

```c
#include <stdio.h> // <-- hier ist printf deklariert

int main(void)
{
    float eps, meps; // Deklaration

    eps = 1.0f;
    // ausdruecken Namen fuer die Spalten:
    printf("eps              1 + eps\n");
    do {
        meps=eps;          // der Antwort ist eps
        eps = eps/2.0f;    // Reduzierung der eps
        // ausdruecken die Zwischenergebnisse:
        printf("%e  %.15e\n", eps, 1.0f + eps);
        // wenn 1+eps ist nicht gleich 1, widerholen.
    } while(1.0f + eps != 1.0f);

    // ausdruecken das Endnergebnis
    printf("meps= %.10e\n", meps);
    return 0; // die Funktion main ist abgeschlossen
}
```

5 **Deklaration** von Variablen `eps` und `meps`.
Die Variablen innerhalb der Funktion zum Speichern gleitpunktzahlige Werte verwendet werden können.

```c
#include <stdio.h> // <-- hier ist printf deklariert

int main(void)
{
    float eps, meps; // Deklaration

    eps = 1.0f;
    // ausdruecken Namen fuer die Spalten:
    printf("eps              1 + eps\n");
    do {
        meps=eps;          // der Antwort ist eps
        eps = eps/2.0f;    // Reduzierung der eps
        // ausdruecken die Zwischenergebnisse:
        printf("%e  %.15e\n", eps, 1.0f + eps);
        // wenn 1+eps ist nicht gleich 1, widerholen.
    } while(1.0f + eps != 1.0f);

    // ausdruecken das Endnergebnis
    printf("meps= %.10e\n", meps);
    return 0; // die Funktion main ist abgeschlossen
}
```

7,11,12 **Zuweisungen**

- die rechte Seite wird *ausgewertet*
- Ergebniswert wird in der links stehenden Variablen gespeichert

```c
#include <stdio.h> // <-- hier ist printf deklariert

int main(void)
{
    float eps, meps; // Deklaration

    eps = 1.0f;
    // ausdruecken Namen fuer die Spalten:
    printf("eps              1 + eps\n");
    do {
        meps=eps;          // der Antwort ist eps
        eps = eps/2.0f;    // Reduzierung der eps
        // ausdruecken die Zwischenergebnisse:
        printf("%e  %.15e\n", eps, 1.0f + eps);
        // wenn 1+eps ist nicht gleich 1, widerholen.
    } while(1.0f + eps != 1.0f);

    // ausdruecken das Endnergebnis
    printf("meps= %.10e\n", meps);
    return 0; // die Funktion main ist abgeschlossen
}
```

14 printf ("%e %.15e\n", eps, 1.0f + eps);

- Wir ausgeben die Zahle in exponentiellen Form (%e),
- die Zahl 1.0f + eps wird mit 15 Nachkommastellen ausgegeben.

Deklarationen und Typen (Teil 2)

Felder dienen dazu, mehrere Datenobjekte gleichen Typs zu einer Einheit zusammenzufassen (z.B. für Vektoren und Matrizen).

Datentyp Feldbezeichner[N];

Es wird ein Feld mit Namen Feldbezeichner der Länge N vom Typ Datentyp deklariert. Somit:

- Die Feldlänge ist dabei die Anzahl der Komponenten des Feldes (Feldeinträge).
- Die eckigen Klammern umschließen Feldlänge bei der Deklaration.
- Mit eckigen Klammern greift man auch auf die Komponenten des Feldes zu.

Dabei beachte man:

Die Nummerierung der Feldeinträge beginnt in C mit der 0. (!!!)

Beispiel:

```
1  int main()
2  {
3      float u[2], v[7]; // Deklaration
4      int i;
5
6      //Zuweisung fuer u
7      u[0] = 0.0;
8      u[1] = 10000.0 * 123.4;
9
10     i=0;
11     do {
12         v[i] = i*i; //Zuweisung fuer v
13         i = i + 1;
14     } while(i<7);
15
16     return 0;
17 }
```

Mehrdimensionale Felder

Datentyp Name[**Laenge1**][**Laenge2**]...[**LaengeN**];
Hierdurch wird ein N-dimensionales Feld vom Typ **Datentyp** deklariert. Der Zugriff auf die Komponenten erfolgt auch hier durch die entsprechende Verwendung von eckigen Klammern.

Beispiel: Durch

```
double A[2][3];
```

wird ein zweidimensionales Feld mit insgesamt 6 Einträgen vom Typ `double` deklariert. Die Reihenfolge der Einträge ist

A [0][0], **A** [0][1], **A** [0][2], **A** [1][0], **A** [1][1], **A** [1][2]

```
 1  int main()
 2  {
 3      float u[8][3]; // Deklaration
 4      int i;
 5
 6      i=0;
 7      do {
 8          u[i][0] = i*i;
 9          u[i][1] = i*i + 1.0;
10          u[i][2] = i*i + 2.0;
11          i = i + 1;
12      } while(i<8);
13
14      return 0;
15  }
```

```
 1  int main()
 2  {
 3      int u[8][80]; // Deklaration
 4      int i,j;
 5
 6      i=0;
 7      do {
 8          j=0;
 9          do {
10              u[i][j] = i*i + j;
11              j = j + 1;
12          } while(j<80);
13          i = i + 1;
14      } while(i<8);
15
16      return 0;
17  }
```

Initialisierung von Feldern.

Auch Felder können bereits bei der Deklaration mit Werten belegt werden. Dazu werden die geschweiften Klammern verwendet. Zum Beispiel

```
float u[3] = {1.0f, -1.2e+02f, 0.0f};
```

Durch die explizite Angabe aller Werte kann die Größenangabe entfallen.

```
float u[]  = {1.0f, -1.2e+02f, 0.0f};
```

Entsprechend kann man ein zweidimensionales Feld **A** zeilenweise durch

```
double A[][] = { {0.0, -1.2, 0.4}, {12.3, -2.99, 27.3} };
```

initialisieren.

Arithmetische Zuweisungsoperatoren

Benötigt man bei der Summation den Wert eines der Summanden nicht mehr, so kann der betreffende Variablenwert mit dem Summenwert uberschrieben werden, z.B.

$$\mathbf{a} = \mathbf{a} + \mathbf{b};$$

Da dies recht häufig vorkommt, stellt C als Abkürzung die so genannten *arithmetischen Zuweisungsoperatoren* bereit:

Operation	Bezeichnung	äquivalent zu
Op1 += **Op2**;	Additionszuweisung	**Op1** = **Op1**+**Op2**;
Op1 −= **Op2**;	Subtraktionszuweisung	**Op1** = **Op1**−**Op2**;
Op1 ∗= **Op2**;	Multiplikationszuweisung	**Op1** = **Op1**∗**Op2**;
Op1 /= **Op2**;	Divisionszuweisung	**Op1** = **Op1**/**Op2**;
Op1 %= **Op2**;	Modulozuweisung	**Op1** = **Op1**%**Op2**;

Inkrement- und Dekrementoperatoren

In vielen Programmen werden ganzzahlige Variablen zum Zählen verwendet, d.h. sie werden jeweils um 1 erhöht (inkrementiert) oder verringert (dekrementiert). Zur Abkürzung der entsprechenden Anweisungen bietet C den Inkrementoperator $++$ und den Dekrementoperator $--$. Beide Operatoren sind unär und existieren sowohl in *Präffix*- als auch in *Suffixform*.

Beispiel Für eine Variable `i` vom Typ `int` sind die fier folgenden Anweisungen äquivalent:

```
i = i + 1;
i += 1;
i++;
++i;
```

Inkrement- und Dekrementoperatoren: Rangfolge

Der Unterschied zwischen Suffix- und Präffixformen besteht in der Rangfolge relativ zu anderen Operatoren. So wird durch die Anweisung

sum += **i**++;

erst der Wert von **i** zum Wert in **sum** addiert und dann erst der Wert in **i** um 1 erhöht, wohingegen durch

sum += ++**i**;

erst **i** inkrementiert wird und danach der Wert von **i** zu **sum** addiert wird.

Beispiel

```
1  int main()
2  {
3      int u[8][80]; // Deklaration
4      int i,j;
5
6      i=0;
7      do {
8          j=0;
9          do {
10             u[i][j] = i*i + j;
11             j++;
12         } while(j<80);
13         i++;
14     } while(i<8);
15
16     return 0;
17 }
```

Beispiel

```
1  int main()
2  {
3      int u[8][80]; // Deklaration
4      int i,j;
5
6      i=0;
7      do {
8          j=0;
9          do {
10             u[i][j] = i*i + j;
11             ++j;
12         } while(j<80);
13         ++i;
14     } while(i<8);
15
16     return 0;
17 }
```

Beispiel

```
1  int main()
2  {
3      int u[8][80]; // Deklaration
4      int i,j;
5
6      i=0;
7      do {
8          j=0;
9          do {
10             u[i][j] = i*i + j++;
11
12         } while(j<80);
13         ++i;
14     } while(i<8);
15
16     return 0;
17 }
```

Beispiel

```
 1  int main()
 2  {
 3      float v[7];
 4      int i;
 5
 6      i=0;
 7      do {
 8          v[i] = i*i;
 9          i = i + 1;
10      } while(i<7);
11
12      return 0;
13  }
```

Beispiel

```
1  int main()
2  {
3      float v[7];
4      int i;
5
6      i=0;
7      do {
8          v[i++] = i*i;
9
10     } while(i<7);
11
12     return 0;
13 }
```

Funktionen

C ist eine so genannte prozedurale Programmiersprache, d.h. C unterscheidet zwischen den Daten einerseits und den Funktionen, die die Daten verarbeiten, andererseits

```
C-Programm = Daten + Funktionen
```

Gründe für den Einsatz von Funktionen

- Programmiertechnische Umsetzung der Partitionierung des Problems: ein Mal implementieren, lebenslang benutzen.
- Aufteilung der Entwicklungsarbeit.
- Ubersichtlichkeit des Quelltextes.

Deklaration von Funktionen

Wie Variablen auch, müssen Funktionen *vor ihrer ersten Verwendung* deklariert werden.

Rueckgabetyp Funkname(**Datentyp1 Bez1**,..., **DatentypN BezN**);

- Die Liste von Deklarationen in Klammern heißt auch Parameterliste. Die Parameterbezeichner sind dabei optional, die Typen müssen jedoch stets angegeben werden.
- Ist die Parameterliste leer, so gilt sie als unbestimmt.
- Dass eine Funktion keine Argumente besitzt, wird bei der Deklaration dadurch ausgedrückt, dass man `void` als Parameter angibt.

Beispiele

Durch

```
float mwert(float a, float b);
```

wird eine Funktion namens **mwert** deklariert, die einen Wert vom Typ `float` zurückliefert und zwei `float`-Argumente entgegennimmt. Äquivalent:

```
float mwert(float, float);
```

Weitere Beispiele:

```
float tuwas(int, double);
```

```
double finde_eps(void);
```

Funktionsaufrufe

- Beim Funktionsaufruf muss die Reihenfolge der Argumentdatentypen mit den Angaben der Parameterliste in der Deklaration übereinstimmen.
- Ist ein Rückgabewert vorhanden, so können die entsprechenden Funktionsaufrufe als Ausdruck mit Operatoren behandelt werden.
- Funktionen ohne Rückgabewert (Typ `void`) sind als Ausdruck unbestimmt.
- Wird eine Funktion nicht explizit deklariert, so wird eine *implizite Deklaration vorgenommen*: Wird nichts anderes angegeben, so gilt als Rückgabetyp stets `int`. Die Argumentliste wird als leer, d.h. als unbestimmt, angenommen.

Definition von Funktionen

Die Definition einer Funktion ist nichts anderes als der vollständige Quelltext des Unterprogramms. Sie hat folgende Gestalt

```
1  Datentyp Funkname(Datentyp1 Bez1, ..., DatentypN BezN)
2  {
3      Anweisungsblock (Funktionsrumpf)
4      ...
5  }
```

- Die erste Zeile der Definition wird **Funktionskopf** genannt.
- Bei der Definition müssen die Parameterbezeichner angegeben werden.
- Nach dem Funktionskopf folgt der **Funktionsrumpf**.
- Wie realisiert man das Zurückliefern des Rückgabewerts? Hierfür existiert in C das Schlüsselwort `return`.

Beispiel

```
1  #include <stdio.h>
2  int quadrat(int x); //Deklaration
3
4  // Definition
5  int quadrat(int x) {
6      return x*x;
7  }
8
9  int main(){
10    printf("%i\n",quadrat(5));              // Aufruf
11    printf("%i\n",quadrat(10));             // Aufruf
12    printf("%i\n",quadrat(0));              // Aufruf
13    printf("%i\n",quadrat(-12));            // Aufruf
14    printf("%i\n",quadrat(quadrat(8)));     // Aufruf
15    return 0;
16 }
```

Beispiel

```
1  #include <stdio.h>
2  int quadrat(int x); //Deklaration
3
4  int main(){
5     printf("%i\n",quadrat(5));                 // Aufruf
6     printf("%i\n",quadrat(10));                // Aufruf
7     printf("%i\n",quadrat(0));                 // Aufruf
8     printf("%i\n",quadrat(-12));               // Aufruf
9     printf("%i\n",quadrat(quadrat(8)));        // Aufruf
10    return 0;
11 }
12
13 // Definition
14 int quadrat(int x) {
15      return x*x;
16 }
```

Deklaration: lokale und globale Variablen

- In einem Block $\{...\}$ gemachte Deklaration sind nur innerhalb dieses Blocks gültig. Deswegen, heißen sie **lokale** Variablen.
- Man darf Variablen auch außerhalb von Funktionen und Blöcke deklarieren.
- Diese, **globale** Variablen, sind dann tatsächlich für alle Funktionen benutzbar (und existieren dann nur einmal).
- Wenn die lokale Variablen die gleiche Bezeichnung wie globale haben, "verdecken" die lokale Werte ihre "globale" Konkurrenten.

Beispiel

```
1  #include <stdio.h>
2  int y = 1;
3  int f1();
4  int f1(){
5      y=y+1;
6      return y;
7  }
8
9  int main(){
10     y = f1();
11     printf("%i\n",y);
12     y = f1();
13     printf("%i\n",y);
14     y = f1();
15     printf("%i\n",y);
16     return 0;
17 }
```

Beispiel: Verdeckung

```
 1  #include <stdio.h>
 2  int a = 3 ;
 3  int main( void ) {
 4      int b = 1 ;
 5      printf( "Ausg. 1: a = %d, b = %d\n", a, b ) ;
 6      {   int b = 10 ;
 7          {   int a ;
 8              a = b + 1 ;
 9              printf( "Ausg. 2: a = %d, b = %d\n", a, b )
10          }
11          b = b + 1 ;
12          a = b + 1 ;
13          printf( "Ausg. 3: a = %d, b = %d\n", a, b ) ;
14      }
15      printf( "Ausg. 4: a = %d, b = %d\n", a, b ) ;
16      { printf( "Ausg. 5: a = %d, b = %d\n", a, b ) ; }
17      return 0 ;
18  }
```