ModProg 15-16, Vorl. 5

Richard Grzibovski

Nov. 18, 2015

Übersicht

Übersicht

1. Logische Operationen
2. Priorität
3. Verzweigungen
4. Schleifen

Logische Werte

Hauptkonzept:

Besitzt ein C-Ausdruck den Wert Null (egal in welcher Form), so gilt er im logischen Sinne als falsch, andernfalls als wahr.

- Der einfachste logische Ausdruck ist eine ganze Zahl.
  - $= 0$ Ist die Zahl **gleich Null**, wird der Ausdruck als logischen Wert **falsch** interpretiert.
  - $\neq 0$ Ist die Zahl **ungleich Null**, wird der Ausdruck als logischen Wert **wahr** interpretiert.
- Ergebnisse von logischen Operationen sind gleich **1** für **wahr** und gleich **0** für **falsch**.

Vergleichende und logische Operatoren

Operator	Beispiel	Bedeutung
<	k < 3	kleiner als
>	k > 3	größer als
<=	k <= 3	kleiner als oder gleich
>=	k >= 3	größer als oder gleich
==	k == 3	gleich
!=	k != 3	ungleich
!	!i	logische Negation
&&	i && j	logisches UND
||	i || j	logisches ODER

Die Logische Operationen sind für alle elementare Datentypen definiert.

Warnung: Operationen == und != sind für Gleitpunktzahlen instabil.

Logische Operatoren

&&	0	1
0	0	0
1	0	1

||	0	1
0	0	1
1	1	1

Beispiel:

```
int A = -4, C = 1;
double B = -0.5;
```

Ausdruck	Wert
A	−4
!A	0
A−B	−3.5
!(A−B)	0
!!(A −B)	1
!A − B	0.5
!A − !B	0

Ausdruck	Wert
A<B<C	0
A<B<=C	1
A<(B<C)	1
A<B && B<C	1
C>A && B	1
A==2 || B>C	0
A=2 || B>C	1

Beispiel

```
1  int main()
2  {
3      int i = 17, j;
4      j = !i;    /* ergibt 0 (falsch),
5                    da i ungleich 0 (wahr) ist */
6
7      i = !j;    /* ergibt 1 (wahr),
8                    da j gleich 0 (falsch) ist   */
9      return 0;
10 }
```

Beispiel

```
 1  int main()
 2  {
 3      double x, y, z;
 4      int wert;
 5      x = 0.0;
 6      y = -1.5;
 7      z = 3.0;
 8
 9      wert = x;                          /* 0 (falsch) */
10      wert = !x;                         /* 1   (wahr) */
11      wert = y<0.0;                      /* 1          */
12      wert = y*z >= 0.0;                 /* 0          */
13      wert = (z < 0.0) || (x == 0);      /* 1          */
14      wert = (z > 0.0) && !x;            /* 1          */
15      wert = (z >= 0.0) && x;            /* 0          */
16
17      return 0;
18  }
```

Warnungen

Warnung 1

Man verwechsele nicht den Ausdruck **a=b** mit **a==b**. Letzterer ist nur bei Gleichheit zutreffend, wohingegen ersterer immer zutrifft, solange nur **b** einen von Null verschiedenen Wert besitzt.

Warnung 2

In C wird die Verarbeitung eines logischen Ausdrucks sofort beendet, sobald dessen Wert feststeht.

```c
int main()
{
    int i=0, j=1, k=2;
    int wert;

    /* zur Warnung 1 */
    wert = (i = k); /* ergibt 2
                       auch i ist gleich 2 */
    wert = (j == k); /* ergibt 0
                        da 1 ungleich 2 ist */

    /* zur Warnung 2 */
    printf("j= %i\n",j);
    wert = ( 0 && ( j = k ));
    /* wert ist 0, da (0 && m) Null fuer alle m ist.*/
    /* auch die Zuweisung j=k ist nicht ausferuert! */
    printf("j= %i\n",j);
    return 0;
}
```

Regeln zum Bilden von Ausdrücken

Priorität	Operator	Reihenfolge
1 högste	[] () . −>` ++ −− (postfix) { }	links nach rechts
2	++ −− (prefix) `sizeof` ~ (bitweise Negation) ! − + (als Vorzeichen) & (Adressoperator) ∗ (Variablenoperator)	rechts nach links

Regeln zum Bilden von Ausdrücken

Priorität	Operator	Reihenfolge
3	(**type name**) (cast)	rechts nach links
4	$*$ / \%	links nach rechts
5	$+$ $-$	links nach rechts
6	<< >>	links nach rechts
7	< > <= >=	links nach rechts
8	== !=	links nach rechts
9	& (bitweises UND)	links nach rechts
10	^ (bitweises Exklusiv-ODER)	links nach rechts
11	| (bitweises ODER)	links nach rechts
12	&& (logisches UND)	links nach rechts
13	|| (logisches ODER)	links nach rechts
14	?:	rechts nach links
15	alle Zuweisungen wie =, +=, −=, ...	rechts nach links
16	,	links nach rechts

Regeln zum Bilden von Ausdrücken

- Auf gleicher Prioritätsstufe wird ein Ausdruck von links nach rechts abgearbeitet. Dies wird auch linksassoziativ genannt.
- Ausnahme: Die unären und Zuweisungsoperatoren werden von rechts nach links abgearbeitet (rechtsassoziativ).
- Generell werden aber zuerst immer die Klammern ausgewertet.

```c
int main()
{
    int i=0, j=1, k=2, m=3;
    int wert;

    i = j + k + m; /* ist gleich mit */
    i = ((j + k) + m);

    i = j = k = m; /* ist gleich mit */
    i = (j = (k = m));

    wert = (j == k); /* ist gleich mit */
    wert =  j == k ;

    return 0;
}
```

Nichtrivialer Beispiel, Warnung

Restriktion

"A scalar object shall have its stored value modified at most once by the evaluation of an expression." aus C-Reference

```
1  #include<stdio.h>
2  int main()
3  {
4      int Teil = 0, Summe;
5      Summe = (Teil = 3) + (++Teil);
6      printf("Summe = %i\n",Summe);
7      return 0;
8  }
```

Mit dem gcc kompilierte Programm druckt 8 (!!!).
Mit dem icc kompilierte Programm druckt 7.

`if`-Anweisung und bedingter Ausdruck

Die `if-else`-Anweisung dient dazu, bestimmte Anweisungen in Abhängigkeit davon auszuführen, ob eine gewisse Bedingung erfüllt ist oder nicht. Die Struktur der Anweisung ist die folgende:

```
1 if( Ausdruck )
2     Anweisungsblock1
3 else
4     Anweisungsblock2
```

Ist der in Klammern gesetzte **Ausdruck** wahr, so werden die Anweisungen in **Anweisungsblock1** ausgeführt,
andernfalls die Anweisungen in **Anweisungsblock2**.

Der `else`-Teil ist optional, wie wir im folgenden Beispiel sehen.

Beispiele

Die Funktion betrag bestimmt $|x|$.

```
1  double betrag(double x) {
2      if(x<0.0)
3          return -x;
4      return x;
5  }
```

Die Funktion maximum bestimmt die maximale Zahl

```
1  double maximum(double x, double y) {
2      double mxy;
3      if(x>y)
4          mxy=x;
5      else
6          mxy=y;
7      return mxy;
8  }
```

Verzweigungen

Warnung

Soweit nicht durch geschweifte Klammern etwas anderes angegeben ist, bezieht sich eine `else`-Anweisung immer auf das letzte `if`, für das noch kein `else`-Teil existiert.

Das bedeutet, dass die linke Anweisung zu unterscheiden ist von die Anweisung rechts.

```
1  if(   )
2
3      if(   )
4          Block1
5
6  else
7      Block2
```

```
1  if(   )
2  {
3      if(   )
4          Block1
5  }
6  else
7      Block2
```

Der bedingte Ausdruck

Der bedingte Ausdruck ist der einzige ternäre Operator in C.
Seine Struktur ist

Ausdruck1 ? **Ausdruck2** : **Ausdruck3**

Ist **Ausdruck1** wahr, so ist der Wert des bedingten Ausdrucks gleich **Ausdruck2**, andernfalls gleich **Ausdruck3**.

Damit kann das Maximum zweier Variablenwerte als Einzeiler geschrieben werden:

max = (**a** < **b**) ? **b** : **a**;

Fallunterscheidung: `switch`-Anweisung

```
1  switch (Ausdruck)
2  {
3      case Wert1: Anweisungen1;
4      break;
5      case Wert2: Anweisungen2;
6      break;
7      case Wert3: Anweisungen3;
8      break;
9      ...
10     default:    Ersatzanweisungen;
11 }
```

Der Datentyp von Ausdruck muss eine ganze Zahl (`int`) sein.

Fallunterscheidung: `switch`-Anweisung

- Die Werte hinter dem `case` müssen konstante Werte sein.
- Mit Hilfe des `break`; wird die `switch`-Anweisung beendet. Andernfalls werden die Anweisungen der weiteren `case`-Zeilen auch durchgeführt.

Beispiel

```
1  #include <stdio.h>
2  int main()
3  {
4      int k,m,fall;
5      scanf("%i %i", &k,&m);
6      fall=( k  < m )? 0 : 1    ;
7      fall=( k == m )? 2 : fall;
8      switch (fall)
9      {
10         case 0:
11         case 2: printf("k<=m\n");
12         break;
13         case 1: printf("k>m\n");
14         break;
15         default: printf("unerwartete Ereignis!\n");
16     }
17     return 0;
18 }
```

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       double x;
5       printf(" Geben Sie eine positive Zahl ein : ");
6       scanf(" % lf " ,& x );
7       while (x <=0)
8       {
9          printf("\nGeben Sie eine positive Zahl ein : ");
10         scanf(" % lf " ,& x );
11      }
12      return 0;
13 }
```

Beispiel

```
1  #include <stdio.h>
2  int main()
3  {
4      double x;
5      do {
6         printf("\nGeben Sie eine positive Zahl ein : ");
7         scanf( " % lf " ,&x );
8      } while(x <= 0);
9
10     return 0;
11 }
```

for-Schleifen

Die for-Schleife hat die Struktur

```
1   for (Initialisierung ; Erhaltungsbedingung ; Update )
2           Schleifenrumpf
```

- Die Initialisierung wird nur zu Beginn ausgeführt.
- Der Schleifenrumpf wird so lange ausgeführt, wie die Erhaltungsbedingung erfüllt ist. Die Uberprüfung findet zu Beginn des Blocks statt.
- Die Update-Anweisung wird jeweils nach Verarbeitung des Anweisungsblocks ausgeführt.

Gleichwertigkeit

Die `for`-Schleife

```
1   for (Initialisierung ; Erhaltungsbedingung ; Update )
2         Schleifenrumpf
```

ist äquivalent zur `while`-Schleife

```
1   Initialisierung
2   while ( Erhaltungsbedingung ) {
3         Schleifenrumpf
4         Update
5   }
```

Beispiel

```
1  double maxelem(int N, double x[]) {
2   int i;
3   double erg=x[0];
4   for(i=0; i<N ; i++)
5         if(erg<x[i]) erg=x[i];
6   return erg;
7  }
```

Steuerung von Wiederholungen

`break`	beendet **aktuelle** Wiederholungsanweisung
`continue`	Rest der Scheleife wird übersprungen und der nächste Scheleifendurchlauf gestartet

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

Beispiel

```
1  int suche_i(int N, int v[], int i) {
2   int j, erg= -1;
3   for(j=0; j<N ; j++) {
4           if(v[j]==i) {
5                   erg=j;
6                   break;
7           }
8   }
9   return erg;
10 }
```