Haute école d'ingénierie et d'architecture Fribourg
Hochschule für Technik und Architektur Freiburg

HTA-FR – Kurs Telekommunikation

Verfasser:
D. Gachet / HTA-FR - Telekommunikation

Embedded systems 1 und 2

Programmiersprache C – Die Befehle

Klasse T-2 // 2018-2019

Inhalt

- Ausdrücke
- Schleifen
- Iterationen
- Bedingungen
- Besonderheiten

Ausdruck - Definition

Ausdrücke sind "mathematische Sätze", die aus Operanden und Operatoren bestehen. Zum Beispiel:

```
x+2/3 - x < y ? x : y % (int)z << f(u)
```

Operanden sind numerische Werte, die von einer Variablen, einer Konstanten, einer Funktion oder einem Ausdrucksoperator stammen.

Die Auswertung eines Ausdrucks erfolgt in einer durch die Sprache vorgegebenen Reihenfolge und einem vordefinierten Vorrang. In der nachstehenden Tabelle sind diese Regeln zusammengefasst.

Note 1:
Parentheses are also used to group subexpressions to force a different precedence; such parenthetical expressions can be nested and are evaluated from inner to outer

Note 2:
Postfix increment/decrement have high precedence, but the actual increment or decrement of the operand is delayed (to be accomplished sometime before the statement completes execution). So in the statement `y = x * z++;` the current value of `z` is used to evaluate the expression (*i.e.*, `z++` evaluates to `z`) and `z` only incremented after all else is done.

Operator	Description	Associativity
() [] . -> ++ --	Parentheses (function call) (see Note 1) Brackets (array subscript) Member selection via object name Member selection via pointer Postfix increment/decrement (see Note 2)	left-to-right
++ -- + - ! ~ (*type*) * & sizeof	Prefix increment/decrement Unary plus/minus Logical negation/bitwise complement Cast (change *type*) Dereference Address Determine size in bytes	right-to-left
* / %	Multiplication/division/modulus	left-to-right
+ -	Addition/subtraction	left-to-right
<< >>	Bitwise shift left, Bitwise shift right	left-to-right
< <= > >=	Relational less than/less than or equal to Relational greater than/greater than or equal to	left-to-right
== !=	Relational is equal to/is not equal to	left-to-right
&	Bitwise AND	left-to-right
^	Bitwise exclusive OR	left-to-right
|	Bitwise inclusive OR	left-to-right
&&	Logical AND	left-to-right
||	Logical OR	left-to-right
?:	Ternary conditional	right-to-left
= += -= *= /= %= &= ^= |= <<= >>=	Assignment Addition/subtraction assignment Multiplication/division assignment Modulus/bitwise AND assignment Bitwise exclusive/inclusive OR assignment Bitwise shift left/right assignment	right-to-left
,	Comma (separate expressions)	left-to-right

Zuweisung

Die Zuweisung hat folgende Form:

```
variable = expression;
```

z. B.

```
var1 = 10;                  var2 = (10 * 25) + 1;
var3 = var2 + 76;           var4 = function(10);
```

Die Zuweisung einer Variablen lässt sich vereinfachen, wenn der rechte und der linke Operand gleich sind.

```
variable = variable operator expression;
```

In diesem Fall nimmt die Zuweisung folgende Form an:

```
variable operator= expression;
```

z. B.

```
var1 += 10;                 → var1 = var1 + 10
var2 -= (10 * 25) + 1;      → var2 = var2 - (10 * 25) + 1;
var3 *= var2 + 76;          → var3 = var3 * (var2 + 76);
var4 %= function(10);       → var4 = var4 % (function(10));
```

Liste der unterstützten Operatoren: `+ - * / % << >> & ^ ¦`

Zuweisung (II)

Es existiert eine 3. Form der Zuweisung, die das Inkrementieren/Dekrementieren einer Variablen um 1 erlaubt.

- **Inkrementierungsoperator:** `++`
- **Dekrementierungsoperator:** `--`

Sie können als Präfix (`++variable`) oder Suffix (`variable++`) benutzt werden. Wird er als Präfix eingesetzt, wird die Variable zuerst aktualisiert (inkrementiert oder dekrementiert) und anschliessend benutzt. Wird er als Suffix eingesetzt, wird die Variable zuerst benutzt und anschliessend aktualisiert.

z. B.

```
n = 10;
x = --n;          → n ==  9 and x == 9
n = 17;
x = n++;          → n == 18 and x == 17
n = 8;
s[n++] = 8;       → n ==  9 and s[8] == 8
n++;              → n == 10
s[--n] = 6;       → n ==  9 and s[9] == 6
```

Lokaler Block

C erlaubt, die Deklarationen und Befehle im gleichen Block zusammenzufassen, der durch geschweifte Klammern `{` und `}` begrenzt wird.

z. B.

```
{
   int var1 = 10;
   int var2 = 20;
   int var3 = --var1 * sin (var2++);
}
```

Alle in einem Block enthaltenen Deklarationen und Befehle werden aus syntaktischer Sicht wie ein einziger und eindeutiger Ausdruck/Befehl angesehen.

while - Schleife

C bietet eine `while`-Schleife, die es erlaubt, eine Befehlsausführung so oft zu wiederholen, bis eine Bedingung erfüllt ist. Der Befehl `while` hat die folgende Form:

```
while (expression)
    statement
```

Wenn die Auswertung des oben stehenden Ausdrucks `expression` nicht null (`true`) ergibt, wird der Befehl `statement` ausgeführt und der Ausdruck `expression` neu ausgewertet.
Dieser Zyklus wiederholt sich, bis der Ausdruck `expression` null (`false`) wird.

z. B.

```
char string1[] = "hello the world!"
char string2[32];
int  i = 0;
while (string1[i] != '\0')
    string2[i] = string1[i++];
string2[i] = '\0';
```

for - Schleife

C bietet eine `for`-Schleife, die es erlaubt, eine Befehlsausführung so oft zu wiederholen, bis eine Bedingung erfüllt ist. Der Befehl `for` hat die folgende Form:

```
for (init_expr; cond_expr; loop_expr)
   statement
```

Die Schleife `for` entspricht:

```
init_expr;
while (cond_expr) {
   statement
   loop_expr;
}
```

Typisches Beispiel für die Verwendung der Schleife `for`:

```
char string1[] = "hello the world!";
char string2[32];
int  i;
for (i=0; i<32; i++)
  string2[i] = string1[i];
```

do-while-Schleife

C bietet eine `do-while`-Schleife, die es erlaubt, eine Befehlsausführung so oft zu wiederholen, bis eine Bedingung erfüllt ist. Der Befehl `do-while` hat die folgende Form:

```
do {
    statement
} while (expression);
```

Wenn die Auswertung des oben stehenden Ausdrucks `expression` nicht null (`true`) ergibt, wird der Befehl `statement` ausgeführt und der Ausdruck `expression` neu ausgewertet.
Dieser Zyklus wiederholt sich, bis der Ausdruck `expression` null (`false`) wird. Im Gegensatz zur `while`-Schleife wird der Befehl `statement` mindestens einmal ausgeführt.

z. B.

```
char string1[] = "hello the world!"
char string2[32];
int  i = 0;
do {
    string2[i] = string1[i];
} while (string1[i++] != '\0');
```

Endlosschleife

In C existiert kein spezifischer Befehl, um Endlosschleifen zu erzeugen. Sie lassen sich dagegen leicht mithilfe einer `for`- oder `while`-Schleife erzeugen. Sie hat die folgende Form:

```
while(1)          for(;;)
  statement         statement
```

Der Befehl `break` erlaubt den Ausstieg aus der Schleife.

z. B.

```
while(1) {
    if (expression1) break;
    if (expression2) break;
   statement
}
```

Endlosschleifen erweisen sich bei der Auswertung sehr komplexer Ausdrücke als sehr nützlich.

Achtung:

Dieser Schleifentyp darf den andern nur vorgezogen werden, wenn dadurch die Lesbarkeit des Codes verbessert wird.

if-else-Bedingung

Die Bedingung `if-else` erlaubt, Alternativen zu beschreiben und den Befehlsfluss eines Programms zu steuern. Sie hat die folgende Form:

```
if (expression)
  statement1
else
  statement2
```

Beachten Sie, dass der `else`-Teil optional ist.

Wenn die Auswertung des oben stehenden Ausdrucks `expression` nicht null (`true`) ergibt, wird der Befehl `statement` ausgeführt. Ergibt sie dagegen null (`false`) und es ist ein `else`-Teil vorhanden, wird der Befehl `statement2` ausgeführt.

Die Bedingung `if-else` lässt sich in eine Mehrweg-Entscheidung verallgemeinern, indem einfach mehrere Befehle `if-else` hintereinander aufgereiht werden. Sie hat die folgende Form:

```
if (expression1)       statement1
else if (expression2)  statement2
else if (expression3)  statement3
else                   statement4
```

Bedingung - ? :

Der Ternäroperator `"?:"` bietet eine sehr kompakte Form der Bedingung `if-else.` Sie hat die folgende Form:

```
expr1 ? expr2 : expr3
```

Das bedeutet: Wenn der Ausdruck `expr1` wahr (true) ist, wird der Ausdruck `expr2` ausgewertet, andernfalls der Ausdruck `expr3`.

Als Beispiel für den folgenden Fall:

```
if (a > b) z = a; else z = b;
```

diese Befehle lassen sich reduzieren auf:

```
z = (a > b) ? a : b;
```

Indexierter Sprung - switch

C bietet eine indexierte Sprunganweisung `switch`, die erlaubt, auf sehr einfache Art Mehrwegbedingungen zu beschreiben. Sie hat die folgende Form:

```
switch (expression) {
  case const-int-expr1: statements
  case const-int-expr2: statements
  default: statements
}
```

Das Programm führt seine Ausführung mit dem Befehl fort, der auf das Label folgt, das der Auswertung des Ausdrucks `expression` entspricht. Die Labels müssen zwingend Ganzzahl-Konstanten sein.

Wenn kein Label der Auswertung des Ausdrucks `expression` entspricht, setzt das Programm seine Ausführung mit dem Befehl fort, der auf das optionale Label `default` folgt. Es ist indessen gute Programmierpraxis, das Label einzufügen.

Der Befehl `break` erlaubt den Ausstieg aus der Bedingung.

z. B.

```
switch (string[i]) {
  case 'a': string[i] = 'A'; break;
  case 'b': string[i] = 'B'; break;
  case '0':
  case '1': string[i] = '.'; break;
  default : string[i] = '?'; break;
}
```

Besonderheiten

Die Kontrolle des Operationsflusses eines Programms kann mithilfe der folgenden Befehle geändert werden:

- **`break`:**
  beendet den Iterationsbefehl `for`, `while` und `do-while` und die Bedingung `switch`, in der er erscheint.

- **`continue`:**
  übergibt die Kontrolle an die nächste Iteration des Befehls `for`, `while` und `do-while`, in der er erscheint.

- **`goto <label>` :**
  Bedingungsloser Sprung zum Befehl, der durch das Label markiert ist.

- **`return`:**
  beendet die Ausführung einer Funktion bedingungslos.

Achtung:

Mit Ausnahme des Befehls **`break`**, der zwingend mit der Bedingung **`switch`** oder einer Endlositeration und einem Befehl benutzt werden muss, der die Rückgabe des Wertes einer Funktion erlaubt, sollten alle diese Befehle vermieden werden, da sie die Lesbarkeit des Codes massiv verschlechtern und die Fehlersuche (das Debugging) sehr erschweren.