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

Systèmes Embarqués 1 & 2

a.04 – C – Les instructions

Classes T-2/I-2 // 2018-2019

Hes·so
Haute Ecole Spécialisée de Suisse occidentale
Fachhochschule Westschweiz

Daniel Gachet | HEIA-FR/TIC
a.04 | 10.09.2018

Contenu

- Expressions
- Boucles
- Itérations
- Conditions
- Particularités

Expression - définition

Les expressions sont des « phrases mathématiques » composées d'opérandes et d'opérateurs. Par exemple:

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

Les opérandes sont des valeurs numériques fournies par une variable, une constante, une fonction ou un opérateur de l'expression.

L'évaluation d'une expression se fait dans un ordre et une précédence prédéfini par le langage. Le tableau ci-dessous résume ces règles.

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

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.

L'assignation a la forme suivante:

```
variable = expression;
```

p.ex.

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

L'assignation d'une variable peut être simplifiée, si l'opérande de droite et la même que celle de gauche.

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

dans ce cas l'assignation prend la forme suivante :

```
variable operator= expression;
```

p.ex.

```
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 des opérateurs supportés : `+ - * / % << >> & ^ ¦`

Il existe une 3ème forme d'assignation permettant l'incrémentation/décrémentation par 1 d'une variable.

- **Opérateur d'incrémentation :** `++`
- **Opérateur de décrémentation :** `--`

Ils peuvent être utilisés comme préfixe (`++variable`) ou comme suffixe (`variable++`). Si il est employé comme préfixe, la variable est d'abord mise à jour (incrémentée ou décrémentée) et ensuite utilisée. Si il est employé comme suffixe, la variable est d'abord utilisé et ensuite mise à jour.

p.ex.

```
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
```

Bloc local

C permet de regrouper des déclarations et des instructions and dans un même bloc délimité par des accolades { et }.

p.ex.

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

Toutes les déclarations et instructions contenues dans un bloc sont considérées, du point de vue syntaxique, comme une seule et unique expression/instruction.

Boucle - while

C propose une boucle `while` permettant de répéter l'exécution d'un instruction jusqu'à ce qu'une condition soit remplie. L'instruction `while` prend la forme suivante :

```
while (expression)
   statement
```

Si l'évaluation de l'expression ci-dessus `expression` est non nulle (`true`), l'instruction `statement` est exécutée et l'expression `expression` est réévaluée. Ce cycle continue jusqu'à ce que l'expression `expression` soit zéro (`false`).

p.ex.

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

C propose une boucle `for` permettant de répéter l'exécution d'un instruction jusqu'à ce qu'une condition soit remplie. L'instruction `for` prend la forme suivante:

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

La boucle `for` est équivalente à:

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

Exemple typique d'utilisation de la boucle `for` :

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

Boucle - do-while

C propose une boucle `do-while` permettant de répéter l'exécution d'un instruction jusqu'à ce qu'une condition soit remplie. L'instruction `do-while` prend la forme suivante :

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

Si l'évaluation de l'expression ci-dessus `expression` est non nulle (`true`), l'instruction `statement` est exécuté et l'expression `expression` est réévaluée. Ce cycle continue jusqu'à ce que l'expression `expression` soit zéro (`false`). Contrairement à la boucle `while`, l'instruction `statement` est exécuté au moins une fois.

p.ex.

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

Boucle - sans fin

En C il n'existe aucune instruction spécifique pour construire des boucles sans fin. Par contre celles-ci peuvent facilement être créées à l'aide de la boucle `for` ou `while`. Elle prend la forme suivante:

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

L'instruction `break` permet de sortir de la boucle.

p.ex.

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

Les boucles sans fin peuvent être très pratique si des expressions très complexes doivent être évaluées.

Attention:

Ce type boucle ne doit être privilégié aux autres que si la lisibilité du code en est augmentée.

La condition `if-else` permet de décrire des alternatives et de contrôler le flux d'instructions d'un programme. Elle prend la forme suivante:

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

Remarquer que la partie `else` est optionnelle.

Si l'évaluation de l'expression `expression` ci-dessus est non nulle (`true`), l'instruction `statement1` est exécutée. Par contre, si elle est zéro (`false`) et qu'il existe une partie `else`, alors c'est l'instruction `statement2` qui sera exécutée.

La condition `if-else` peut être généralisée en une décision multivoie en cascadant simplement les instructions `if-else`. Elle prend la forme suivante :

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

L'opérateur ternaire `"?:"` propose une forme très compacte de la condition `if-else.` Elle prend la forme suivante:

```
expr1 ? expr2 : expr3
```

Signifiant que si l'expression `expr1` est vraie (true) l'expression `expr2` est évaluée, sinon c'est l'expression `expr3` qui le sera.

Par exemple pour le cas suivant:

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

ces instructions peuvent être réduites à :

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

Saut indexé - switch

C propose un saut indexé `switch` permettant de décrire très simplement des conditions multivoies. Elle prend la forme suivante:

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

Le programme poursuit son exécution à l'instruction suivant le label correspondant à l'évaluation de l'expression `expression`. Les labels doivent impérativement être des constantes entières.

Si aucun label ne correspond à l'évaluation de l'expression `expression`, le programme poursuivra son exécution à l'instruction placée après le label `default`, lequel est optionnel. Il est cependant de bonne facture de l'implémenter.

L'instruction `break` permet de sortir de la condition.

p.ex.

```
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;
}
```

Particularités

Le contrôle du flux des opérations d'un programme peut être modifié à l'aide des instructions suivantes:

- **`break`:**
  termine l'instruction d'itération `for`, `while` et `do-while` et de condition `switch` la plus proche dans laquelle elle apparait.

- **`continue`:**
  passe le contrôle à la prochaine itération de l'instruction `for`, `while` et `do-while` plus proche dans laquelle elle apparait .

- **`goto <label>` :**
  saute sans condition à l'instruction marquée par le label.

- **`return`:**
  termine sans condition l'exécution d'une fonction.

Attention:

A l'exception de l'instruction **`break`**, devant impérativement être utilisée avec la condition **`switch`** ou l'itération sans fin, et de l'instruction permettant de retourner la valeur d'une fonction, toutes ces instructions doivent être évitées car elles diminuent grandement la lisibilité du code et rendent son débogage très difficile.