

Annexe Langage C/C++

Les instructions conditionnelles et itératives

Thierry Vaira

BTS SN-IR Avignon

tvaira@free.fr © v0.1



Instruction conditionnelle (1/2)

- Il en existe deux formes possibles :

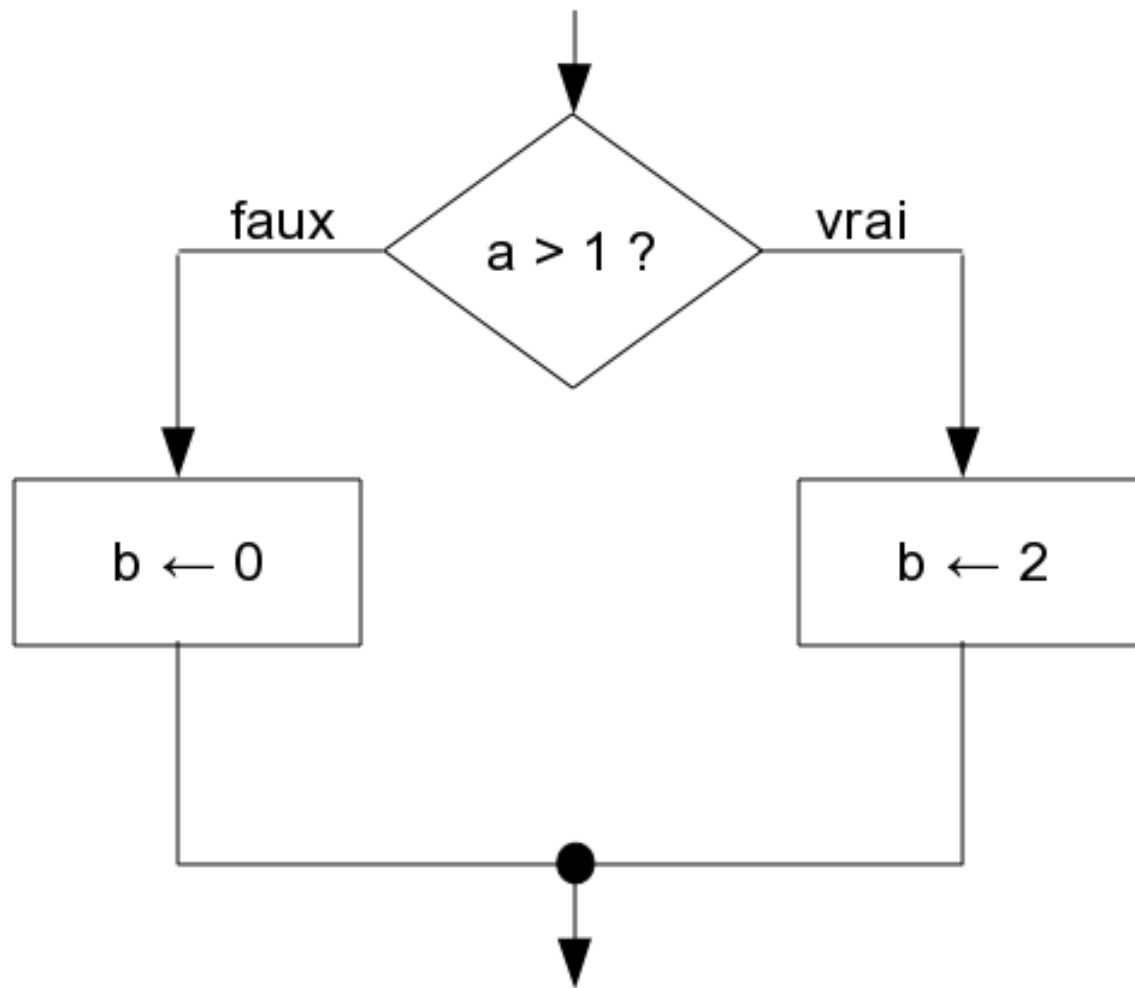
```
if (expression) instruction;
```



```
if (expression) instruction_1; /* SI ... ALORS ... */  
else instruction_2; /* SINON ... */
```
- expression doit être de type numérique ou de type pointeur.
- Cela donne le comportement suivant :
 - **si** l'expression est de type numérique et que sa valeur est différente de 0 (faux), **alors** l'instruction_1 est exécutée, **sinon** c'est l'instruction_2.
 - si l'expression est de type pointeur, cela revient à comparer le pointeur avec la valeur `null`. Une valeur non `null` du pointeur provoque l'exécution de l'instruction_1 et une valeur `null`, celle de l'instruction_2.



Instruction conditionnelle (2/2)



```
if(a > 1)
{
    b = 2 ;
}
else
{
    b = 0 ;
}
```

Instruction de sélection (1/2)

Une instruction `switch` sélectionne une des instructions la composant, en fonction de la valeur d'une expression, qui doit être de type entier. Elle a la forme suivante :

```
switch (expression)
{
    case constante_scalaire_1 : liste_d_instructions_1;
    case constante_scalaire_2 : liste_d_instructions_2;
    // etc ...
    default : liste_d_instructions;
}
```

Important : expression doit rendre un résultat de type entier. La valeur exprimée derrière un `case` est une constante et en aucun cas une instruction.

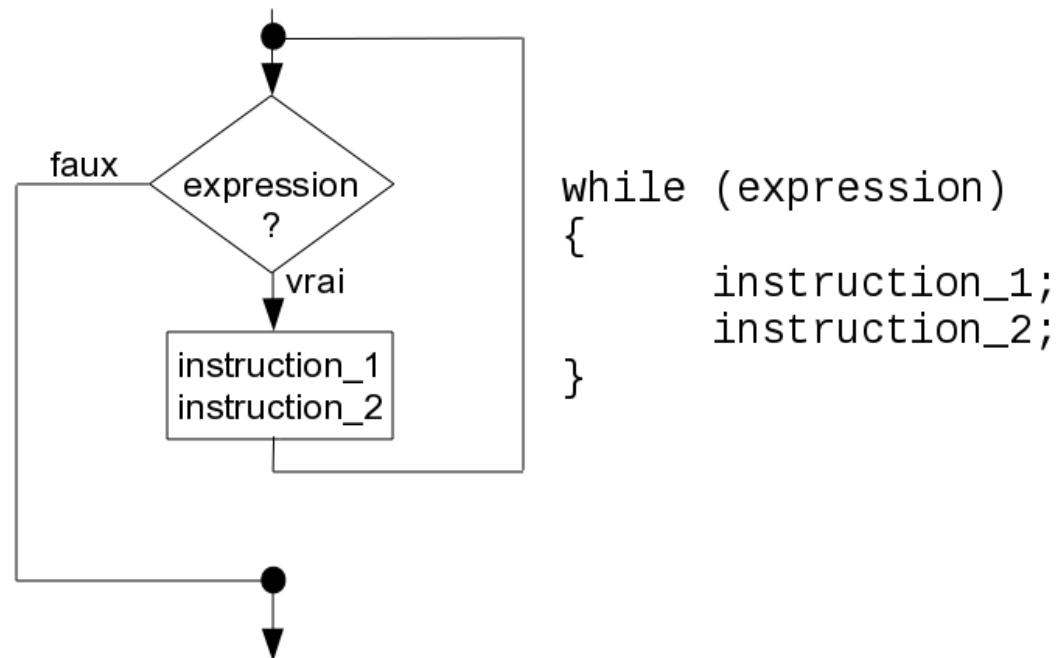


Instruction de sélection (2/2)

- Important : `expression` est comparée aux constantes derrière les `case`. A la première correspondance des deux valeurs, la liste d'instructions correspondante est exécutée.
- Si aucune constante n'est égale à `expression` :
 - s'il existe une partie `default`, elle est exécutée
 - sinon, l'instruction `switch` ne fait rien.
- Attention : Une fois terminée l'exécution d'une liste d'instructions d'un `case`, on continue en séquence dans le `case` suivant. Si on ne veut pas que ce genre de phénomène se produise, il faut utiliser l'instruction `break` (qui fait sortir du `switch`).

Instruction de contrôle d'itération : while

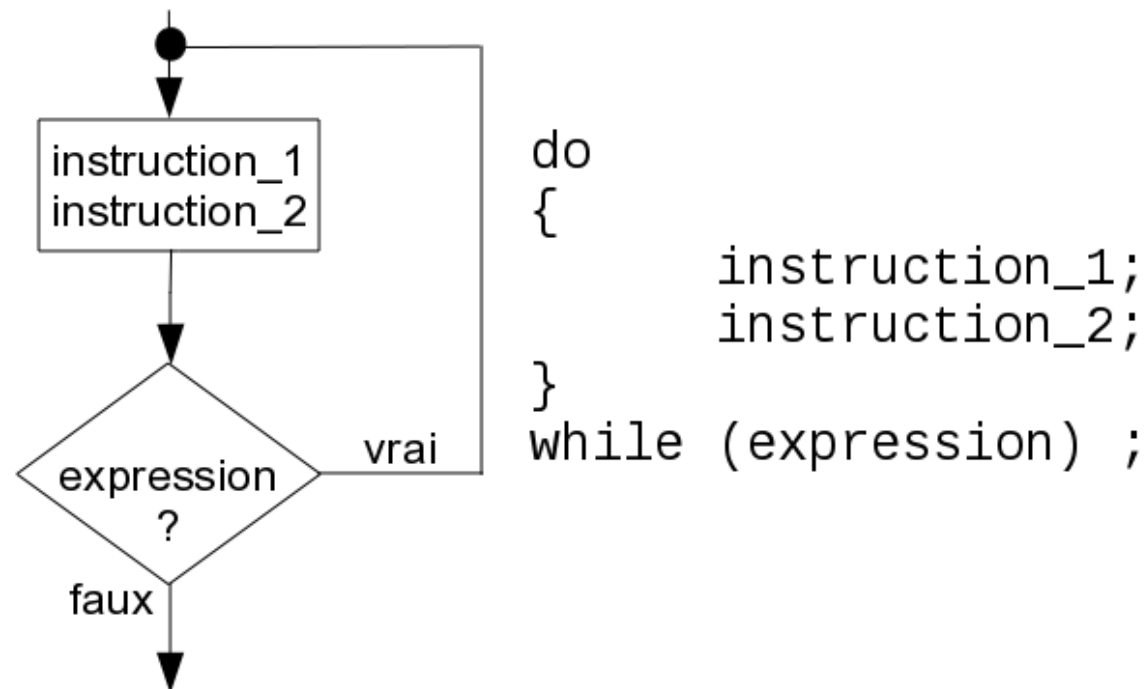
L'instruction `while` contrôle l'exécution répétitive d'une autre instruction TANT QUE `expression` est vraie. Elle a la forme suivante :



- `expression` doit être soit de type entier, soit de type pointeur.
- Au début de chaque itération, `expression` est évaluée. Si celle-ci a une valeur différente de zéro (ou de `null`), l'itération se poursuit, sinon la boucle est terminée. Cette boucle sera effectuée zéro ou plusieurs fois.

Instruction de contrôle d'itération : do

La suite d'instructions se trouvant entre les symboles `do` et `while` est exécutée de manière répétitive, TANT QUE `expression` est vraie (donc produise une valeur différente de zéro (ou de null)). Elle a la forme suivante :



- Comme précédemment, `expression` doit être soit de type entier, soit de type pointeur. Cette suite d'instructions sera exécutée au moins une fois.

Instruction de contrôle d'itération : for (1/2)

L'instruction `for` est une forme singulière et pratique de l'instruction `while`. Cette instruction n'a été rajoutée dans le langage que pour des raisons de clarté de programme. Elle a la forme suivante :

```
for (expression_1; expression_2; expression_3) instruction;  
//ou :  
for (expression_1; expression_2; expression_3)  
{  
    instruction_1;  
    instruction_2;  
    // etc ...  
}
```

- `expression_1` : correspond à la partie initialisation de l'itération
- `expression_2` : correspond à la condition d'arrêt (de type TANT QUE)
- `expression_3` : correspond au pas de l'incrément ou à une réinitialisation
- `instruction` : correspond au corps de l'itération

Instruction de contrôle d'itération : for (2/2)

L'instruction `for` est ainsi équivalente à l'instruction `while` suivante :

```
expression_1;  
while (expression_2)  
{  
    instruction;  
    expression_3;  
}
```

- *Remarque* : N'importe laquelle des 3 expressions peut donc être omise. Si `expression_2` est omise, on a alors affaire à une instruction `while(1)`, c'est à dire une boucle infinie. Exemple : `for (;;)` `instruction`; /* je suis une boucle infinie ! */
- C'est dans les parties initialisation et incrémentation que l'opérateur virgule `,` est le plus utile.



Instructions de rupture de séquence (1/2)

- Il en existe plusieurs :
 - Instruction `break` : Cette instruction fournit un mécanisme pour arrêter une boucle prématurément. Elle permet aussi de sortir d'un case de l'instruction `switch`, ce qui est nécessaire pour éviter d'exécuter en séquence le reste des instructions composant le `switch`. Elle cause ainsi l'arrêt de la première instruction `while`, `do`, `for` ou `switch` englobante.
 - Instruction `continue` : Cette instruction est très proche de l'instruction `break` mais ne s'applique pas à l'instruction `switch`. Elle ne s'applique en effet qu'aux instructions permettant le contrôle d'itération. Elle provoque l'arrêt de l'itération courante et le passage au début de l'itération suivante dans une boucle contrôlée par une instruction `while`, `do` ou `for`. Dans une boucle `for`, la partie réinitialisation de la boucle (`expression_3`) est exécutée.

Instructions de rupture de séquence (2/2)

- Il existe aussi :
 - Instruction `return` : Cette instruction provoque le retour chez l'appelant de la fonction courante. Les deux formes de l'instruction `return` sont : `return;` ou `return expression;`
 - Instruction `goto` (et étiquettes) : La forme d'une instruction `goto` est `goto etiquette;`. Elle a pour effet de provoquer le passage à l'exécution de l'instruction étiquetée par `etiquette`. Elle a pour limitation de n'autoriser le branchement qu'à l'intérieur du bloc régi par une fonction. Bien que cette instruction soit bannie de la programmation structurée, elle est parfois utilisée car elle permet un traitement plus facile de certaines situations.

Comment le C interprète le VRAI et le FAUX ?

```
#include <stdio.h>
int main() {
    int a = 0;

    if(a)
        printf("VRAI : a = %d -> a est strictement supérieur à 0\n", a);
    else printf("FAUX : a = %d -> a est égal à 0\n", a); // FAUX : a = 0 -> a est
        égal à 0

    a = 15;
    if(a)
        printf("VRAI : a = %d -> a est strictement supérieur à 0\n", a); // VRAI
        : a = 15 -> a est strictement supérieur à 0
    else printf("FAUX : a = %d -> a est égal à 0\n", a);

    if(1)
        printf("TOUJOURS VRAI !\n"); // TOUJOURS VRAI !
    else printf("JAMAIS FAUX !\n");

    return 0;
}
```

Utilisation de if avec des types pointeurs

```
#include <stdio.h>

int main()
{
    int a = 0;
    int *p = NULL;

    if(p)
        printf("VRAI : p = %p -> p n'est pas null\n", p);
    else printf("FAUX : p = %p -> p est null\n", p); // FAUX : p = (nil) -> p est
        null

    p = &a;
    if(p)
        printf("VRAI : p = %p -> p n'est pas null\n", p); // VRAI : p = 0
            xbfff3738 -> p n'est pas null
    else printf("FAUX : p = %p -> p est null\n", p);

    return 0;
}
```

Erreurs du débutant (1/3)

Le bug du débutant n°1 : il ne faut pas mettre un ; après un if car cela exécuterait une instruction nulle si expression est vraie. Cela donne un comportement défectueux :

```
int a = 0;
if(a != 0); /* c'est sûrement un bug involontaire */
    printf("bug : a n'est pas nul et pourtant a = %d !\n", a);
```

Le bug du débutant n°2 : il ne faut pas confondre l'opérateur '=' (d'affectation) avec l'opérateur '==' (comparaison d'égalité). Cela risque de donner un comportement défectueux :

```
int a = 0;
if(a = 0) /* c'est sûrement un bug involontaire */
    printf("a = %d : a est égal à 0\n", a);
else printf("bug : a n'est pas égal à 0 et pourtant a = %d !\n", a);
```



Erreurs du débutant (2/3)

Le bug du débutant n° 3 : il faut penser à déterminer si un `break` est nécessaire dans un `case`. Le code suivant le prouve :

```
a = 1;
switch (a)
{
    case 1: printf("a = %d : a est égal à 1\n", a); /* ne faudrait-il pas un break
        ? */
    case 2: printf("bug : a = %d : a est égal à 2 !\n", a); /* ne faudrait-il pas
        un break ? */
}
```

Vous obtiendrez :

```
a = 1 : a est égal à 1
bug : a = 1 : a est égal à 2 !
```



Erreurs du débutant (3/3)

Le bug du débutant n°4 : il ne faut pas mettre un ; après le while car cela exécuterait une instruction nulle si expression est vraie. Le code suivant sera dans un **boucle infinie** sans afficher aucun message "Salut" :

```
long compteur = 15;
while (compteur > 0); // c'est sûrement un bug involontaire
{
    printf("Salut\n");
    compteur = compteur - 1;
} // ici non plus ne pas mettre de ;
```


Boucle for : gérer 2 itérateurs

// Utilisation de l'opérateur ',' pour gérer 2 itérateurs

```
#include <stdio.h>
#include <string.h> /* pour strlen */
int main() {
    int i, j; /* deux itérateurs */
    char s[16] = "Hello wordl !"; /* une chaîne de caractères */
    char c; /* un simple caractère */

    printf("Une chaîne de caractères : %s\n", s); // Affiche : Hello wordl !

    for (i=0, j=strlen(s)-1; i<j; i++, j--)
    {
        c = s[j];
        s[j] = s[i];
        s[i] = c;
    }
    printf("Une chaîne de caractères inversée : %s\n", s); // Affiche : ! ldrow
        olleH
    return 0;
}
```

