Chapitre 9

Structures de contrôle (if, while, for, switch, ...)

Ce chapitre, le neuvième de ce tutorial, va nous permettre d'apprendre ce que sont les structures de contrôle, leur utilité, et, surtout, comment les utiliser.
Nous profiterons de ce chapitre pour parler des opérateurs de comparaison, qui sont à la base de la plupart des conditions.
En plus de cela, nous devrons, avant de pouvoir nous attaquer au sujet principal de ce chapitre, étudier deux autres nouveaux opérateurs, qui nous seront utiles pour combiner des conditions simples, pour on obtenir de plus complexes.

I:\ Quelques bases au sujet des structures de contrôle

Nous commencerons ce chapitre par une partie au cours de laquelle nous définirons grossièrement les structures de contrôle, puis au cours de laquelle nous étudierons les opérateurs de comparaison, pour finir par les opérateurs de logique booléenne.

A: Qu'est-ce que c'est, et pourquoi en utiliser ?

Un programme ayant un fonctionnement linéaire, un comportement déterminé précisément et invariant, n'est généralement pas très utile ; certes, cela permet d'effectuer un grand nombre de fois une tâche répétitive, simplement en lançant le programme le nombre de fois voulu... Mais, généralement, il faut que le programme s'adapte à des cas particuliers, à des conditions, qu'il exécute certaines fois une portion de code, d'autres fois une autre portion de code, qu'il soit capable de répéter plusieurs fois la même tâche sur des données successives...
Tout ceci nécessite ce qu'on appelle "structures de contrôle".

Le C propose plusieurs structures de contrôle différentes, qui nous permettent de nous adapter à tous les cas possibles. Il permet d'utiliser des conditions (structures alternatives), des boucles (structures répétitives), des branchements conditionnels ou non, ...
Certaines de ces structures de contrôle sont quasiment toujours utilisées ; d'autres le sont moins. Certaines sont très appréciées, d'autres sont à éviter...

Une fois la première partie de ce chapitre terminée, nous passerons à l'étude de ces différentes structures de contrôle.

B: Les opérateurs de comparaison

Lorsque l'on travaille avec des conditions, il s'agit généralement pour nous de faire des comparaisons.
Par exemple, "est-ce que a est plus petit que b ?" ou "est-ce que la touche appuyée au clavier est égale à ESC ?" ou encore "est-ce que l'expression _le_fichier_X_existe_ est égale à vrai ?".

En C, pour exprimer une comparaison, on utilise, tout comme en mathématiques, des opérateurs. Ces opérateurs sont quasiment tous des opérateurs binaires, ce qui signifie qu'ils prennent une donnée à gauche, une donnée à droite, et renvoie le résultat de la comparaison.
Le résultat de la comparaison est une valeur booléenne, notée TRUE (vrai) ou FALSE (faux). On associe à TRUE toute valeur non nulle, et à FALSE la valeur nulle, ce qui signifie que toute expression dont le résultat est différent de 0 sera considérée comme valant TRUE si on l'emploie dans une condition. Il en va de même, respectivement, pour FALSE et la valeur 0.

1: Egalité, Différence

Pour déterminer si deux expressions sont égales, on utilise l'opérateur == (deux fois le symbole égal à la suite, sans rien entre !).
Notez bien que l'opérateur = (égal seul) est utilisé pour les affectations, pas pour la comparaison ! C'est un fait que les débutants ont trop souvent tendance à oublier.

Pour déterminer si deux expressions sont différentes, on utilise l'opérateur != (un point d'exclamation, immédiatement suivi d'un symbole égal).

Par exemple,
`10 == 20` renverra FALSE,
`15 != 20` renverra TRUE,
`12 == 12` renverra TRUE,
et `14 != 14` renverra FALSE.
(Notez que ce ne sont que des exemples théoriques : en réalité, il est complètement absurde d'effectuer de telles comparaisons, et ces opérateurs permettent de faire plus que cela !)

2: Supérieur strict, Supérieur ou Égal

Il est aussi, naturellement, possible de comparer des données de façon à déterminer si l'une est plus grande que l'autre.
Pour une comparaison stricte, en excluant l'égalité, on utilise, comme en mathématiques, le symbole >.
Pour une comparaison non stricte, incluant le égal, on utilisera l'opérateur >= (constitué d'un symbole supérieur immédiatement suivi d'un symbole égal).

Par exemple,
`10 > 20` renverra FALSE,
`20 >= 20` renverra TRUE,
`12 > 11` renverra TRUE,
et `24 >= 34` renverra FALSE.

3: Inférieur strict, Inférieur ou Égal

Pour comparer des données de façon à déterminer si l'une est plus petite que l'autre, on agira exactement de la même manière, mais en utilisant les opérateurs < et <= selon que l'on veut une comparaison stricte ou non.

4: Négation

Il existe un dernier opérateur de comparaison, qui lui, est un opérateur unaire, ce qui signifie qu'il ne travaille qu'avec une seule donnée, celle que l'on place à sa droite. Il ne permet de faire une comparaison qu'avec 0.
Cela signifie que si son opérande, la donnée sur laquelle il travaille, vaut 0, la comparaison vaudra TRUE, et que si la donnée a une valeur non nulle, la comparaison vaudra FALSE.

Par exemple,
`!0` renverra TRUE,
`!10` renverra FALSE.

Notez que, comme nous l'avons dit plus haut, FALSE vaut 0, et que TRUE vaut une valeur non nulle... L'opérateur ! peut donc être utilisé pour obtenir la négation d'une condition...
Par exemple,
`!(10 > 20)` renverra TRUE,
`!(20 == 20)` renverra FALSE.

C: Les opérateurs logiques Booléens

Le C propose deux opérateurs binaires de logiques booléenne, qui permette notamment de regrouper plusieurs conditions, afin de former une condition plus complexe.
Ces opérateurs sont :

- OU inclusif, dont l'opérateur est `||` (deux barres verticales l'une à la suite de l'autre ; barres verticales accessibles par Alt-Gr + 6 sur un clavier azerty).
- ET, dont l'opérateur est `&&`

Ces deux opérateurs fonctionnent un peu comme les opérateurs bit à bit `|` et `&` que nous avons étudié au chapitre précédent, à cela près qu'ils ne travaillent pas avec les bits, mais avec des données dans leur intégralité.
Notez aussi qu'il n'existe pas d'opérateur OU exclusif de logique booléenne. Si vous en avez l'utilité, il vous faudra procéder "à la main", avec une succession de `&&` et de `||`.

Par exemple,
`TRUE || TRUE` renverra TRUE,
`TRUE || FALSE` renverra TRUE,
`FALSE || TRUE` renverra TRUE,
`FALSE || FALSE` renverra FALSE.
Ou bien :
`TRUE && TRUE` renverra TRUE,
`TRUE && FALSE` renverra FALSE,
`FALSE && TRUE` renverra FALSE,
`FALSE && FALSE` renverra FALSE.

Notez que les expressions de ce type sont évaluées de gauche à droite, et ne sont évaluées dans leur intégralité que si nécessaire.
Par exemple, dans l'expression `TRUE || FALSE`, le programme verra que le premier terme est à TRUE, et ne prendra donc pas la peine d'évaluer le second, puisqu'il suffit que l'un des deux soit à TRUE pour que toute l'expression soit à TRUE ; l'expression vaudra TRUE.
Respectivement, dans l'expression `FALSE && TRUE`, le programme verra que le premier terme est à FALSE, et n'évaluera pas le second, puisqu'il faut que les deux soient à TRUE pour que l'expression soit à TRUE ; celle-ci sera donc à FALSE.
Cela dit, ceci tient un peu du détail, et l'ordre d'évaluation n'est souvent pris en compte par les programmeurs que lorsqu'ils optimisent leur programme, par exemple, en plaçant l'expression dont le résultat est le plus intéressant en premier. Le résultat global est le même, que vous placiez les opérandes dans un ordre, ou dans l'autre !

D: Résumé des priorités d'opérateurs

Exactement comme nous l'avons fait en fin de chapitre précédent, nous allons, puisque nous venons d'étudier plusieurs nouveaux opérateurs, dresser un tableau résumant leurs priorités respectives. Tout comme pour le tableau du chapitre précédent, la ligne la plus haute du tableau regroupe les opérateurs les plus prioritaire, et les niveaux de priorité diminuent au fur et à mesure que l'on descend dans le tableau. Lorsque plusieurs opérateurs sont sur la même ligne, cela signifie qu'ils ont le même niveau de priorité.

Opérateurs, du plus prioritaire au moins prioritaire

```
()
! ~ ++ -- + - sizeof
* / %
+ -
<< >>
< <= > >=
== !=
&
^
|
&&
||
= += -= *= /= %= &= ^= |= <<= >>=
```

Notez que les opérateurs + et - unaires sont plus prioritaires que leurs formes binaires.

II:\ Structures conditionnelles

Maintenant que nous avons appris ce dont nous avions besoin pour écrire des conditions, nous allons l'appliquer. Nous commencerons par apprendre à utiliser des structures de contrôle conditionnelles, aussi appelées alternatives, du fait que leur forme généralisée permet de faire un choix entre plusieurs portions de code à exécuter.

A: if...

La forme la plus simple de structure conditionnelle est d'exécuter quelque chose dans le cas où une condition est vraie.
Pour cela, on utilisera la structure de contrôle if, dont la syntaxe est la suivante :

```
if ( condition )
    Instruction à exécuter si la condition est vraie.
```

Par exemple, pour afficher "a vaut 10" dans le cas où la variable a contient effectivement la valeur 10, on utilisera le code suivant :

```
if(a == 10)
    printf("a vaut 10");
```

Notez que l'indentation (le fait que l'instruction à exécuter si la condition est vraie soit décallée vers la droite) n'influe en rien le comportement du programme : on peut tout à fait ne pas indenter. Cela dit, indenter correctement permet de rendre le code source beaucoup plus lisible ; je vous conseille de toujours prendre le soin d'indenter votre source !
Sachant qu'un bloc peut prendre la place d'une instruction, on pourra exécuter plusieurs commande dans le cas où une condition est vraie, simplement en les regroupant à l'intérieur d'un bloc, c'est-à-dire en les plaçant entre une accolade ouvrante, et une accolade fermante. Notez que cette remarque est vraie dans tous les cas où l'on attend une instruction ; en particulier, ceci est vrai pour les structures de contrôle que nous verrons dans la suite de ce chapitre.
Voici un exemple en utilisant un bloc à la place d'une seule instruction :

```
if(a == 10)
{
    clrscr();
    printf("a vaut 10");
    ngetchx();
}
```

Dans ce cas, si la variable a contient la valeur 10, l'écran sera effacé, on affichera un message, et on attendra une pression sur une touche.

B: if... else...

Il est souvent nécessaire de pouvoir exécuter une série d'instructions dans le cas où une condition est vraie, et d'exécuter une autre série d'instructions dans le cas contraire... Bien évidemment, on pourrait utiliser une construction de cette forme, en utilisant l'opérateur de logique booléenne de négation :

```
if ( condition )
    Instruction à exécuter si la condition est vraie.
if ( !condition )
    Instruction à exécuter si la condition est fausse.
```

Cela dit, le langage C fourni une forme de structure de contrôle alternative qui rend les choses plus simples et plus lisibles ; elle a la syntaxe qui suit :

```
if ( condition )
    Instruction à exécuter si la condition est vraie.
else
    Instruction à exécuter si la condition est fausse.
```

Ainsi, on ne teste même qu'une seule fois la condition, alors qu'on l'évaluait deux fois si l'on utilise l'idée donnée juste au-dessus (idée qui est à bannir, j'insiste !).

Par exemple, pour afficher un message dans le cas où une variable contient la valeur 10, et un autre message dans tous les autres cas, on peut utiliser l'extrait de code-source qui suit :

```
if(a == 10)
    printf("a vaut 10");
else
    printf("a est différent de 10");
```

C: if... else if... else...

On peut aussi être amené à vouloir gérer plusieurs cas particuliers, et un cas général, correspondant au fait qu'aucun des autres cas n'a été vérifié. Pour cela, nous emploierons la structure de contrôle dont la syntaxe est la suivante, et qui est la forme la plus complète de if :

```
if ( condition1 )
    Instruction à exécuter si la condition1 est vraie.
else if ( condition2 )
    Instruction à exécuter si la condition2 est vraie.
else if ( condition3 )
    Instruction à exécuter si la condition3 est vraie.
...
...
else
    Instruction à exécuter si les condtions 1, 2, et 3 sont toutes les trois fausses.
```

Avec une telle structure de contrôle, on n'évalue une condition que si toutes les précédentes sont fausses, et on finit par le cas le plus général, qui est celui où toutes les conditions énoncées précédemment sont fausses.

Notez que le cas `else` est optionnel : il est parfaitement possible de définir une structure alternative de ce type sans mettre de cas par défaut.
Remarquez aussi que l'on peut positionner autant de `else if` que l'on veut. La forme que nous avons vu au point précédent de ce chapitre n'est rien de plus qu'un cas où l'on n'emploie pas de `else if` !

Finissons cette partie concernant les structures de contrôle conditionnelles par un dernier exemple, utilisant la dernière forme que nous venons de définir, la plus complète des trois :

```
if(a == 10)
    printf("a vaut 10");
if(a == 15)
    printf("a vaut 15");
if(a == 20)
    printf("a vaut 20");
else
    printf("a est différent de 10, de 15, et de 20");
```

Si vous avez compris ce qui précède, cet exemple n'a pas besoin d'explications ; dans le cas contraire, je ne peux que vous inciter à relire ce que nous avons étudié plus haut...

D: Opérateur (condition) ? (vrai) : (faux)

Le C fournit aux programmeurs un opérateur permettant d'obtenir une valeur si une condition est vraie, et une autre si la condition est fausse ; il s'agit de l'opérateur ?:, qui s'utilise comme ceci :

```
condition ? valeur_si_vrai : valeur_si_faux
```

Et tout cette expression prend la valeur valeur_si_vrai si la condition est vraie, ou la valeur valeur_si_faux si la condition est fausse.
Cela explique que le nom de cet opérateur soit "expression conditionnelle" : il se comporte véritablement comme une expression, lorsqu'il est utilisé dans un calcul.

Par exemple, pour affecter à une variable une valeur en fonction du résultat de l'évaluation d'une condition, on pourra utiliser ceci :

```
short a;
short result;

// on initialise a d'une façon ou d'une autre

result = (a==10 ? 100 : 200);
```

Notons que cette écriture n'est que la forme concise de celle-ci :

```
short a;
short result;

// on initialise a d'une façon ou d'une autre

if(a == 10)
    result = 100;
else
    result = 200;
```

Mais utiliser l'opérateur ?: permet d'utiliser une valeur en fonction d'une condition là où on ne pourrait pas mettre un bloc if...else.

Par exemple, l'opérateur ?: est extrêmement pratique dans ce genre de situation :

```
short a;
short result;

// on initialise a d'une façon ou d'une autre

result = 2*(a==10 ? 100 : 200);
```

Alors que si on avait voulu utiliser une construction if...else, il aurait fallu utiliser une variable temporaire, comme ceci :

```
short a;
short result;
short temp;

// on initialise a d'une façon ou d'une autre

if(a == 10)
    temp = 100;
else
    temp = 200;

result = 2*temp;
```

(Ou effectuer la multiplication dans le bloc correspondant au if, et dans celui correspondant au else, ce qui n'est pas toujours possible, et qui grossirait inutilement notre programme.)

L'opérateur ?: est, puisqu'il fonctionne avec trois opérandes, un opérateur "ternaire".
Cela dit, étant donné que c'est le seul opérateur de ce genre que fournit le C, on a généralement tendance à l'appeler "l'opérateur ternaire", plutôt que "expression conditionnelle".

Et, puisque nous venons de voir un nouvel opérateur, voici la liste des priorités d'opérateurs, mise à jour en le prenant en compte :

Opérateurs, du plus prioritaire au moins prioritaire

```
()
! ~ ++ -- + - sizeof
* / %
+ -
<< >>
< <= > >=
== !=
&
^
|
&&
||
?:
= += -= *= /= %= &= ^= |= <<= >>=
```

Notez que les opérateurs + et - unaires sont plus prioritaires que leurs formes binaires.

III:\ Structures itératives

Le C présente trois types de structures de contrôle itératives, c'est-à-dire, de structures de contrôle permettant de réaliser ce qu'on appelle des boucles ; autrement dit, d'exécuter plusieurs fois une portion de code, généralement jusqu'à ce qu'une condition soit fausse.
Le plus grand danger que présentent les itératives est que leur condition de sortie de boucle ne soit jamais fausse... dans un tel cas, on ne sort jamais de la boucle (à moins d'utiliser une opération d'altération de contrôle de boucle, que nous étudierons au cours de cette partie), et on réalise une "boucle infinie". La seule façon de sortir d'une boucle infinie est de force la mort du programme, si vous en avez la possibilité (par exemple, si un kernel permettant de tuer le programme courant est installé sur la calculatrice), ou de réinitialiser la calculatrice : le C ne propose pas une "astuce", contrairement au ti-basic et sa touche ON, permettant de quitter le programme de manière propre en cas de fonctionnement incorrect.
Nous verrons dans ce chapitre trois formes de répétitives ; pour chacune d'entre-elle, nous donnerons au minimum deux exemples, qui seront volontairement assez proche pour chaque structure, afin de montrer comment chacune permet de faire ce que font les autres... ou leurs différences !

A: while...

La première des itératives que nous étudierons au cours de ce chapitre est la boucle "`while`", appelée "tant que" en français, lorsque l'on fait de l'algorithmie.
Avec cette structure de contrôle, tant qu'une condition est vraie, les instructions lui correspondant sont exécutées.
Sa syntaxe est la suivante :

```
while ( condition )
    Instruction à effectuer tant que la condition est vraie.
```

Naturellement, ici aussi, de la même façon que pour les conditionnelles que nous avons étudié et que pour les itératives que nous verrons dans quelques instants, il est possible d'utiliser un bloc d'instructions entre accolades à la place d'une simple instruction.

Voici un premier exemple, qui va afficher la valeur de la variable a :

```
a = 0;
while(a <= 2)
{
    printf("a=%d\n", a);
    a++;    // Il ne faut pas oublier de changer la valeur de a !
            // (dans le cas contraire, a n'aurait jamais la valeur 12,
            //  et on serait dans un cas de boucle infinie !!!)
}
```

On aura à l'écran, dans l'ordre, 0, 1, et 2. Une fois la valeur 2 affichée, la variable a sera incrémentée, et vaudra 3 ; la condition de boucle deviendra alors fausse, et on n'exécutera plus le code correspondant à la boucle : le programme continuera son exécution après la structure de contrôle.

Ci-dessous, un second exemple ; essayez de le comprendre, et, surtout, de comprendre ce qu'il fera, avant de lire le commentaire qui le suit...

```
a = 0;
while(a > 10)
{
    printf("On est dans la boucle");
}
```

Ici, la variable a vaut 0, et ne peut donc pas être supérieure à 10. La condition de boucle est fausse. Etant donné que cette condition est, avec les boucles `while`, testée avant d'exécuter le code correspondant à la boucle, on n'exécutera jamais celui-ci, et n'affichera donc rien à l'écran.
Naturellement, cet exemple est ridicule, puisque l'on fixe la variable a juste avant la répétitive, mais il montre un des principes de ce type de boucle. (Notez qu'un compilateur idéal détecterait que la condition est toujours fausse, et pourrait supprimer toute la structure répétitive du programme, puisqu'elle ne sert finalement à rien ! GCC le fait peut-être même parfois, je ne saurai dire)

B: do... while

La seconde structure de boucle, que nous allons maintenant étudier, est "`do...while`", que l'on pourrait, en français, appeler "faire... tant que".
Ici encore, les instructions constituant la boucle sont exécutées tant que la condition de boucle est vraie. Cela dit, contrairement à `while`, avec `do...while`, la condition est évaluée à la fin de la boucle ; cela signifie que les instructions correspondant au corps de la structure de contrôle seront toujours exécutées au moins une fois, même si la condition est toujours fausse !

La syntaxe correspondant à cette structure répétitive est la suivante :

```
do
    Instruction à exécuter tant que la condition est vraie.
while ( condition );
```

Reprenons l'exemple que nous avions utilisé précédemment, qui affiche les différentes valeurs que prend une variable au fur et à mesure qu'on l'incrémente, et qui quitte une fois que la variable en question atteint une certaine valeur, mais en utilisant une itérative de la forme `do...while`, cette fois-ci :

```
a = 0;
do
{
    printf("a=%d\n", a);
    a++;    // Il ne faut pas oublier de changer la valeur de a !
            // (dans le cas contraire, a n'aurait jamais la valeur 12,
            //  et on serait dans un cas de boucle infinie !!!)
}while(a <= 2);
```

Le résultat sera exactement le même que celui que nous avions précédemment obtenu, lorsque nous utilisions un `while`.

Par contre, si nous essayons de faire la même pour notre second exemple, en écrivant un code tel celui-ci :

```
a = 0;
do
{
    printf("On est dans la boucle");
}while(a > 10);
```

... nous pourrons constater que l'on rentre dans la boucle, alors que la condition n'est pas vraie... Ce qui nous montre bien que la condition de boucle est testée en fin de boucle, et que les instructions correspondant à celle-ci sont toujours exécutées, au moins une fois.
Certes, dans un cas tel celui-ci, c'est plus un inconvénient qu'autre chose... Mais il est des cas lorsque l'on programme, où ce comportement correspond à ce que l'on recherche, et, dans ces occasions, il est plus simple d'utiliser un `do...while` qu'un `while` !

C: for

La troisième, et dernière, structure de contrôle itérative est celle que l'on appelle boucle "`for`". Elle est généralement utilisée lorsque l'on veut répéter un nombre de fois connu une action.
Sa syntaxe générale est la suivante :

```
for ( initialisation ; condition ; opération sur la variable de boucle )
    Instruction à exécuter tant que la condition est vraie.
```

Les trois expressions que j'ai nommé intialisation, condition, et opération sur la variable de boucle sont toutes trois optionnelle ; si aucune condition n'est précisée, le compilateur supposera que la condition est toujours vraie. Les points-virgule, eux, par contre, sont obligatoires !
Les noms que j'ai employé ne sont que purement indicatifs, mais ils correspondent à l'usage que l'on fait généralement de la boucle for, à savoir répéter un nombre de fois connu une suite d'instructions. Pour cela, il faut :

- Disposer d'une variable de boucle, qui sera considérée comme un compteur du nombre de fois dont on est passé dans la boucle.
- Initialiser cette variable, ce que l'on fait grâce à la première expression du for.
- Avoir une condition de boucle : une fois cette condition devenue fausse, on cessera de boucler. En règle générale, il est recommandé que cette condition porte sur la variable de compteur !
- Modifier la valeur de la variable de boucle utilisée comme compteur.

Par exemple, nous pourrions ré-écrire, une fois de plus, notre premier exemple, qui correspond tout à fait au cas d'utilisation le plus courant d'une boucle for, de la forme suivante :

```
for(a=0 ; a<=2 ; a++)
{
    printf("a=%d\n", a);
}
```

Le fonctionnement de ceci est tout simple : on initialise à 0 notre variable, on vérifie qu'elle est inférieure ou égale à deux, on affiche sa valeur, on l'incrémente, on vérifie qu'elle est inférieure ou égale à deux, on affiche sa valeur, on l'incrémente, ...
L'initialisation se fait une et une seule fois, au tout début, avant de rentrer dans la boucle ; la condition est évaluée en début de boucle, exactement comme pour `while`, et l'opération sur la variable de fin de boucle est effectuée en fin de boucle, avant de boucler.

Pour vérifier que le test de la condition se fait avant le passage dans la boucle, vous pouvez essayer avec l'exemple qui suit :

```
for(a=0 ; a>10 ; a--)
{
    printf("On est dans la boucle");
}
```

... et vous constaterez que l'on n'affiche pas de message à l'écran.

Voyons à présent trois exemples, réalisant exactement la même chose que le premier, mais en n'ayant pas mis certaines des trois expressions de l'instruction for ; ces instructions ont été sorties du for, placées soit avant celui-ci, soit dans la boucle, afin de montrer clairement où est-ce que le for les place dans le cas où on le laisse faire (ce que je ne peux que conseiller !)
Tout d'abord, en sortant l'initialisation :

```
a = 0;
for( ; a<=2 ; a++)
{
    printf("a=%d\n", a);
}
```

Ou en plaçant l'opération sur la variable de boucle en fin de boucle :

```
for(a=0 ; a<=12 ; )
{
    printf("a=%d\n", a);
    a++;
}
```

Ou encore en ne laissant que la condition :

```
a = 0;
for( ; a<=2 ; )
{
    printf("a=%d\n", a);
    a++;
}
```

Dans ce dernier cas, on revient exactement à une itérative de type `while`, qu'il vaut mieux utiliser, pour des raisons de facilité de compréhension...

Notez que la forme où on ne place ni initialisation, ni condition, ni opération, est souvent utilisée comme boucle infinie, dont il est possible de se sortir en utilisant certaines instructions d'altération de contrôle de boucle, que nous allons voir très bien ; pour illustrer ce propos, voici la syntaxe correspondant à une boucle infinie :

```
for ( ; ; )
    Instruction à exécuter sans cesse.
```

Naturellement, utiliser une structure de type "`while`" ou "`do...while`" avec une valeur non nulle à la place de la condition revient exactement au même ; mais c'est cette forme qui est censée être utilisée.

D: Instructions d'altération de contrôle de boucle

Le langage C propose plusieurs instructions qui permettent d'altérer le contrôle de boucles itératives, soit en forçant le programme à passer à l'itération suivante sans finir d'exécuter les instructions correspondant à celle qui est en cours, soit en forçant le programme à quitter la boucle, comme si la condition était fausse.
C'est dans cet ordre que nous étudierons ces instructions d'altération de répétitives.

1: continue

L'instruction `continue` permet de passer au cycle suivante d'une boucle, sans exécuter les instructions restantes de l'itération en cours.
Considérez l'exemple suivant :

```
for(a=0 ; a<=5 ; a++)
{
    if(a == 3)
        continue; // On passe directement à l'itération suivante,
                  // sans effectuer la fin de la boucle cette fois-ci.
                  // Donc, lorsque a vaudra 3, on n'affichera pas
                  // sa valeur.
    printf("a=%d\n", a);
}
```

Lorsque a sera différent de 3, l'appel à printf permettra d'afficher sa valeur. Mais, lorsque a vaudra 3, on exécutera l'instruction `while`. On retournera immédiatement au début de la boucle, en incrémentant a au passage.

Naturellement, pour un cas de ce genre, il serait préferable d'utiliser quelque chose de plus simple, et de plus explicite, dans le genre de ceci :

```
for(a=0 ; a<=5 ; a++)
{
    if(a != 3)
        printf("a=%d\n", a);
}
```

Notez que l'instruction `continue` rompt la logique de parcours de la boucle, et rend le programme plus difficilement compréhensible ; elle est donc à éviter autant que possible, et ne devrait être utilisée que lorsque c'est réellement la meilleure, voire la seule, solution possible !

2: break

L'instruction `break`, elle, met fin au parcours de la boucle, sitôt qu'elle est rencontrée, comme si la condition d'itération était devenue fausse, mais sans même finir de parcourir les instructions correspondant au cycle en cours.
Etudions l'exemple qui suit :

```
for(a=0 ; a<=5 ; a++)
{
    if(a == 3)
        break;  // Lorsque a vaut 3, on met fin à la répétitive,
                // sans même terminer la boucle courante.
                // On n'affichera donc, avec cet exemple,
                // que 0, 1, et 2 ; c'est tout.
    printf("a=%d\n", a);
}
```

Ce programme affichera la valeur de a lorsque a est inférieur à 3. Lorsque a vaudra 3, on exécutera l'instruction `break`... On quittera alors la boucle, sans même afficher la valeur 3, puisque l'appel à printf suit le `break`.

Ici encore, nous avons choisi un exemple extrêmement simple, qui pourrait être écrit de manière plus judicieuse (Si l'on ne veut pas aller jusque 3, pourquoi est-ce qu'on ne limiterait pas à 3 la valeur de a en condition d'itération ?), comme suit :

```
for(a=0 ; a<3 ; a++)
{
    printf("a=%d\n", a);
}
```

De la même façon que pour `continue`, `break` rompt la logique de parcours de la boucle... Cette instruction est donc elle aussi à éviter, dans la mesure du possible.

Un peu plus loin dans ce chapitre, nous parlerons de l'instruction `return`, qui peut aussi être utilisée de façon à altérer le contrôle de boucle, mais qui est d'un usage plus général.

IV:\ Structure conditionnelle particulière

Le langage C présente une structure conditionnelle particulière, le `switch`.

Cette structure est particulière dans le sens où elle ne permet que de comparer une variable à plusieurs valeurs, entières.

```
switch(nom_de_la_variable)
{
    case valeur_1:
            Instructions à exécuter dans le cas où la variable vaut valeur_1
        break;
    case valeur_2:
            Instructions à exécuter dans le cas où la variable vaut valeur_2
        break;
    default:
            Instructions à exécuter dans le cas où la variable vaut une valeur autre
que valeur_1 et valeur_2
        break;
}
```

Une structure `switch` peut avoir autant de case que vous le souhaitez. Le cas '`default`' est optionnel : si vous le mettez, les instructions lui correspondant seront exécutée si la variable ne vaut aucune des valeurs précisée dans les autres cas ; si vous ne le mettez pas et que la variable est différente des valeurs précisées dans les autres cas, rien ne se passera.
Je me permet d'insister sur le fait que `switch` ne permet de comparer une variable qu'à des valeurs ENTIERES ! Il est impossible d'utiliser cette structure conditionnelle pour comparer une variable à un flottant, par exemple !

Et voici un exemple d'utilisation de la structure conditionnelle `switch`, sans le cas `default` :

```
short a = 10;
switch(a)
{
    case 5:
            printf("a vaut 5\n");
        break;
    case 10:
            printf("a vaut 10\n");
        break;
    case 15:
            printf("a vaut 15\n");
        break;
}
```

Étant donné que la variable a vaut 10, et que l'on a un cas qui correspond à cette valeur, on affichera un message disant "a vaut 10".

A présent, si la variable ne correspond à aucune des valeurs proposées, toujours sans cas default :

```
short b = 20;
switch(b)
{
    case 5:
            printf("b vaut 5\n");
        break;
    case 10:
            printf("b vaut 10\n");
        break;
}
```

Ici, b vaut 20... mais on n'a aucun cas correspondant à cette valeur... On n'affichera donc rien à l'écran.

La même chose, avec un cas `default` :

```
short b = 20;
switch(b)
{
    case 5:
            printf("b vaut 5\n");
        break;
    case 10:
            printf("b vaut 10\n");
        break;
    default:
            printf("b ne vaut ni 5 ni 10\n");
        break;
}
```

Ici encore, aucun cas ne correspond de manière précise à la valeur 20... Puisque l'on a un cas default, c'est donc dans celui-ci qu'on se trouve, et on affichera un message disant que "b ne vaut ni 5 ni 10".

Notez que l'instruction `break`, que nous avons déjà étudié un petit peu plus haut dans ce chapitre, à la fin de chaque cas est optionnelle ; elle permet d'éviter que les cas suivants celui correspondant à la valeur de la variable soient exécutés, puisque le `break` permet de quitter une structure de contrôle. Si on ne le met pas, il se passera quelque chose dans le genre de ce que nous propose cet exemple :

```
short c = 5;
switch(c)
{
    case 5:
            printf("c vaut 5\n");
    // Volontairement, on omet ici le break !
    case 10:
            printf("c vaut 10\n");
        break;
    case 15:
            printf("c vaut 15\n");
        break;
}
```

Qu'est-ce qui se passe ici ?
La variable c vaut 5. On entrera donc dans le cas correspondant, et on affichera le message "c vaut 5". Cela dit, puisqu'on n'a pas d'instruction `break` à la fin de ce cas, on ne quittera pas la structure de contrôle ; on continuera donc à exécuter les instructions... du cas 10 ! Et on affichera aussi le message "c vaut 10" ! Une fois ceci fait, on parviendra à une instruction `break`, et on quittera la structure `switch`.
Il arrive parfois que l'on ne mette pas une instruction `break`... Parfois, on le fait volontairement, et cela correspond à ce que nous voulons faire... Mais, souvent, en particulier pour les débutants, c'est un oubli qui entraîne de drôles de résultats à l'exécution du programme ! Soyez prudents.

V:\ Branchement inconditionnel

Pour finir ce chapitre, nous allons rapidement parler des opérateurs de branchement inconditionnels.
Un opérateur de branchement inconditionnel permet de "sauter" vers un autre endroit dans le programme, sans qu'il n'y ait de condition imposée par l'opérateur, au contraire des boucles ou des opérations conditionnelles, par exemple.

Les opérateurs `continue` et `break`, dont nous avons parlé plus haut, ont tout à fait leur place ici, même si nous avons choisi de les présenter dans le contexte où ils sont utilisés.

A: L'opérateur goto

Lorsque l'on parle d'opérateurs de branchement inconditionnels, le premier qui vienne généralement à l'esprit des programmeurs est le `goto`. Il permet de brancher sur ce qu'on appelle une "étiquette" (un "label", en anglais), déclaré comme suit :

```
nom_de_l_etiquette:
```

C'est à dire un identifiant, le nom de l'étiquette, qui doit être conforme aux normes concernant les noms de variables, suivi d'un caractère deux-points.

Et l'instruction `goto` s'utilise de la manière suivante :

```
goto nom_de_l_etiquette_sur_laquelle_on_souhaite_brancher;
```

Notez cependant que `goto` ne peut brancher que sur une étiquette placée dans la même fonction que lui. (Nous verrons au chapitre suivant ce que sont précisément les fonctions, notion que nous avons déjà eu l'occasion d'évoquer).

Voici un petit exemple simple utilisant un `goto` :

```
    printf("blabla 1\n");
    goto plus_loin;
    printf("blabla 2\n"); // Cette instruction ne sera
                          // jamais exécutée...
plus_loin:
    printf("blabla 3\n");
```

Comme vous pourrez le constater si vous essayez d'exécuter ce programme, le message "blabla 2" ne sera pas affiché... En effet, l'instruction `goto` sautera l'instruction lui correspondant...

Notez que j'ai l'habitude d'indenter les étiquettes d'un cran de moins que le reste du programme, afin de pouvoir les repérer plus facilement... Cette habitude me vient fort probablement de la programmation en langage d'Assembleur, et vous n'êtes nullement tenu de la respecter : comme je l'ai sûrement déjà dit, l'indentation ne sert à rien du point de vue du compilateur... Elle permet juste de rendre vos programmes plus lisibles ; à vous de déterminer quelles sont les habitudes d'indentation qui vous correspondent.

Pour finir, je me permet d'ajouter que l'utilisation massive de l'opérateur `goto` n'est pas vraiment une programmation propre. Je vous conseille ne l'utiliser que le plus rarement possible. En particulier, pensez à utiliser tout ce que nous avons déjà vu au cours de ce chapitre avant de vouloir utiliser `goto` !
En théorie, il est toujours possible de se passer de l'opérateur `goto`... même si, je le reconnais, dans certains cas (pour se sortir d'une triple boucle imbriquée ou d'une situation joyeuse dans ce genre, par exemple), il est bien pratique !

B: L'opérateur return

Pour finir ce chapitre, juste deux petits mots sur l'opérateur de branchement inconditionnel return.

Cet opérateur permet de quitter la fonction dans laquelle on l'appelle. Si la fonction courante est la fonction _main, alors, return quittera le programme.

Nous verrons probablement au chapitre suivant, traitant des fonctions, une utilisation plus générale de l'opérateur return, mais, en attendant, voici une façon de l'utiliser :

```
return;
```

Utilisé juste comme ceci, cet opérateur n'a pas une grande utilité... Mais nous verrons bientôt que ses possibilités sont supérieures à ce que nous présentons ici !