Les boucles en C

Jean-Gabriel Luque

October 14, 2024

1 Que sont les instructions de boucles et qu'est-ce qu'elles permettent?

Considérons un programme qui propose à un utilisateur de jouer à une partie de Taquin et qui à la fin de chaque partie lui demande s'il veut recommencer. Dans un tel programme, il y a une ligne de la forme

```
....
printf("Voulez-vous rejouer o/n? :");
....
```

qui peut être executée un nombre indéterminé de fois.
Si on n'utilise que les instructions que l'on a déjà présentées dans le cours (affectations, gestion des entrées/sorties, tests), il n'est pas possible de programmer un tel comportement[1]. Si on décidait de borner le nombre de parties, on pourrait, du moins en principe, écrire ce genre de programme... au prix d'un grand nombre de répétitions le rendant très lourd. Par exemple :

```
...
score += taquin();
printf("Voulez-vous rejouer o/n? :");
assert(scanf("%c", &rep) == 1);
if (rep == 'o') {
   score += taquin();
   printf("Voulez-vous rejouer o/n? :");
   assert(scanf("%c", &rep) == 1);
   if (rep == 'o') {
     score += taquin();
     printf("Voulez-vous rejouer o/n? :");
     assert(scanf("%c", &rep) == 1);
     if (rep == 'o') {
       score += taquin();
       printf("Voulez-vous rejouer o/n? :");
       assert(scanf("%c", &rep) == 1);
```

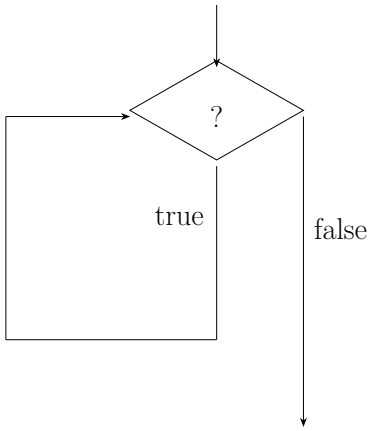
[1] En fait, ce n'est pas tout à fait vrai. En faisant en sorte que certaines fonctions s'appellent elles-mêmes, on peut avoir écrire des programmes dont la trace peut être arbitrairement longue en fonction des interactions avec l'utilisateur. Nous nous interdirons d'utiliser ce procédé pour l'instant car il nécessite de bien connaître la notion de récursivité, notion qui ne sera développée qu'en seconde année.

```
      ...
      }
    }
  }
 printf("Votre score  : %d", score);
 return EXIT_SUCCESS;
```

Ce n'est vraiment pas un code propre ni lisible. Notez surtout qu'il est impossible de ne pas imposer une borne sur le nombre de parties car cela nécessitearait un code de longueur infinie.
Les boucles sont des outils qui vont nous permettre :

1. d'alléger le code pour le rendre plus lisible,
2. d'avoir des traces d'exécution arbitrairement longues.

Comme pour les instructions de test, il existe une représentation graphique qui permet de visualiser la structure et l'imbrication des boucles :



Ce type de schemas nous permettra de comprendre le comportement de chaque type de boucles en `C`. Néanmoins, pour modéliser la structure et le comportement de boucles dans un programme nous devrons introduire quelques notions de logique qui seront développées plus longuement au second semestre.

2 Les instructions d'itérations en `C`

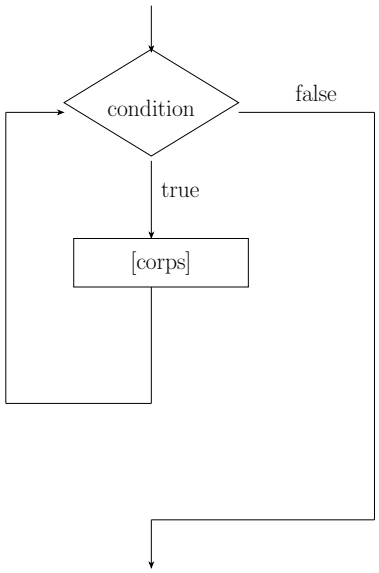
2.1 Les boucles `while`

Le mot clef `while` permet d'exécuter un bloc d'instructions tant qu'une condition (de continuité) est vraie. Il s'utilise de la façon suivante

```
while (Condition de continuité) {
  [Séquences d'instructions]
}
```

Lors de l'exécution, le programme commence par évaluer la `condition de continuité`, qui est une expression booléenne. Si celle-ci produit la valeur `true` alors la séquence d'instructions du

bloc est executée puis le programme se repositionne sur le test de début de boucle. Si l'évaluation de la `condition de continuité` produit `false` alors le programme passe à l'instruction suivante. Graphiquement, cela donne le schema :



Remarque 1. Le bloc d'instructions de la condition peut contenir n'importe quel type d'instructions, y compris des tests et des boucles.

Exemple 1. La suite de Collatz (aussi appelée suite de Syracuse) est une suite récurrente définie par

$$u_n = \begin{cases} \frac{u_n}{2} & \text{si } u_n \text{ est pair} \\ 3u_n + 1 & \text{si } u_n \text{ est impair} \end{cases}$$

L'initialisation de la suite est une valeur entière $u_0 = N$.
Il est conjecturé[2] que, quelque soit la valeur initiale $N > 1$, la suite fini par atteindre la valeur 1.
Le code suivant permet de vérifier la conjecture sur des exemples en calculant la durée du vol, c'est à dire le nombre d'itérations nécessaires pour atteindre 1.
Le nombre d'itérations d'une boucle est le nombre d'exécutions consécutives du son corps.

```
...
1   int n;
2   assert(scanf("%d",&n) == 1);
3   int vol = 0;
4   while (n>1) {
5     if (n%2 == 0) {
6       n = n / 2;
7     } else {
8       n = 3 * n + 1
9     }
10    ++vol;
11   }
12  printf("La durée du vol est %d :",n);
....
```

[2]Cela a été constaté dans tous les exemples qui ont été testés, mais la propriété n'a jamais été prouvée.

C'est typiquement le genre de programme qui nécessite l'utilisation d'une boucle : même en abusant des répétitions, il n'est pas possible d'écrire un programme qui ait ce comportement car il faudrait prévoir à l'avance le nombre d'itérations qui seront nécessaires. En effet, même en supposant la conjecture est vraie, la trace du programme est arbitrairement longue.

Supposons que l'utilisateur saisisse la valeur `3`, voici quelle serait la trace du programme :

`n`	`vol`	`numéro de ligne`	`condition (n>1)`
3	0	4	true
...			
10	1	4	true
...			
5	2	4	true
...			
16	3	4	true
...			
8	4	4	true
...			
4	5	4	true
...			
2	6	4	true
...			
1	7	4	false
...			

Le nombre d'itérations est égal à la valeur du `vol`.

Remarque 2. Attention, le mot clef `while` est suivi d'un bloc. Aucune variable déclarée à l'interieur de ce bloc ne sera accessible une fois la boucle terminée.

2.2 Les boucles `for`

Le mot clé `for` est, de préférence, utilisé lorsqu'une variable parcourt un intervalle entier. Il s'utilise de la façon suivante :

```
for([initialisation];[condition];[iteration]) {
  [corps de la boucle]
}
```

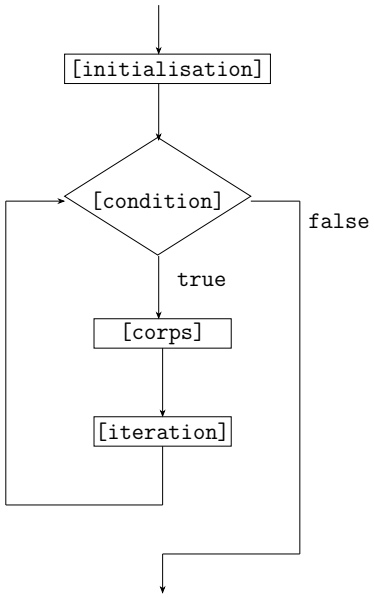
Les `[...]` indiquent un caractère optionnel.

- `[initialisation]` : est une séquence d'instructions[3]; chaque instruction est séparée de la suivante par une virgule. Ces instructions sont exécutées avant le début de la boucle. Si cette rubrique est vide, aucune instruction n'est exécutée.

[3]Pour être plus précis, la rubrique `+[initialisation]+` n'est pas réellement une séquence d'instructions. Il s'agit d'une expression qui est évaluée par le compilateur. La valeur produite par cette expression est celle du dernier élement de la séquence. Cette expression peut avoir n'importe quel type et, comme toute expression en `C` celle-ci peut faire appel à des commandes qui ont une action sur la mémoire. Par exemple, dans `for(k=0;...;...)`, `k=0` affecte la valeur `0` à la variable `k` mais il s'agit aussi d'une expression qui renvoie la valeur `0`. Testez aussi `int a; int p=(a=10,++a); printf("%d",p);`.

- `[condition]` : est une expression booléenne qui est évaluée avant chaque tour de boucle. Si celle-ci produit la valeur `true` alors les instructions du corps de boucle et de la rubrique itération sont exécutées. Si cette partie est vide, la condition est considérée comme ayant produit `true`.
- `[corps de la boucle]` : il s'agit du bloc d'instructions qui sera exécuté à chaque tour de boucle.
- `[itération]` : est une séquence d'instructions[4] qui sera executée à la fin de chaque tour de boucle (après l'exécution des instructoins du `[corps]`).

Graphiquement, le schémas d'exécution se représente par le dessin :



Remarque 3. Faites attention aux points suivants :

1. Les instructions des rubriques `[initialisation]` et `[iteration]` font partie d'un bloc associé à la fonction. Cela signifie que si des variables sont déclarées dans ces rubriques alors elles ne seront pas utilisables à l'extérieur de la boucle. Par exemple :

   ```
   for (size_t i=0; ...; ...) {
     ....
   }
   ++i;
   ```

   produira une erreur de compilation sauf si une autre variable `i` a été déclarée avant la boucle.

2. Bien que cela soit autorisé, il n'est pas conseillé d'écrire plusieurs instructions dans les rubriques `[initialisation]` et `[iteration]`. Néanmoins, si vous le faites il faut veillez

[4] Même remarque que pour `[initialisation]`.

à écrire des instructions qui soit indépendantes les unes des autres (le comportement du programme ne doit pas dépendre de l'ordre des instructions dans une rubrique). Par exemple, évitez d'écrire `++i,j+=i` dans la rubrique `[itération]`.

3. Les instructions de la rubrique `[itération]` sont exécutées même si la première évaluation de la `condition` produit `true`.

4. Faites très attention : `[itérations]` est exécuté APRÈS `[corps de la boucle]`!!!

Exemple 2. Considérons le code suivant :

```
1   int s = 0;
2   int k;
3   printf("0");
4   for(k = 1;k <= n;++k) {
5   printf(" + %d",k);
6   s += k;
7   }
8   printf(" == %d\n",s);
9   if ( s == (n * (n + 1)) / 2) {
10  printf("La formule de mon cours de math est correcte!\n");
11  } else {
12    printf("On m'aurait menti?");
13  }
14  printf("Attention k == %d!"),k);
```

Pour l'entier `3` saisi par l'utilisateur, le programme va exécuter les lignes dans cet ordre

```
1 -> 2 -> 3 -> 4 Ici k==1 et s==0
-> 5 -> 6 -> 7 -> 4 Ici k==2 et s==1
-> 5 -> 6 -> 7 -> 4 Ici k==3 et s==3
-> 5 -> 6 -> 7 -> 4 Ici k==4 et s==6 (Fin de la boucle)
-> 8 -> 9 -> 10 -> 14 -> ...
```

et afficher

```
0 + 1 + 2 + 3 == 6
La formule de mon cours de math est correcte!
Attention k == 4!
```

Faites bien attention à la valeur de `k` après la boucle.

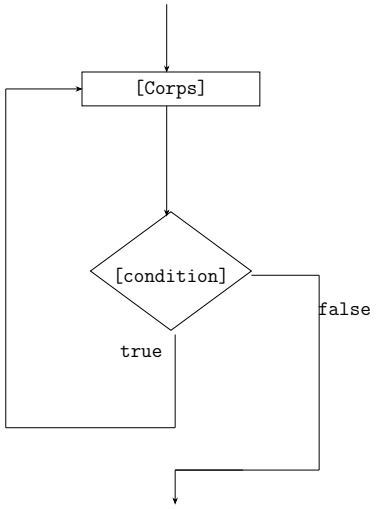
2.3 Les boucles `do...while`

Ces boucles se rapprochent des boucles `while` à ceci près que le corps de la boucle est exécuté AVANT le test. La syntaxe est la suivante

```
do {
  [corps de la boucle]
} while (Condition de continuité)
```

- `[corps de la boucle]` : un bloc d'instructions. Il sera exécuté à chaque tour de boucle et au moins une fois.
- `[Condition de continuité]` : une expression booléenne. Si celle-ci produit la valeur `true`, l'exécution recommence à la ligne du `do` sinon la boucle se termine et le programme reprend les instructions après celle-ci.

Graphiquement, le comportement peut se représenter par



Remarque 4.
- Cette boucle est à utiliser afin de garantir que le code du corps de la boucle soit exécuté au moins une fois.
- Attention, les variables utilisées dans `[condition]` doivent avoir été déclarées AVANT la ligne contenant le `do`. En effet, l'évaluation de cette expression ne prend pas en compte les variables déclarées dans le corps de la boucle.

Exemple 3. Considérons le code suivant

```
1  int n = 0;
2  double s = 0;
3  char c;

4  do {
5    double x;
6    printf("une valeur, svp :");
7    assert (scanf("%lf",&x) == 1);

8    s += x;
9    ++n;

10   printf("Voulez-vous ajouter une valeur? [o/n]");
11   assert(scanf(" %c",&c) == 1);
12 } while (c == 'o');

13  printf("La moyenne des valeurs est %lf", s / n);
```

Il permet de calculer et d'afficher la moyenne de réels donnés par l'utilisateur. Le calcul de la moyenne n'étant valable que si l'on a au moins une valeur, nous utilisons une boucle `do` pour garantir que l'utilisateur saisisse au moins un nombre. La boucle s'arrête lorsque l'utilisateur répond autre chose que `o` à la question `Voulez-vous ajouter une valeur?`. Notez que la variable `c` est bien déclarée avant la boucle (ligne 3). Si vous déplacez la déclaration à l'intérieur de la boucle, vous déclenchez une erreur à la compilation au niveau de l'évaluation de la condition de continuité (ligne 12).
Question auxiliaire : pourquoi il y a-t-il un espace devant `%c` à la ligne 11? Est-ce nécessaire?

2.4 Équivalence des instructions d'itération

D'un point de vue compilation, les trois type de boucles que l'on vient de voir sont équivalents. Ceci signifie que tout ce que l'on peut faire avec l'une peut-être réalisé avec les autres.

2.4.1 `while`

La boucle `while` doit être utilisée (de préférence) lorsqu'aucune variable ne parcourt un intervalle et qu'il n'est pas nécessaire d'exécuter le corps de la boucle au moins une fois.
Voici comment elle peut être traduite en utilisant les autres types de boucle :

```
     while        |       for         |      do
[I1]              | [I1]              | [I1]
while (C) {       | for ( ; C ; ) {   | do {
  [I2]            |   [I2]            |   if (C) {
}                 | }                 |     [I2]
[I3]              | [I3]              |   }
                  |                   | } while (C);
                  |                   | [I3]
```

Notez que l'écriture sous la forme `do` évalue l'expressions `C` un plus grand nombre de fois.

2.4.2 `for`

La boucle `for` doit être utilisée (de préférence) lorsqu'une variable parcourt un intervalle d'entiers (on dit que cette variable est un compteur de boucle). Voici comment elle peut être traduite en utilisant les autres types de boucle :

```
      for              |     while     |       do
[I1]                   |[I1]           |[I1]
for ([E1];[C];[E2]) {  |{              |{
  [I2]                 | [E1];         | [E1];
}                      | while ([C]) { | do {
[I3]                   |   [I2]        |   if ([C]) {
                       |   [E2];       |     [I2]
                       |  }            |     |E2];
                       |}              |   }
                       |[I3]           |  } while ([C]);
                       |               |}
                       |               |[I3]
```

Notez le bloc qui commence avant `[E1];` et qui termine avant `[I3]` dans les boucles `while` et `do`. En effet, les variables déclarées dans `[E1]` ne sont pas valides en dehors de la boucle.
Attention si la rubrique `[C]` est vide pour la boucle `for`, il faut écrire `true` dcomme condition de continuité pour les boucles `while` et `do`.

2.5 `do`

On a l'équivalence suivante :

```
      do        |     while     |     for
[I1]            |[I1]           |[I1]
                |{              |{
do {            | [I2]          | [I2]
  [I2]          | while (C) {   | for ( ; C ; ) {
} while (C);    |   [I2!]       |   [I2!]
[I3]            | }             | }
                |}              |}
                |[I3]           |[I3]
```

Notez la répétition du code `[I2]` pour les boucles `while` et `for` ainsi que les blocs.
La notation `\verb`[I2!]+ signigie que l'on a enlevée les déclarations, si il y en avait dans les instructions de @I2@. Par exemple, on a écrit @k = 0@ à la place de @int k=0@.

3 Exemples, trucs et astuces

3.1 Choix du type de boucle (on insiste!)

Le choix de la boucle à utiliser doit être guidé par des critères de lisibilité de code : éviter les répétitions, avoir un code dont la lecture est compréhensible etc.
Par exemple, on ne choisira d'utiliser une boucle `for` que si il y a une variable qui parourt un intervalle.
Lisez et traduisez votre code dans votre tête :

```
for(k = 0; k < n; ++k) {
  print("*");
}
=> Pour k partant de 0 et tant que k est inférieur à n on affiche une étoile.
```

En revanche, on évitera d'écrire ce genre de code :

```
for (int x=0; scanf("%d",&x) == 1; ++nb) {
...
}
```

En effet, bien que ce soit syntaxiquement correct, la signification que l'on voudrait donner à la lecture de cette ligne (parcours d'un intervalle) est en contradiction avec son comportement. Ici il vaudrait mieux écrire :

```
{
  int x=0;
  while (scanf("%d", &x) == 1) {
    ...
    ++nb;
  }
}
```

afin de signifier que l'on exécute le corps de la boucle tant que l'on a réussi à lire une valeur entière sur l'entrée standard.

3.2 Attention aux boucles (involontairement) infinies...

Certaines erreurs de programmation n'interrompent pas la compilation mais produisent des résultats indésirables lors de l'exécution. Il est très utile de savoir les détecter. Un cas classique est celui de la boucle (involontairement) infinie. Considérons cet exemple :

```
int n;
assert(scanf("%d",&n) == 1);

int i=1, fact=1;

while (i<n) {
  fact = fact * i;
}
```

Si la valeur saisie par l'utilisateur est strictement supérieure à `1` alors la boucle `while` ne s'arrêtera jamais. En effet, les variables utilisées dans la condition de continuité ne sont pas modifiées par le corps de la boucle. Si cette condition est évaluée à `true` au début de la boucle, alors sa valeur ne variera pas et la boucle ne s'arrêtera pas.
C'est une erreur classique qui arrive lorsque l'on convertit trop rapidement une boucle `for` en boucle `while`. Dans cet exemple, il manque la ligne `++i;` à la fin de la boucle.
Mais il peut se produire des erreurs plus subtiles. Examinons par exemple le code suivant :

```
int n;
assert(scanf("%d",&n) == 1);
assert( n > 0 );
int  lg=1;

while (n >= 0) {
  ++ lg;
  n /= 2;
}
```

La variable `n` qui apparaît dans la condition de boucle est bien modifiée dans le corps. Néanmoins, elle ne pourra jamais atteindre une valeur négative et, donc, l'évaluation donnera toujours `true`. Dans l'exemple, il aurait fallut écrire `n > 0` à la place de la condition de continuité `n >= 0` afin de garantir un arrêt dans tous les cas.

Il arrive (malheureusement) que seules certaines valeurs provoquent un comportement de boucle infini. Dans cet exemple :

```
int n, base;
assert(scanf("%d%d",&n,&base) == 1);
assert( n > 0 && base > 0);
int  lg=1;

while (n >= 0) {
  ++ lg;
  n /= base;
}
```

Ce code fonctionne dans la majorité des cas sauf si `base == 1`. Cette valeur implique que la ligne `n /= base;` ne modifie pas la valeur de `n`, l'évaluation de `n>= 0` donne toujours `true`. Pour corriger ce code, il suffit de prendre en compte ce cas particuler, par exemple en l'excluant `assert( n > 0 && base > 1);`.
En conclusion, il est toujours un peu délicat d'éviter ce genre d'erreurs. Pour ce faire :

- Vérifiez bien qu'au moins une variable apparaissant dans la condition de continuité soit modifiée dans le corps de la boucle.
- Vérifiez que la condition de continuité ne soit pas toujours égale à `true`.
- Vérifiez qu'il n'y ait pas de cas particulier provoquant une boucle infinie.

3.3 ... et aux boucles qui ne servent à rien...

Ce genre d'erreur a moins de conséquence que la précédente. Cependant, il peut être plus difficile à détecter et est aussi révelateur que quelque chose n'a pas été bien compris ou traduit lors du passage à la programmation. Par exemple :

```
int n, base;
assert(scanf("%d%d",&n,&base) == 1);
assert( n > 0 && base > 1);
int  lg=1;

while (n <= 0) {
  ++ lg;
  n /= base;
}
```

Ici le corps de la boucle ne sera jamais exécuté. Si c'est ce qui est voulu alors, cette boucle ne sert à rien et il faut l'effacer. Néanmoins, en général, ce genre de comportement provient d'une erreur, parfois une simple erreur de frappe. Ici, la condition `(n <= 0)` est erronée..., il aurait fallut écrire `(n >= 0)`.

3.4 Les boucles (volontairement) infinies

Dans certaines situations, il est parfois souhaitable d'écrire volontairement des boucles infinies. Par exemple, imaginons que l'on veuille afficher une horloge, on peut vouloir que celle-ci fonctionne jusqu'à ce que l'on éteigne la machine. Dans ce cas, il faut faire apparaître explicitement le fait que l'on veuille une boucle infinie, en écrivant `while(true)`.
On peut aussi vouloir, pour des raisons pratiques, sortir d'une boucle sans utiliser la condition de continuité. Par exemple, pour gérer une erreur de l'utilisateur (cela peut être pratique en particulier dans le cas d'une boucle infinie) ou tout simplement pour des raisons de lisibilités pour ne pas surcharger la condition. Dans ce cas, on utilise le mot clef `break` ou tout simplement un `return` si on se trouve dans une fonction. À titre d'exemple, vous pouvez considérer la définition de fonction suivante :

```
char getupper(void) {
 while(true) {
   char c;
   assert(scanf("%c",&c) == 1);
   if (isupper(c)) {
     return c;
   }
 }
}
```

Cette fonction lit indéfiniement un caractère sur l'entrée standard, jusqu'à ce qu'il soit égal à une lettre majuscule et il renvoie cette lettre.
Nous insistons sur le point que, bien que tout ceci soit autorisé en `C`, ce genre d'écriture doit être reservée à des cas très particuliers.

3.5 Boucles et dépassement de capacité

Nous attirons votre attention sur le fait que l'évaluation d'une condition de continuité[5] prend en compte la représentation mémoire de la donnée et non la valeur que l'on veut représenter. Par exemple, une variable de type `int` représente un entier mais le type `int` n'est pas l'anneau des entiers. En particulier, le type `int` admet une valeur limite `INT_MAX`. Lorsque celle-ci est dépassée, différentes stragégies peuvent avoir été choisies par le compilateur afin de gérer ce problème. Comme il n'y a pas de comportement imposé par la norme, on dit que le comportement est indéterminé. Ce sont des situations qu'il faut absolument éviter car si on n'y fait pas attention, il est possible d'écrire des programmes qui s'exécutent différemment suivant le compilateur qui a été utilisé ! Éxaminons le bout de code suivant :

```
int i = 1;
while (i > 0) {
   i = i * 2;
   printf("%d\n",i);
 }
 printf("Fin");
```

[5]Et plus généralement, d'une expression booléenne

La première idée que l'on peut avoir est qu'il s'agit d'une boucle infinie qui va afficher, sans jamais s'arrêter, des puissances de 2 successives. En y réflechissant, un peu le fait de multiplier par 2 le contenu de la variable i, va finir par lui faire avoir la valeur (négative) `INT_MIN` ce qui va causer l'arrêt du programme.

```
2
4
8
16
32
64
128
...
268435456
536870912
1073741824
-2147483648 Fin
```

C'est le choix du compilateur `gcc` et des options de compilation qui a determiné ce comportement. Un autre choix aurait pu donner autre chose!
De façon générale, il faut faire attention aux comportements aux limites des types utilisés. Autre exemple : l'évaluation de `3e3+1==3e3` donne `false` alors que l'évaluation de `3e30+1==3e30` donne `true`. Que penser d'une boucle dont le test de continuité serait `x <= 3e30` pour une variable `x` de type `double` servant de compteur de boucle?

3.6 Boucles et saisies

Pour réaliser un nombre indeterminé de saisies au clavier, il est possible d'utiliser la valeur de retour de la fonction `scanf` dans la condition de continuité d'une boucle. Par exemple :

```
int i=1;
while (scanf("%d",&i) == 1) {
  printf("Valeur saisie : %d\n",i);
}
```

Dans ce cas, la boucle s'arrêtera lorsque l'on saisira un caractère qui n'est pas un chiffre. Attention tout de même : cela peut produire une erreur si on veut continuer à lire des entiers. Supposons que l'on ajoute la ligne `assert(scanf("%d",&i) == 1);` après la boucle. Le fait qu'il y a eu une erreur de lecture, a laissé la "tête" de lecture au même endroit. Elle est donc positionnée sur un caractère qui ne permet pas de construire un `int`.

```
1
Valeur saisie : 1
é
main : Assertion ‘scanf("%d",&i) == 1’ failed.
Aborted (core dumped)

------------------
```

```
(program exited with code : 134)
Press return to continue
```

Une solution pour éviter ce soucis, consiste à lire un caractère en utilisant une instruction `getchar()` Néanmoins, cela ne fait que repousser le problème car la même erreur se produit si on a entré deux lettres, trois lettres etc.
Nous allons changer de stratégie. Pour indiquer que l'utilisateur a terminé sa saisie, il utilisera la séquence de touche `ctrl+D`. C'est une façon de forcer la fin de l'entrée standard. Si un tel évenement se produit et que l'on essaie de lire grâce à un `scanf`, la fonction renvoie la valeur `EOF`. Voici donc une façon de procéder :

```
int i=1;
int rep;
do {
  printf("Veuillez saisir un entier :");
  rep = scanf("%d",&i);
  if (rep == 0) {
    printf("Mauvaise saisie recommencez\n");
    getchar();
  } else {
    printf("Valeur saisie : %d\n",i);
  }
} while (rep != EOF) ;

printf("\n Veuillez saisir un dernier entier :");
rewind(stdin);
assert(scanf("%d",&i) == 1);
printf("Valeur saisie : %d\n",i);
```

la variable `rep` récupère la valeur de `scanf`. Lorsque celle-ci vaut 0, cela veut dire que le caractère en cours de lecture ne permet pas de construire un `int`. Dans ce cas, on affiche une erreur, on force la lecture d'un caractère et on recommence. Si on detecte la fin de l'entrée standard, alors on sort de la boucle. Afin d'éviter de rester "coincé" à la fin de la boucle, il faut "rembobiner" en utilisant l'instruction `rewind(stdin)`. On se retrouve alors dans une configuration dans laquelle on peut continuer la lecture.

3.7 De la bonne interprétation de la condition de continuité

Il est interessant de constater que la condition de continuité devient `false` lorsque la boucle se termine. Bien penser à cette remarque qui permet d'éviter quelques erreurs. Donnons un exemple :

```
for(k=0,s=0;k <= 10; ++k) {
  s +=k;
}
```

Quelle est la valeur de `k` après la boucle? `10` ou `11`? Pour bien trancher cette question, il suffit de se rappeler que l'évaluation de `k <= 10` doit donner `false` pour que la boucle se termine. La

valeur de `k` est donc `11`.
En revanche, la variable `s` contient la somme des entiers de 1 à 10 (et non pas 11). En effet, la différence provient de l'instruction `++k;` qui est exécutée après `s += k`.

4 Méthodologie de construction d'une boucle

Écrire proprement le code d'une boucle demande de maîtriser de notions d'algorithmique que nous aborderons au second semestre. Néanmoins, nous pouvons dès maintenant, sur des cas simple, vous sensibiliser à des éléments de méthodologie.
La première étape doit se faire avec un papier et un crayon : il s'agit d'expérimenter afin de trouver une méthode pour résoudre le problème qui nous intéresse. Une fois la méthode choisie, il va falloir la décrire suffisament précisement afin de pouvoir la programmer en `C`. Ici nous supposerons la méthode connue et nous l'illustrerons avec l'exemple du calcul de la somme $1 + \cdots + n$ pour n fourni en entrée.
Nous commençons donc par décrire la méthode que l'on va appliquer en expliquant ce qu'est une étape de calcul. Nous explicitons :

- Une proposition logique `P`, qui est vraie à la fin de chaque tour de boucle. Nous l'appelerons invariant de boucle.
- Une condition (expression booléenne) `C` pour laquelle le calcul est terminé. Nous l'appelerons condition d'arrêt.
- Des valeurs des variables impliquées dans l'expression de `P` pour lesquelles on sait que la proposition logique vaut vraie. Nous appelerons cette configuration conditions initiales.

Cela peut sembler un peu abstrait... donc examinons notre exemple. La méthode que l'on va appliquer consiste à ajouter les entiers un à un à la somme de façon croissante [6]. Pour formaliser ceci, nous allons choisir une variable `s` qui contiendra la somme et une variable `k` qui contiendra de dernier entier que l'on a ajouté. Une étape intermédiaire sera donc caractérisée par la proposition `/* s == 0+1+...+k && k <= n */`, que l'on a écrite ici de façon à pouvoir l'insérer en commentaire. On peut aussi écrire de façon plus précise `/* s == somme( i, i = 0..k) && k<= n */` [7]
L'écriture `somme...` se rapproche de l'écriture mathématique $s = \sum_{i=0}^{k} i$.[8].
Attention :

- Ne pas oublier l'assertion `k <= n` qui indique que le calcul n'a pas nécessairement encore abouti.
- Nous avons introduit une nouvelle indeterminée `i` afin de formaliser la description de la somme. Il s'agit d'une indeterminée mathématique et non d'une variable informatique. Son nom doit être choisi de façon à ce qu'il n'y ait pas d'ambiguité avec les variables du programme (ne pas la confondre avec `k`, par exemple).

[6] Il s'agit de la méthode que l'on a choisi de coder. Nous ne discutons pas ici de l'efficacité de l'algorithme.
[7] Il ne s'agit pas de commande `C`. Le programmeur a donc le droit d'utiliser le formalisme qui lui convient pour exprimer la proposition... du moment que celle-ci est claire et précise et qu'elle puisse être insérée en commentaire dans le programme.
[8] On rappelle la convention suivante : une somme portant sur un intervalle vide vaut 0. Par exemple $\sum_{i=1}^{-1} = 0$.

Le calcul sera considéré comme terminé lorsque la valeur `k` aura atteint ou dépassé la borne `n`. C'est ce qui donne la condition d'arrêt : `k >= n`.
Dernière étape : on connaît une configuration simple pour laquelle l'invariant de boucle est vraie : `s == 0, k == 0`.
De cette analyse nous allons déduire du code `C` qui réalise le calcul voulu. Ce code aura la forme :

```
 /* initialisation */
 I1

 /* Invariant de boucle : ... */

 while (Condition de continuité) {
  I2
}
[I3]
```

- `I1` est un bloc d'instructions choisi pour que la configuration initiale soit atteinte avant le début de la boucle.
- `Condition de continuité` : c'est la négation de la condition d'arrêt.
- `I2` est un bloc d'instructions expliquant comment passer d'une étape à la suivante.
- `[I3]` La boucle peut être éventuellement suivie d'une étape de post-traitement si nécessaire.

Dans notre exemple, cela donnera le code suivant :

```
 /* initialisation */
 int s = 0, k = 0;

 /* Invariant de boucle :
    s == somme( i, i = 0 .. k) && k <= n
  */

 while (k < n) {
    ++k;
    s += k;
}
```

Remarquez que

- La condition de continuité a été obtenue en prenant la négation de la condition d'arrêt `k >= n`.
- `s == somme( i, i = 0 .. k) && k <= n && k >= n` est équivalente à `s == somme( i, i = 0 .. n) && k == n`. Ceci donne l'état des variables après l'exécution de la boucle. Notez bien que `k == n`.

- Il n'y a pas de consigne particulière pour choisir les instructions de `I2`. Il faut bien décrire comment passer à l'étape suivante (en respectant donc l'invariant de boucle). Il y a plusieurs solutions à ce problème.
- Il n'y a pas d'instruction dans `[I3]` car `s` contient le résultat recherché.

On peut éventuellement compléter le travail en remarquant que la valeur de la variable `k` parcourt l'intervalle `{0,..,n}`. Il serait donc légitime d'utiliser une boucle `for`. Nous allons procéder en deux étapes : tout d'abord nous allons écrire la boucle `for` en remplissant les rubrique `initialisation` et `condition` sans toucher à `iteration` et on recopie tel quel le corps de la boucle :

```
  int s, k;
 /* Invariant de boucle :
    s == somme( i, i = 0 .. k) && k <= n
  */

 for(s=0, k=0; k < n;) {
    ++k;
    s += k;
 }
```

Nous voudrions écrire l'incrémentation `++k` dans la rubrique `itération` de la boucle `for`. Pour cela, il faut permuter les deux lignes du corps du programme, en faisant bien attention à modifier l'affectation de `s`

```
++k;            s += k + 1; /* k n'a pas encore été modifié!!*/
s += k;   <=>   ++k;
```

Maintenant, on peut finir la traduction de la boucle :

```
  int s, k :
 /* Invariant de boucle :
    s == somme( i, i = 0 .. k) && k <= n
  */

 for(s=0, k=0; k < n; ++k) {
    s += k + 1;
 }
```