Chapitre 3

Boucles

3.1 Introduction

L'efficacité d'un ordinateur est en partie liée à sa capacité à effectuer un grand nombre d'opérations rapidement. Dans certains cas ces opérations sont différentes, mais dans d'autres cas il faut répéter plusieurs fois la même opération ou le même ensemble d'opérations pour aboutir au résultat escompté.

Les boucles sont un des outils permettant la programmation d'une tâche répétitive et sont donc à ce titre un élément central de la programmation ; Elles *permettent de répéter l'exécution* d'un même ensemble d'instructions (appelé bloc d'instructions) *sans avoir à répéter l'écriture* de ces instructions. Ce bloc d'instructions peut contenir une ou plusieurs instructions de toute nature : affichages, affectations, calculs ... Un bloc d'instructions *placé* dans une boucle est exécuté *plusieurs fois à la suite - i.e. séquentiellement.*

Les boucles permettent donc d'éviter de copier/coller un ensemble d'instructions identiques qui se suivent les unes à la suite des autres. Ainsi, pour afficher à l'écran 100 fois la phrase *"hello world"*, plutôt que d'écrire 100 fois la même instruction d'affichage,

```
affichage("hello world \n") ; affichage("hello world \n" );
... ... ... ... ... ... ... ... ... ... ... ...
affichage("hello world \n") ; affichage("hello world \n" );
```

il suffit d'écrire une seule fois l'instruction `affichage("hello world \n");` et de spécifier que cette instruction doit être effectuée 100 fois par l'ordinateur. Le code est ainsi plus concis, il n'est pas nécessaire d'écrire 100 fois la même instruction.

Remarque : Dans l'exemple précédent le nombre de répétitions est fixé lors de l'écriture du programme au moyen d'une valeur littérale : l'entier 100. Cependant, spécifier explicitement le nombre de répétitions lors de l'écriture du programme n'est pas obligatoire. D'autant que, dans certains cas, le nombre de répétitions peut ne pas être connu lors de la conception du programme !

Exemple : On peut vouloir spécifier que les instructions seront répétées exactement N fois, N ayant une valeur déterminée lors du déroulement du programme. La valeur N peut par exemple être demandée à l'utilisateur. La valeur saisie est alors stockée dans la variable N. Le nombre de répétitions n'est donc plus fixé par un littéral mais par une variable. Cela donne une grande souplesse aux programmes qu'il est possible d'écrire.

3.2 La boucle `while`

La "boucle" (répétition de l'exécution d'un même ensemble d'instructions) est un raisonnement logique indépendant de la programmation qui est couramment utilisé dans la vie quotidienne.

Exemple : Pour planter un clou dans un mur on utilise souvent l'algorithme suivant :

```
1        Etape 1: Je place le clou
2        Etape 2: Tant que le clou n'est pas suffisamment enfoncé, je tape sur le clou.
```

Cet algorithme peut être vu comme une boucle. L'instruction *"je tape sur le clou"* est effectuée autant de fois que nécessaire, et ce, jusqu'à ce que la condition *"le clou n'est pas suffisamment enfoncé"* soit fausse.
À retenir : En algorithmique ce type de raisonnement logique est supporté par une *structure de contrôle* appelée boucle "tant que", aussi appelée boucle "while" (qui veut dire "tant que" en anglais).

3.2.1 Structure du `while`

La boucle `while` s'écrira de la manière suivante :

```
0 Instruction 0;
1 while (condition)
2 {
3   Instruction 1;
4   Instruction 2;
5   ... ;
6   Instruction n;
7 }
8 Instruction n+1;
```

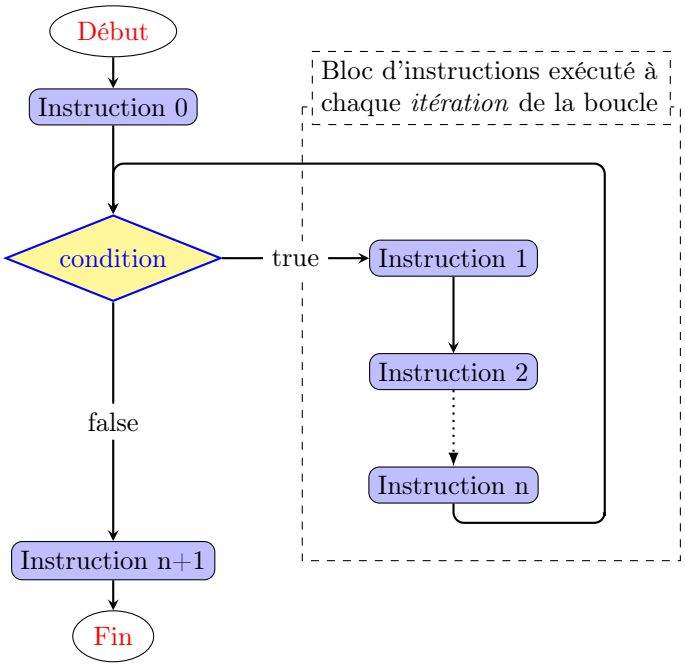


FIGURE 3.1 – Ordinogramme du fonctionnement d'une boucle

Attention :
- Les accolades (ouvrante et fermante) qui permettent de délimiter le corps de la boucle (*i.e.* le bloc d'instructions dont l'exécution est répétée).
- L'indentation des Instructions 1 à n n'est pas obligatoire mais elle permet une meilleur lisibilité de l'algorithme en mettant bien en évidence le *bloc* d'instructions qui est répété dans la boucle, de la suite (l'Instruction n+1) qui n'en fait pas partie.

Ce code indique que les instructions situées entre accolades, *i.e.* les instructions 1 à n, sont répétées *tant que* la `condition` est vraie. Ces instructions peuvent indifféremment être un calcul, une affectation ou encore être elles-mêmes d'autres structures de contrôle de type alternatives (`if`) ou boucles (`while`).

L'ordre dans lequel ces instructions sont exécutées est représenté dans la Figure 3.1 et suit les règles de séquentialité. Lorsque le programme arrive à l'instruction `while`, il évalue la condition associée puis

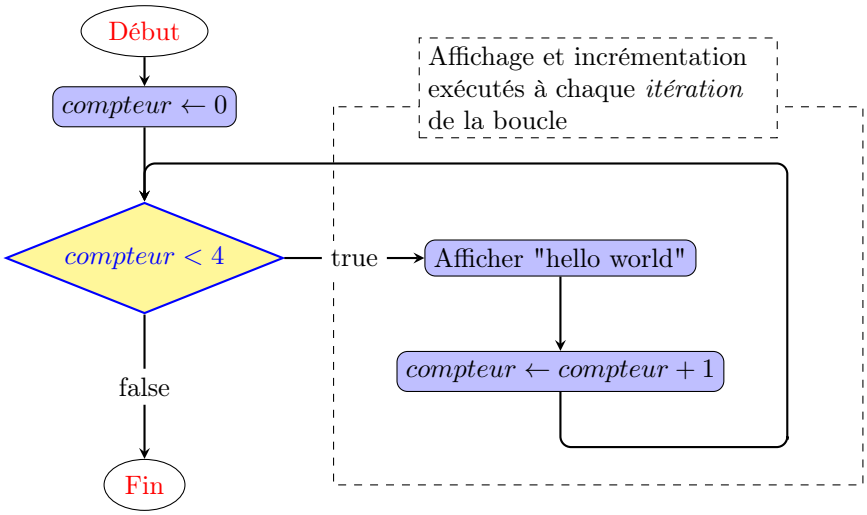
— si `condition` vaut `true`, *i.e.* la condition est vérifiée, alors les instructions 1 à n sont exécutées (c'est le bloc d'instruction) puis on reprend à l'instruction d'évaluation de la condition (on *boucle*).

— si `condition` vaut `false`, *i.e.* la condition n'est pas vérifiée, le programme exécute directement l'instruction n+1 (*on sort de la boucle*).

À retenir : Le fait d'exécuter une fois les instructions 1 à n (*i.e.* exécuter une fois le bloc d'instructions) est appelé une *itération* de la boucle.

Remarque : La `condition` est très souvent donnée par la valeur d'une variable dont on modifie la valeur à chaque itération.

Exemple : Pour afficher 4 fois la phrase "hello world", on peut utiliser une variable, dont la valeur est fixée initialement à 0, et incrémenter cette variable de un à chaque itération. Le bloc d'instructions associé au `while` devra alors être exécuté tant que la valeur de la variable est strictement inférieure à 4.



On obtient alors l'algorithme/code suivant :

```
affichage("********************\n");
// Compteur de boucle commençant à 0
int i = 0;

/*
Le corps de la boucle est repété 4 fois pour les valeurs de i égales à 0, 1, 2, 3.
*/
while (i < 4)
{
  affichage("hello world \n") ;
  i=i+1;
}
// i vaut 4 après la boucle
affichage("********************\n");
```

Pour comprendre ce qui se passe durant l'exécution, on déroule à la main cet exemple :

- ◇ **[ligne 1]** affichage de `"********************"`
- ◇ **[ligne 3]** définition d'une variable entière appelée `i` de valeur 0
- ◇ **[ligne 8]** test `i < 4` avec `i` égal à 0 est vrai
  ⟶ exécuter les instructions des lignes 10 et 11
- ◇ **[ligne 10]** affichage de "hello world"
- ◇ **[ligne 11]** modifie la valeur de la variable `i` : `i` vaut 1
- ◇ **[ligne 12]** fin de bloc
  ↺ on boucle à l'instruction de la ligne 8
- ◇ **[ligne 8]** test `i < 4` avec `i` égal à 1 est vrai
  ⟶ exécuter les instructions des lignes 10 et 11
- ◇ **[ligne 10]** affichage de "hello world"
- ◇ **[ligne 11]** modifie la valeur de la variable `i` : `i` vaut 2
- ◇ **[ligne 12]** fin de bloc
  ↺ on boucle à l'instruction de la ligne 8
- ◇ **[ligne 8]** test `i < 4` avec `i` égal à 2 est vrai
  ⟶ xécuter les instructions des lignes 10 et 11
- ◇ **[ligne 10]** affichage de "hello world"
- ◇ **[ligne 11]** modifie la valeur de la variable `i` : `i` vaut 3
- ◇ **[ligne 12]** fin de bloc
  ↺ on boucle à l'instruction de la ligne 8
- ◇ **[ligne 8]** test `i < 4` avec `i` égal à 3 est vrai
  ⟶ exécuter les instructions des lignes 10 et 11
- ◇ **[ligne 10]** affichage de "hello world"
- ◇ **[ligne 11]** modifie la valeur de la variable `i` : `i` vaut 4
- ◇ **[ligne 12]** fin de bloc
  ↺ on boucle à l'instruction (3)
- ◇ **[ligne 8]** test `i < 4` avec `i` égal à 4 est faux
  ↓ exécuter l'instruction de la ligne 14 (et suivantes)
- ◇ **[ligne 14]** affichage de `"********************"`

L'exécution de ce code produit alors l'affichage suivant :

```
********************
hello world
hello world
hello world
hello world
********************
```

3.2.2 Notation d'affectation compacte

Lorsque l'on affecte à une variable une valeur définie à partir de la variable elle-même, la variable en question apparaît alors dans chaque membre de l'affectation.

Exemple : Pour multiplier la valeur d'une variable par 2, on peut utiliser l'instruction `i=i*2`. La variable `i` apparaît des 2 côtés du signe d'affectation `=`.

À retenir : Si la variable apparaît comme premier terme du membre droit de l'affection, il est possible de rendre l'expression plus compacte en supprimant la variable du membre droit et en plaçant l'opérande avant le signe d'affectation. L'instruction précédente devient alors `i*=2`.

Cette transformation est valide pour les opérandes `+,-,*,/`. Le tableau suivant résume ces raccourcis d'écriture.

▶ Sur ce thème : **Exercices 1, 2 et 3, TD3**

Compacte		Explicite
`a*=2`	⇔	`a=a*2`
`b+=10`	⇔	`b=b+10`
`c-=3`	⇔	`c=c-3`
`d/=4`	⇔	`d=d/4`
`e++`	⇔	`e=e+1`
`f--`	⇔	`f=f-1`

3.3 Construire une boucle : exemple simple

Dans cette partie, on montre comment élaborer une boucle simple. On se concentre sur l'écriture d'un programme permettant d'afficher tous les nombres pairs entre 1 et 10 inclus.

Le premier algorithme qui vient à l'esprit est :

```
affichage( 2 , '\n');
affichage( 4 , '\n');
affichage( 6 , '\n');
affichage( 8 , '\n');
affichage( 10 , '\n');
```

Cependant, c'est loin d'être le meilleur algorithme : il n'est pas assez *general* ! En effet, si l'objectif était d'afficher tous les nombres pairs entre 1 et 10000, l'écriture du programme avec la méthode précédente deviendrait laborieuse et compliquée à vérifier ! Comme on répète presque la même instruction, à savoir *afficher un nombre*, il est possible (en modifiant légèrement le code), d'utiliser une boucle pour afficher ces nombres pairs.

Attention : Dans une boucle, les *instructions* effectuées à chaque itération sont identiques.

Il convient donc de modifier le programme afin de mettre en évidence le caractère similaire de ces affichages. Ici, cela consiste à utiliser une variable `i` pour faire référence toujours de la même manière à la partie variable de l'affichage (2, 4,...,10). Ainsi, l'instruction d'affichage devient identique pour tous les nombres, à savoir `affichage( i , "\n");` où seule la valeur numérique affectée à la variable `i` change et l'algorithme devient :

```
int i; // i déclaré sans initialisation
i = 2;
affichage( i , '\n');
i = 4;
affichage( i , '\n');
i = 6;
affichage( i , '\n');
i = 8;
affichage( i , '\n');
i = 10;
affichage( i , '\n');
```

On répète bien 5 fois strictement la même instruction `affichage( i , "\n");`. En revanche, on effectue 5 affectations différentes de la variable `i`. De plus, l'affectation consiste toujours à ajouter 2 à la valeur de i. On peut donc encore faire apparaître une instruction qui de répète. et remplacer les 5 affectations différentes par l'affectation identique `i = i + 2` ou de façon compacte `i+=2`. Pour que l'algorithme reste juste, il faut préalablement penser à initialiser i avec la valeur 0. L'algorithme devient alors :

```
int i = 0;     // i déclarée et initialisée à 0

i += 2;
affichage( i , '\n');
i += 2;
```

```
6  affichage( i , '\n');
7  i += 2;
8  affichage( i , '\n');
9  i += 2;
10 affichage( i , '\n');
11 i += 2;
12 affichage( i , '\n');
```

Dans cet algorithme, il est facile d'identifier un bloc de deux instructions répété 5 fois. Avant le dernier bloc d'instructions (lignes 11 et 12), la variable i est égale à 8. On répète donc ces instructions tant que la variable i est inférieure ou égale à 8. Ayant identifié un bloc d'instructions à répéter ainsi qu'une condition d'arrêt à cette répétition, la boucle est facile à écrire, et l'algorithme devient le suivant :

```
1 int i = 0;     // initialisation de la boucle avant d'entrer dans la boucle
2 while (i <= 8)
3 {
4   i += 2;
5   affichage( i , '\n');
6 }
```

La difficulté de l'élaboration d'une boucle vient du fait qu'il faut être capable de modifier les instructions afin qu'elles soient identiques lors de chaque itération. L'exemple présenté ici est simple si l'on comprend qu'il s'agit d'afficher toujours une valeur `i` qui est *incrémentée* de deux à chaque itération. Cette "transformation" du code (en instructions identiques) peut cependant être difficile si l'on n'a pas la bonne "intuition".

Attention : En informatique, les usages sont importants. Dans une telle situation, de nombreux programmeurs préfèreront initialiser i à 0 et utiliser une inégalité stricte pour faire apparaitre la borne supérieur de la condition de sortie de boucle. L'algorihtme *standard* serait alors :

```
1 int i = 0;
2 while (i < 10)
3 {
4   i += 2;
5   affichage( i , '\n');
6 }
```

Dans le code précédent, à chaque itération, on effectue d'abord l'incrémentation de la variable i puis l'affichage de i. L'ordre dans lequel ces deux instructions sont effectuées est complètement arbitraire. On pourrait, tout aussi bien considérer qu'à chaque itération, on affiche d'abord i avant de l'incrémenter.

Dans ce cas, comme on souhaite d'abord afficher 2, il faut affecter la valeur 2 à i avant la boucle `while`. De plus, comme le dernier entier à afficher est 10, il est nécessaire d'effectuer le bloc d'instruction tant que i est inférieur ou égal à 10. On obtient alors l'algorithme suivant :

```
1 int i = 2;
2 while (i <= 10)
3 {
4   affichage( i , '\n');
5   i += 2;
6 }
```

À retenir : La boucle est une structure de contrôle adaptée à la répétition d'une même tâche dans un algorithme.

Remarque : À titre d'illustration, avec cet algorithme il n'est pas plus difficile de concevoir un programme affichant tous les entiers pairs inférieur à 10 qu'un programme affichant tous les entiers paris inférieurs à 10000 ! Il suffit de changer la borne de la condition de sortie de la boucle :

```
int i = 0;
while (i < 10000)
{
  i += 2;
  affichage( i , '\n');
}
```

► Sur ce thème : **Exercices 4 et 5, TD3**

3.4 Construire une boucle : autre exemple

Dans les exemples de boucles précédents, le nombre d'itérations était connu à l'avance. Ainsi dans l'algorithme suivant, on sait avant d'évaluer pour la première fois la condition que l'on fera exactement N itérations :

```
int N;
int i = 0;
affichage("Nombre d'affichage\n");
saisie(N)
while (i < N)
{
  affichage( i, '\n');
  i++;
}
```

Dans certains cas, il est plus simple de concevoir des algorithmes dans lesquels le nombre d'itérations n'est pas connu à l'avance. Il n'est alors plus adapté d'initialiser une variable avec une valeur donnée et de la modifier à chaque itération de manière jusqu'à ce qu'elle atteigne une valeur fixée. Il faut donc trouver une autre façon d'exprimer la condition de sortie de boucle `while` (d'arrêt des itérations).

Exemple : Pour illustrer ceci, on intéresse à la conception d'un algorithme permettant de calculer la moyenne des notes d'un étudiant.

Les spécifications décrivant ce que l'algorithme doit faire sont les suivantes :

1. On suppose que le nombre de notes dont on calcule la moyenne n'est pas connu à l'avance.
2. On décide que l'utilisateur doit saisir au clavier les différentes notes.
3. Tout nombre réel compris entre 0.0 et 20.0 est une note et tout nombre réel hors de cet intervalle n'est pas une note.
4. Lorsque l'utilisateur saisit un nombre qui ne correspond pas à une note, alors on doit considérer que la saisie des notes est terminée.
5. On doit alors afficher le résultat du calcul de la moyenne des notes saisies par l'utilisateur.

La saisie d'une note peut être répétée indéfiniment (jusqu'à ce que l'utilisateur choisisse de saisir autre chose qu'une note). L'utilisateur peut choisir de saisir 3, 10 ou n'importe quel nombre de notes. Le critère d'arrêt n'est plus lié à un compteur numérique du nombre de notes, mais à un *"événement"* : l'utilisateur saisit autre chose qu'une note. Toutefois tant que l'utilisateur saisit des notes valides, on répète identiquement les mêmes instructions consistant à récupérer le nombre saisi au clavier, c'est-à-dire, l'instruction `saisie(nombre);`.

Une ébauche de l'algorithme comportant une boucle pourrait être :

```
//Déclaration des variables nécessaires à l'algorithme
//Important ! Leur initialisation sera abordée plus loin

double nombre; // Contiendra la note courante saisie au clavier
// tant que la note est comprise entre 0 (inclus) et 20 (inclus)
while (nombre >= 0.0 && nombre <= 20.0)
{
  affichage("Saisir une note comprise ente 0 et 20 (bornes incluses)) : ") ;
  saisie(nombre);
}
```

La saisie des nombres ne suffit évidement pas pour réaliser l'algorithme. Afin de calculer la moyenne de plusieurs notes, il faut connaître le **nombre total de notes** ainsi que la **somme de ces notes**.

Il faut donc deux nouvelles variables :

— `nbNotes` comptant le nombre de notes qui seront saisies,
— `somme` contenant la somme cumulée des notes saisies.

On peut alors procéder, à chaque itération, de la façon suivante :

— Pour compter le nombre de note, il suffit d'incrémenter la variable `nbNotes`.
— Pour calculer la somme, il suffit d'ajouter à `somme` la valeur de la variable `nombre` contenant la dernière note saisie par l'utilisateur.
— On finit en affichant un message demandant à l'utilisateur de saisir la prochaine note.

L'algorithme devient alors :

```
/*
Déclaration des variables nécessaires à l'algorithme
Important ! Leur initialisation sera abordée plus loin
*/
double somme;          // contiendra la somme des notes
int nbNotes;           // contiendra le nombre de note effectivement saisies
double nombre;         // Contiendra la note courante saisie au clavier

while (nombre >= 0.0 && nombre <= 20.0)
{
  nbNotes++;           // Une note de plus est comptabilisée
  somme += note;       // On ajoute la note courante à la somme
  affichage("Saisir une note comprise ente 0 et 20 (bornes incluses)) : ") ;
  saisie(nombre);
}
```

L'ordre est très important ! Comme l'utilisateur peut saisir un nombre qui ne correspond pas à une note, il est nécessaire de vérifier que `nombre` est dans le bon intervalle avant de l'ajouter à `somme`. Pour cela, la saisie de `nombre` est la dernière instruction du `while`. Si ce nombre est une note, autrement dit, si la condition du `while` est vraie, alors on incrémente `nbNotes` de +1 et on ajoute la dernière note saisie à `somme` à l'itération suivante.

Il faut aussi faire attention aux **valeurs initiales des variables**. En effet, lors du premier test du `while`, **la variable `nombre` n'est pas encore initialisée**. Une solution consiste à y *stocker* la **première note à saisir** par l'utilisateur en ajoutant avant la boucle l'instruction `saisie(nombre);`. Afin que le résultat soit correct, il faut également initialiser à 0 les variables `nbNotes` et `somme`.

L'algorithme devient alors :

```
// Déclaration des variables et initialisation nécessaires à l'algorithme

double somme = 0.0;          // contiendra la somme des notes
int nbNotes = 0.0;           // contiendra le nombre de note effectivement saisies
double nombre;        // Contiendra la note courante saisie au clavier

// Saisie de la première note
affichage("Saisir une note comprise ente 0 et 20 (bornes incluses)) : ") ;
saisie(nombre);

while (nombre >= 0.0 && nombre <= 20.0)
{
  nbNotes++;
  somme += nombre;
  // Saisie des notes suivantes
  affichage("Saisir une note comprise ente 0 et 20 (bornes incluses)) : ") ;
  saisie(nombre);
}
```

À la fin de cette boucle, les variables `somme` et `nbNotes` correspondent respectivement à la somme des notes saisies et à leur nombre. Pour afficher la moyenne, il faut donc afficher `affichage("La moyenne est ", somme/compteur , "\n");`.

Cependant, il faut impérativement que le programme fonctionne quelles que soient les notes saisies par l'utilisateur. Il est donc possible que l'utilisateur ne saisisse aucune note. Dans ce cas, `nbNotes` est nul et l'affichage engendre une division par 0, donc une erreur. Un test sur le nombre de notes saisies est alors nécessaire. Le code après la boucle devient alors

```
//Au moins une note a été saisie pour le calcul de la moyenne
if (nbNotes==0)
  affichage("Aucune note n'a été saisie. On ne peut pas calculer la moyenne !\n");
else
  affichage("La moyenne est ", somme/nbNotes , '\n');
```

L'algorithme complet correspondant au problème est finalement :

```
int main()
{
  // Déclaration des variables nécessaires avec les initialisations nécessaires
  double somme = 0.0;         // contiendra la somme des notes
  int nbNotes = 0.0;          // contiendra le nombre de note effectivement saisies
  double nombre;              // Contiendra la note courante saisie au clavier

  // Saisie de la première note
  affichage("Saisir une note comprise ente 0 et 20 (bornes incluses)) : ") ;
  saisie(nombre);
  while (nombre >= 0.0 && nombre <= 20.0)
  {
    nbNotes++;
    somme += nombre;
    // Saisie des notes suivantes
    affichage("Saisir une note comprise ente 0 et 20 (bornes incluses)) : ") ;
    saisie(nombre);
  }

```

```
20  if (nbNotes==0)
21  {
22    affichage("Aucune note n'a ete saisie. On ne peut pas calculer la moyenne !");
23  }
24  else
25  {
26    affichage("La moyenne est ", somme/nbNotes, '\n');
27  }
28  return 0;
29 }
```

▶ Sur ce thème : **Exercices 6 et 7, TD3**

3.5 Boucles infinies

Les boucles infinies sont des boucles dont les instructions sont exécutées une infinité de fois. De telles boucles interviennent lorsque la condition du while est toujours vraie. Elles correspondent à une erreur de programmation et ne sont pas détectées par l'ordinateur. De telles erreurs sont dues soit à une mauvaise initialisation de la ou des variables intervenant dans la condition, soit à une modification inadaptée ou inexistante de ces variables au cours d'une itération de la boucle, soit à une mauvaise conception du test.

Voici un exemple de boucle infinie :

```
1 int i = 1;
2 while (i != 10)
3 {
4   i += 2;
5   affichage( i, '\n');
6 }
```

Un peu (ou beaucoup !) de reflexion avant l'écriture d'une boucle permet d'éviter de telles boucles infinies. ▶ Sur ce thème : **Exercice 8, TD3**

TD3 : Boucles (Corrigé)

✓ Exercice 1 : Test de compréhension*

Question 1.1 : [Syntaxe] Quelle est la syntaxe du `while` ?
Correction :

```
//Initialisation des variables nécessaires au bon fonctionnement de la boucle;
while (condition)
{
  Instructions du corps de la boucle;
}
```

◇

Question 1.2 : [Corps de la boucle] Comment délimite -t-on le corps d'une itérative ?
Correction :

Corps de la boucle : On utilise des accolades. L'accolade ouvrante délimite le début de la séquence faisant partie du corps de la boucle et l'accolade fermante délimite la fin de la séquence appartenant au corps de la boucle. S'il n'y a **qu'une seule instruction** dans le corps de la boucle `while`, les accolades ne sont pas obligatoires. ◇

Question 1.3 : [Itération] Qu'est ce qu'une *itération* ?
Correction :

L'exécution du bloc d'instructions associé au `while` pour une valeur de la condition du `while` évaluée à `true`. ◇

Question 1.4 : [Nombre minimal d'itérations] Y a-t-il toujours au moins une itération dans une boucle ? Justifier votre réponse.
Correction :

Si la condition régissant le `while` n'est pas vérifiée au départ (évaluée à `false`), il n'y a aucune itération. ◇

Question 1.5 : [Initialisation] Qu'est-ce qu'une *initialisation* ? Pourquoi est-ce une notion importante dans le cas des boucles ?
Correction :

L'initialisation consiste à affecter une valeur *initiale* à une variable, notamment avant un test ou une boucle. L'initialisation des variables intervenant dans les conditions de boucle joue un rôle essentiel car elle permet au programmeur de contrôler la situation initiale et, par suite, son évolution au cours des itérations. Cela évite notamment les boucles infinies ou les boucles contenant 0 itérations qui n'étaient pas prévues. ◇

Question 1.6 : [Affectation compacte] À quoi correspondent les instructions compactes `i+=3, j-=7, k*=2, h/=4` ?
Correction :

Elles correspondent respectivement à `i=i+3, j=j-7, k=k*2, h=h/4` ◇

Question 1.7 : [Affectation compacte] Comment écrire de manière compacte `a=3+a, a=8*a, a=2-a, a=3/a` ?
Correction :

Les deux premières instructions s'écrivent sous la forme `a+=3, a*=8`, les deux suivantes ne peuvent être écrites de manière compacte. ◇

✓ Exercice 2 : Trace de programme*

Effectuer la trace de ce programme :

```
int i = 10;
int nb = 1;

while (i < 5)
{
  i++;
  nb += 2 * i;
}

affichage("nb = ", nb, '\n');
```

Même question lorsque la première instruction est remplacée par l'instruction `i = 0` puis par `i = 3`.

Correction :

On obtient l'affichage suivant :

```
Pour i = 10, on a nb = 1
Pour i = 0, on a nb = 31
Pour i = 3, on a nb = 19
```

◇

® Exercice 3 : Trace de programme**

Effectuer la trace de ce programme :

```
int i = 12;
int j = 43;

affichage("i = ", i, ", j = ", j, '\n');
while (i < 25 && j > 3*i)
{
  i += 3;
  j -= i - 18;
  affichage("i = ", i, ", j = ", j, '\n');
}
```

Correction :

On obtient l'affichage suivant :

```
i = 12, j = 43
i = 15, j = 46
i = 18, j = 46
```

◇

✓ Exercice 4 : Affichage d'entiers et décrémentation*

Afficher les entiers de 1 à 20 dans l'ordre décroissant.

Correction :

```
int i = 20;
while (i > 0)
{
  affichage( i, '\n');
  i --;
}
```

® Exercice 5 : Table de multiplication*

Afficher la table de multiplication de 7 jusqu'à 20 comme suit :

1 * 7 = 7
2 * 7 = 14
...
19 * 7 = 133
20 * 7 = 140

Correction :

```
int i = 1;
while (i <= 20)
{
  affichage( i, " * 7 = ", 7 * i, '\n');
  i ++;
}
```

✓ Exercice 6 : Saisie contrôlée d'un entier positif**

Écrire un programme demandant à l'utilisateur de saisir

1. un nombre entier positif. La saisie sera répétée jusqu'à ce que le nombre soit positif
2. un nombre entier positif et multiple de 3.

Correction :

```
//Question 1
affichage("Saisissez un nombre entier positif: ");
int nb;
saisie(nb);
while (nb < 0)
{
  affichage("Saisissez un nombre entier positif: ");
  saisie(nb);
}

//Question 2
affichage("Saisissez un nombre entier positif et multiple de 3: ");
int nb;
saisie(nb);
while ( nb < 0 || nb%3!=0)
{
  affichage("Saisissez un nombre entier positif et multiple de 3: ");
  saisie(nb);
}
```

◇

® Exercice 7 : Test de primalité**

Écrire un programme permettant de vérifier si un nombre saisi par l'utilisateur est premier. *Rappel :* Un nombre est dit *premier* si ses deux seuls diviseurs entiers positifs sont 1 ou lui-même. Le nombre 13 est donc un nombre premier, alors que 6 ne l'est pas puisque $6 = 2 \times 3$.

Correction :

```
affichage("Saisissez un nombre pour déterminer s'il est premier : \n");
int nb;
saisie(nb);
int tmp = 2;

//Comme il n'y a qu'une instruction dans le while, on peut ne pas mettre d'accolades
while (nb%tmp != 0)
{
  tmp++;
}

//Comme il n'y a qu'une instruction associée au if et une seule instruction
//associée au else, on peut ne pas mettre d'accolades
if (tmp == nb)
{
  affichage(nb, " est premier \n");
}
else
{
  affichage(nb, "n'est pas premier, il est divisible par ", tmp, '\n');
}
```

Cependant, il est clair que l'on n'est pas obligé de tester tous les entiers compris entre 2 et la valeur saisie par l'utilisateur moins 1. Il est possible de s'arrêter aux entiers inférieurs à la racine carrée de ce dernier. En tenant compte de cette propriété, on peut alors modifier l'algorithme de la manière suivante :

```
affichage("Saisissez un nombre pour déterminer s'il est premier : \n");
int nb;
saisie(nb);
int tmp = 2;

while ((nb%tmp!= 0) && (tmp*tmp < nb))
  tmp++;

if (tmp*tmp >= nb )
  affichage(nb, " est premier \n");
else
  affichage(nb, "n'est pas premier, il est divisible par ", tmp, '\n');
```

Ce deuxième algorithme est ainsi beaucoup plus rapide lorsque le nombre à tester est premier. À titre d'exemple, 65537 est premier. Avec l'algorithme 1, on teste donc tous les entiers jusqu'à 65537 alors qu'avec l'algorithme 2, on teste seulement tous les entiers compris entre 1 et 257 ($257^2 = 66049$). ◇

Exercice 8 : Encore et toujours plus*

Ecrivez un programme permettant d'afficher indéfiniment les entiers successifs à partir de 0.

Correction :

```
int main() {
       int i = 0 ;
       while ( i > -1 ) {
              affichage( i, '\n') ;
              i ++ ;
       }
       return 0;
}
```

TP3 : Boucles (Corrigé)

Exercice 9 : Carrés et cubes*

Écrire un programme qui, pour tout entier compris entre 4 et 9 (compris), affiche sur une même ligne, les valeurs de cet entier, de son carré et de son cube. L'affichage doit donc être équivalent à :

```
4  16  64
5  25  125
6  36  216
7  49  343
8  64  512
9  81  729
```

Correction :

```
int main(){
        int carre;
        int cube;
        int i=4;
        while (i < 10) {
                carre = i * i ;
                cube = i * carre ;
                affichage(" ", i, " ", carre, " ", cube, '\n');
                i ++ ;
        }
        return 0;
}
```

Exercice 10 : Calcul d'intérêts*

Soit un compte bancaire permettant de placer son argent à taux fixe. Définissez un programme permettant :

1. de saisir le taux d'intérêt annuel du compte,
2. de saisir une somme initiale placée sur le compte,
3. de saisir la durée en années du placement.

Le programme calculera et affichera le solde du compte à l'issue de la durée du placement.
Correction :

```
int main(){
        double taux ;
        double somme ;
        double duree ;

        affichage("Saisir le taux d'interet (%): \n") ;
        saisie(taux) ;
        affichage("Saisir la somme du depot initial:\n") ;
        saisie(somme) ;
        affichage("Saisir la duree du placement:\n") ;
        saisie(duree) ;

```

```
13        while (duree > 0)
14        {
15                somme = somme * (1 + taux / 100 ) ;
16                duree -- ;
17        }
18        affichage("Somme a l'issue du placement = ", somme, '\n') ;
19        return 0;
20 }
```

Exercice 11 : Calcul du modulo sans l'opérateur module %**

On souhaite écrire un algorithme qui affiche si un nombre entier est divisible ou non par un autre. Quelque soit l'ordre de saisie des arguments, on cherchera toujours à savoir si le plus grand nombre est divisible par le plus petit. Les deux nombres seront fournis par l'utilisateur.

Vous ne devrez pas utiliser l'opérateur modulo (%), mais la méthode des soustractions successives.

Exemple : : Si `a = 6` et `b = 2`, ou `a=2` et `b=6`, on appliquera la méthode :
`6 -2 = 4`
`4 - 2 = 2`
`2 - 2 = 0` (c'est fini car je ne peux plus soustraire 2 à 0)
Donc 6 est divisible par 2 (car la dernière soustraction permet d'obtenir un 0).

Exemple : : Si `a = 7` et `b = 3`, ou `a=3` et `b=7`, on appliquera la méthode :
`7 - 3 = 4`
`4 - 3 = 1` (c'est fini car je ne peux plus soustraire 3 à 1)
Donc 7 n'est pas divisible par 3 (car la dernière soustraction ne permet pas d'obtenir un 0).

Correction :

```
1  int main(void)
2  {
3    // Données en entrées non initialisées
4    int nbA;
5    int nbB;
6
7    affichage("Entrez un premier nombre :");
8    saisie(nbA);
9    affichage("Entrez un deuxième nombre :");
10   saisie(nbB);
11
12   if (nbA<nbB)
13   {
14     int tmp = nbA;
15     nbA = nbB;
16     nbB = tmp;
17   }
18
19   while (nbA - nbB >= 0)
20     nbA -= nbB;
21
22   if (nbA == 0)
23     affichage("L'un des deux nombres divise l'autre\n");
24   else
25     affichage("Aucun des deux nombres n'est un diviseur de l'autre\n");
26
27   return 0;
28 }
```

Exercice 12 : Factorielle**

Écrire un programme permettant de calculer la factorielle d'un nombre saisi par l'utilisateur. *Rappel :* si n est un entier positif, alors la *factorielle* de n, notée $n!$, est égale à :

$$n! = n \times (n-1) \times (n-2) \times \cdots \times 2 \times 1$$

Par convention, $0! = 1$.

Correction :

```
int main()
{
  //Saisie d'un nombre positif ou nul
  affichage("Saisissez un nombre positif ou nul : ");
  int nb;
  saisie(nb);

  // Filtrage de nb
  while (nb < 0 )
  {
    //Saisie d'un nombre positif ou nul
    affichage("Saisissez un nombre positif ou nul");
    saisie(nb);
  }

  // Si le nombre saisi vaut 0, alors la factorielle vaut 1
  if (nb==0)
    affichage("0! = 1\n");
  else
  {
    // Calcul de la factorielle
    int res = 1;
    int compt = nb ;  // Pour ne pas modifier les données en entrées de l'algorithme

    while (compt > 1)
    {
      res *=compt;
      compt --;
    }

    // Affichage du résultat
    affichage(nb, " ! = ", res, '\n');
  }
  return 0;
}
```

Exercice 13 : Chiffres romains***

Écrire un programme demandant à l'utilisateur un nombre compris entre 1 et 50 et affichant ce nombre en chiffres romains.

Correction :

```
int main()
{

  int n;
  affichage("Saisir un entier entre 1 et 50 inclus : ");
  saisie(n) ;
  while (n < 1 || n > 50)
  {
    affichage("Saisir un entier entre 1 et 50 inclus : ");
    saisie(n) ;
  }

  string s = "";              // Chaine resultat

  if (n >= 50)
  {
    s += "L";
    n -= 50;
  }
  if (n >= 40)
  {
    s += "XL";
    n -= 40;
  }
  while (n >= 10)
  {
    s += "X";
    n -= 10;
  }
  if (n >= 9)
  {
    s += "IX";
    n -= 9;
  }
  if (n >= 5)
  {
    s += "V";
    n -= 5;
  }
  if (n >= 4)
  {
    s += "IV";
    n -= 4;
  }
  while (n >= 1)
  {
    s += "I";
    n --;
  }
  affichage("En romain :", s, '\n');
  return 0;
}
```