

UNIVERSITÉ DE LIÈGE

INFO0946

INTRODUCTION À LA PROGRAMMATION

---

# Un exercice dont vous êtes le Héros · l'Héroïne

## Manipulation de Structures de Contrôle – Itération

---

Benoit DONNET

Simon LIÉNARDY

Tasnim SAFADI

23 septembre 2020



# Préambule

## Exercices

Dans ce « TP dont vous êtes le héros », nous vous proposons de suivre pas à pas la résolution d'un exercice sur la lecture et la compréhension de code (en particulier les boucles). En outre, vous pourrez vous exercer à manipuler syntaxiquement et sémantiquement les différents types de boucles.

**Il est dangereux d'y aller seul <sup>1</sup> !**

Partagez vos commentaires, questions, solutions alternatives sur le forum [eCampus](#). N'hésitez jamais !

---

1. Référence vidéoludique bien connue des Héros.

## 2.1 Rappels sur le Concept d'Itération

Une *itération* ou *structure itérative* est une séquence d'instructions (placées dans un *bloc*) destinée à être exécutée plusieurs fois. On parle parfois aussi de *structures répétitives* : c'est schtroumpf vert et vert schtroumpf.

Le but d'une boucle est de répéter un bloc d'instructions plusieurs fois en un minimum d'instructions (plutôt qu'un copier/coller bête et méchant des instructions à répéter). Dans cette optique, on peut voir la boucle comme du *sucré syntaxique*.

Selon le type de boucle, le bloc d'instructions sera répété un nombre fixe de fois ou selon un certain nombre de critères (combinés à l'aide des opérateurs booléens – cfr. Chap. 1).

Aussi longue soit-elle, une boucle compte toujours pour une seule instruction.

On envisage trois types de boucle :

1. la boucle **tant que** (instruction **while**)
2. la boucle **pour** (instruction **for**)
3. la boucle **faire ... tant que** (instruction **do ... while**)

## 2.2 Enoncé

Soit le bout de code suivant :

```
1 unsigned int x, k, n;  
2  
3 scanf("%u %u", &k, &n);  
4  
5 for(x = k; x >= n; x -= n);  
6  
7 printf("%u\n", x);
```

▷ **Exercice** Que calcule ce code (i.e., que contient la variable `x` à la fin de l'exécution de ce code) ? Soyez le plus précis possible (notamment dans le vocabulaire que vous utilisez).

▷ **Exercice** Réécrivez la boucle `for` (ligne 5) du code ci-dessus à l'aide d'une boucle `while`.

▷ **Exercice** Réécrivez la boucle `for` (ligne 5) du code ci-dessus à l'aide d'une boucle `do ... while`.

### 2.2.1 Méthode de résolution

Voici le programme :

1. Compréhension du code (Sec. 2.3);
2. Réécriture avec une boucle `while` (Sec. 2.4);
3. Réécriture avec une boucle `do ... while` (Sec. 2.5);

## 2.3 Compréhension du Code

Avant d'entamer la résolution de l'exercice, il est impératif que vous vous sentiez à l'aise avec la notion de traitement itératif. Si ce n'est pas le cas, jetez un oeil au **rappel**.

- Si vous voyez directement comment procéder, voyez la suite **2.3.5**
- Si vous êtes un peu perdu, voyez le rappel sur le fonctionnement de la boucle **for** **2.3.1**
- Si l'opérateur = vous semble obscur, voyez le rappel sur l'opérateur d'affectation **2.3.2**
- Si les entrées/sorties sont un concept compliqué, voyez le rappel **2.3.3**
- Si vous ne voyez pas comment démarrer, voyez l'indice **2.3.4**

### 2.3.1 Rappel sur la Boucle for

L'instruction `for` est une **structure itérative** qui permet de spécifier explicitement les opérations à effectuer :

1. avant d'entrer dans la boucle pour la première fois ;
2. afin de décider s'il faut continuer (ou non) à itérer ;
3. entre deux itérations ;
4. dans le bloc d'instructions à répéter.

La syntaxe<sup>2</sup> de l'instruction `for` est la suivante :

```
1 for(instr1; expression; instr2){  
2   instr3;  
3 }
```

L'instruction commence toujours par le mot-clé `for` (*pour*, en français). Dans un langage de programmation, un *mot-clé* est un mot réservé qui a une signification bien particulière dans le langage. Cela signifie donc que vous ne pouvez pas déclarer une variable (par exemple) dont l'identificateur est `for`. Cela provoquerait nécessairement une erreur de compilation car le compilateur ne pourra pas faire la différence entre une boucle `for` et une variable nommée `for`.

Le mot-clé `for` est suivi par des parenthèses entre lesquelles on place trois informations séparées par un `;`. Chacune de ces informations se présente sous la forme d'une expression. La différence entre les trois expressions se situe dans le flux d'exécution de la boucle (cfr. la sémantique de la boucle `for` ci-dessous). La première expression (`instr1`) permet de construire une instruction exécutée une seule fois avant la boucle. La deuxième `expression` est évaluée de manière booléenne et indique si oui ou non on peut rentrer dans la boucle (i.e., exécuter le **bloc d'instructions** associé à la boucle). Enfin, la troisième expression, `instr2`, construit une instruction qui sera exécutée après le **bloc d'instructions** associé à la boucle. Après les parenthèses, nous trouvons un **bloc d'instructions** (appelé le *Corps de la Boucle*), entouré d'accolades (elles ne sont pas obligatoires si le **bloc** contient une seule instruction). Les différentes instructions du **bloc** sont séparées par un `;`.

L'instruction `for` est comprise, par le compilateur, comme une seule instruction (quelque soit la quantité d'instructions du **bloc**). Il n'est néanmoins pas nécessaire de la terminer par un `;` (les accolades jouent ce rôle).

La sémantique<sup>3</sup> de l'instruction `for` est donnée par la Fig. 1. Dans cette figure, le **bloc magenta** correspond aux instructions avant la boucle. Le **bloc couleur asperge** correspond aux instructions après la boucle. Les flèches indiquent le sens du flux du programme, i.e., l'ordre d'exécution des différentes instructions et d'évaluation des expressions.

La Fig. 1 indique que lorsque le **bloc magenta** se termine, `instr1` est exécutée. Cette instruction ne sera exécutée qu'une et une seule fois. Typiquement, `instr1` sera l'initialisation du compteur de la boucle. Ensuite, on évalue l'`expression` de manière booléenne. Si c'est vrai ( $\neq 0$ ), alors on entre dans la boucle et on exécute le Corps de la Boucle (`instr3`). Une fois le Corps de la Boucle terminé, `instr2` est exécutée. Typiquement, il s'agit de l'incréméntation (ou décréméntation) du compteur de boucle. Ensuite, on réévalue l'`expression`. Quand l'évaluation devient fausse ( $= 0$ ), alors on exécute les **instructions** qui suivent la boucle.

2. La syntaxe fait référence à l'ensemble des règles grammaticales d'une langue, i.e., comment les différents éléments doivent être assemblés pour former une phrase. Appliquée à la programmation (le C est un langage), c'est l'ensemble des règles permettant de construire des instructions correctes. La syntaxe des langages de programmation sera étudiée en détails dans le cours [INFO0085](#).

3. La sémantique fait référence à l'étude du sens, de la signification des signes, dans le langage. Appliquée à la programmation, la sémantique consiste à donner une signification mathématique au programme. La sémantique permettra, entre autre, de faire des raisonnements formels – donc mathématiques – sur le programme. Ceci est intéressant pour les programmes requérant un niveau qualité très important (e.g., l'informatique dans l'aéronautique, dans les véhicules, ...). La sémantique des langages de programmation sera étudiée en détails dans le cours [INFO0085](#).

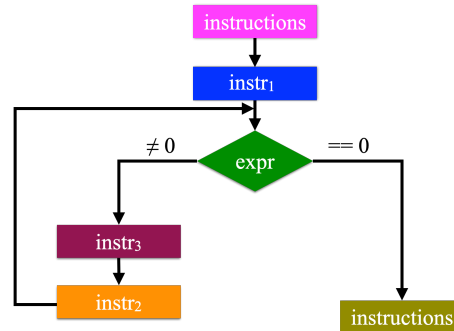


FIGURE 1 – Sémantique de l'instruction **for**. Un rectangle représente une suite d'instructions – éventuellement sous la forme d'un bloc (sauf pour le blocs **bleu** et **orange** qui représentent chacun une seule instruction). Un losange représente expression évaluée de manière booléenne (i.e., une condition).

#### Alerte : Vocabulaire

Le vocabulaire associé à une boucle est extrêmement important, dans le cadre de ce chapitre (bien entendu) mais, plus largement, dans le cadre des cours INFO0946 et INFO0947. Voici les éléments importants :

*Gardien de Boucle.* Il s'agit de l'expression qui est évaluée avant chaque entrée dans le Corps de la Boucle (losange vert sur la Fig. 4). Le résultat de l'évaluation de l'expression est de type booléen. Si l'évaluation est vraie ( $\neq 0$ ), alors le Corps de la Boucle est exécuté. Si l'évaluation est fausse ( $= 0$ ), alors on sort de la boucle et on exécute l'instruction (ou la suite d'instructions) après la boucle (rectangle de couleur asperge sur la Fig. 4). Le Gardien de Boucle peut être une expression simple ou une expression plus complexe construite à l'aide des opérateurs booléens.

*Critère d'Arrêt.* Il s'agit de la condition qui est rencontrée (i.e., vraie) quand on sort de la boucle. Il y a donc un lien extrêmement fort entre Gardien de Boucle et Critère d'Arrêt : le Critère d'Arrêt est **la négation** du Gardien de Boucle. Le Critère d'Arrêt est un concept important. Toute boucle doit, un jour, rencontrer son Critère d'Arrêt. Sinon, la boucle ne se termine pas. On parle alors de *boucle infinie*. C'est l'une des erreurs les plus fréquentes et les plus mortelles.

*Corps de la Boucle.* Il s'agit de l'instruction ou du bloc d'instruction qui est exécuté chaque fois que le Gardien de Boucle est vrai (instr3 et instr2 sur la Fig. 4). Ce bloc est donc répété autant de fois que nécessaire, jusqu'à ce que le Critère d'Arrêt soit atteint.

#### Suite de l'Exercice

À vous ! Déterminez ce que calcule le code et passez à la Sec. 2.3.5.

Si l'opérateur  $=$  vous semble obscur, passez à la Sec. 2.3.2 pour un rappel.

Si les entrées/sorties sont un concept compliqué, passez à la Sec. 2.3.3 pour un rappel.

Si vous ne voyez pas comment démarrer, voyez la Sec. 2.3.4 pour une astuce.

### 2.3.2 Rappels sur l'Opérateur d'Affectation

L'opérateur d'*affectation* (=) est probablement l'opérateur le plus important car il permet la sauvegarde des informations en mémoire (i.e., dans une variable). Il se présente de la façon suivante :

```
1 var = expr
```

L'opérateur d'affectation permet de stocker le résultat de l'expression `expr` (appelée aussi *valeur à droite*) dans la variable `var` (appelée aussi *valeur à gauche*). Il est évident que le type du résultat de l'évaluation de la valeur à droite doit être cohérent avec le type de la valeur à gauche. On ne stocke pas des poires dans des abricots!

La priorité des opérateurs indique bien que l'évaluation de l'expression se fait de droite (la valeur à droite) à gauche (la valeur à gauche). Dès lors, après l'affectation, l'expression entière devient égale à la valeur affectée. Le fonctionnement général de l'affectation est illustré à la Fig. 2.

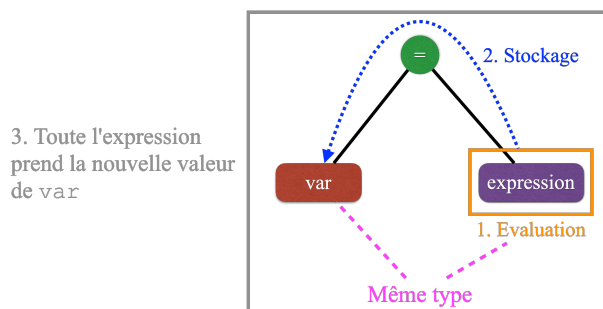


FIGURE 2 – Illustration du fonctionnement général de l'affectation.

#### Alerte : Risque de confusion

Il est très fréquent que les étudiants confonde l'opérateur d'*affectation* (=) et l'opérateur d'*égalité* (==). Le premier permet de stocker le résultat d'une expression dans une variable (et l'expression complète est évaluée à la valeur stockée), tandis que le deuxième permet de tester l'égalité entre deux expressions (le résultat de l'expression est alors booléen, i.e., soit *vrai*, soit *faux*). La confusion entre les deux opérateurs peut amener à des situations problématiques, en particulier dans le cadre des structures de contrôle (cfr. Chap. 2).

Une expression classique qui utilise l'opérateur d'affectation est l'incrément (ou la décrémentation) d'une variable.<sup>4</sup> Par exemple :

```
1 x = x + 2
```

Dans ce cas, l'expression va stocker dans la variable `x` l'ancienne valeur de `x` augmentée de 2.

Le langage C fournit un raccourci (on parle de *sucré syntaxique*<sup>5</sup>) pour ce genre d'expression qui permet de combiner en un seul opérateur l'affectation et l'opération arithmétique (i.e., incrément ou décrémentation).

4. *Incrémenter* signifie *augmenter* une valeur. À l'inverse, *décrémenter* signifie diminuer une valeur.

5. Le sucre syntaxique est une extension de la syntaxe d'un langage de programmation afin de le rendre plus agréable à lire et à écrire, sans changer son expressivité.



Le sucre syntaxique pour l'incrémentation/décrémentation se présente de la façon suivante :

1 `var  $\alpha$  = expr`

où  $\alpha \in \{+, -, *, /, \%\}$ .

Ce raccourci est équivalent à

1 `var = var  $\alpha$  expr`

L'évaluation de l'expression se fait comme pour une affectation normale.

Attention, l'utilisation du sucre syntaxique ajoute, implicitement, des parenthèses autour de `expr`. Il faut donc comprendre l'expression comme suit :

1 `var = var  $\alpha$  (expr)`

Par exemple :

1 `x += 2`

est équivalent à

1 `x = x + 2`

### Suite de l'Exercice

À vous ! Déterminez ce que calcule le code et passez à la Sec. [2.3.5](#).

Si les entrées/sorties sont un concept compliqué, passez à la Sec. [2.3.3](#) pour un rappel.

Si vous ne voyez pas comment démarrer, voyez la Sec. [2.3.4](#) pour une astuce.

### 2.3.3 Rappels sur les Entrées/Sorties

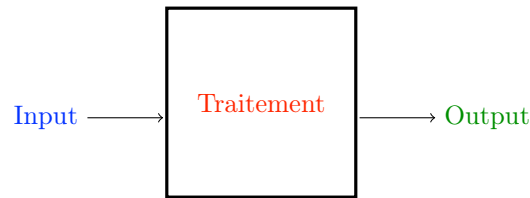


FIGURE 3 – Les trois étapes d’un programme.

La Fig. 3 présente les trois étapes d’un programme classique :

1. lire les données en entrées (**Input**) ;
2. effectuer les calculs (**Traitement**) ;
3. écrire les données/résultats en sorties (**Output**) ;

#### Alerte : Comportement

Attention, un programme doit **toujours** avoir un output (un programme qui ne produit rien n’a aucune utilité). Par contre, il est toujours possible qu’un possible programme n’ait pas besoin d’input. Dans la suite du cours, nous verrons comment rendre un peu plus spécifique ce concept de programme, d’abord grâce à la Définition d’un problème (Chapitre 3) et, ensuite, les spécifications (Chapitre 6).

Ce fonctionnement implique la présence d’*entrées/sorties* dans un programme. Il s’agit pour le programme d’aller chercher de l’information (entrées) et de produire le résultat (sorties) à l’extérieur. Dans le cadre du cours INFO0946, les entrées consisteront en des lectures au clavier (ou dans un fichier – voir Chapitre 5) et les sorties en des écritures sur l’écran (ou dans un fichier – voir Chapitre 5).

#### 2.3.3.1 Lecture au Clavier


Les données peuvent être saisies sur l’*entrée standard* (i.e., le clavier).

Pour lire des données au clavier, il suffit

1. d’inclure la librairie `stdio.h` qui contient toutes les routines pour les entrées/sorties ;
2. d’utiliser l’instruction `scanf(format, variable);` où
  - **message** est une chaîne de caractères entre guillemets contenant, **uniquement**, un **formatage** ;
  - **variable** est une variable, précédée de l’opérateur `&` (référencement), qui va contenir la valeur lue au clavier (il y a possibilité de lire plusieurs variables d’un seul coup, elles sont alors séparées par une virgule).

Par exemple :

```
1 #include <stdio.h> //Etape 1
2
3 int main(){
4     float f;
5     int x;
6
7     scanf("%d", &x); //Etape 2
8     scanf("%d%f", &x, &f);
9 }//fin programme
```

Attention, l'instruction `scanf(...)`; est bloquante. Cela signifie que le programme est bloqué tant que l'utilisateur n'a pas entré les valeurs demandées au clavier et n'a pas tapé sur la touche .

Si jamais l'utilisateur entre une valeur différente du type indiqué dans le formatage, ça ne sera pas un problème. La valeur entrée par l'utilisateur sera interprétée en fonction du formatage (ce qui pourrait, potentiellement, mener à une erreur d'exécution).

### 2.3.3.2 Écriture à l'Écran

Les résultats peuvent être affichés sur la *sortie standard* (i.e., l'écran).

Pour écrire un message à l'écran, il suffit

1. d'inclure la librairie `stdio.h` qui contient toutes les routines pour les entrées/sorties;
2. d'utiliser l'instruction `printf(message)`; , où `message` est une chaîne de caractères entre guillemets.

Par exemple :

```
1 #include <stdio.h> //Etape 1
2
3 int main(){
4     printf("Bonjour!"); //Etape 2
5 }//fin programme
```

Si on veut écrire un message à l'écran qui comprend, aussi, la valeur d'une (ou plusieurs) variable(s), il suffit

1. d'inclure la librairie `stdio.h` qui contient toutes les routines pour les entrées/sorties;
2. d'utiliser l'instruction `printf(message, expression)`; , où
  - `message` est une chaîne de caractères entre guillemets avec un **formatage**;
  - `expression` est la liste des expressions (voir Chapitre 1) qu'on veut afficher (les différentes expressions sont séparées par une virgule).

Par exemple :

```
1 #include <stdio.h> //Etape 1
2
3 int main(){
4     int x = 5;
5     printf("Vive le chiffre %d", x); //Etape 2
6 }//fin programme
```

### 2.3.3.3 Formatage

Les entrées/sorties nécessite un *formatage* en fonction du type primitif (voir le Chapitre 1). Le Tableau 1 présente la liste des formatages possibles en C.

### Suite de l'Exercice

À vous! Déterminez ce que calcule le code et passez à la Sec. 2.3.5.

Si vous ne voyez pas comment démarrer, voyez la Sec. 2.3.4 pour une astuce.

Type Primitif	Formatage	Exemple
int	%d	int i=-10; printf("%d", i);
unsigned int	%u	unsigned int ui = 10; printf("%u", ui);
short	%hd	short si=1; printf("%hd", si);
unsigned short	%hu	unsigned short us = 10; printf("%hu", us);
long	%ld	long int li=1000; printf("%ld",li);
unsigned long	%lu	unsigned long ul = 10; printf("%lu", ul);
char	%c	char car='a'; printf("%c", car);
float	%f	float x=2.0/3.0; printf("%f", x);
double	%lf	double x=2.0/3.0; printf("%lf", x);

TABLE 1 – Formatage

### 2.3.4 Indice

Pour résoudre ce problème, il est souhaitable de travailler sur base d'exemples numériques et voir, à chaque étape du code, l'évolution des valeurs contenues dans chacune des variables du code. Idéalement, il faut travailler avec des valeurs générales et d'autres plus particulières.

Dans le **code** fourni, les valeurs initiales des variables `m` et `n` sont lues au clavier. C'est donc avec elles qu'il faut jouer.

Une fois cette étape réalisée, il faut essayer de trouver une relation entre les données obtenues à la fin de la boucle et exprimer cette relation de la manière la plus générale qui soit (application du concept de *transcodage* – ©F. Bastin).

### Suite de l'Exercice

À vous ! Déterminez ce que calcule le code et passez à la Sec. **2.3.5**.

### 2.3.5 Mise en Commun de la Compréhension de la Boucle

Le bout de code à analyser est le suivant :

```
1 unsigned int x, k, n;  
2  
3 scanf("%u %u", &k, &n);  
4  
5 for(x = k; x >= n; x -= n);  
6  
7 printf("%d\n", x);
```

Passons-le en revue, ligne par ligne.

1.  
1 unsigned int x, k, n;

Toute variable doit être déclarée avant son utilisation ; c'est précisément le but de cette instruction. Il s'agit donc d'une *déclaration de variables*. Cette instruction doit définir le **type** d'une variable et son **identificateur**. Ici, trois variables de type **unsigned int** (naturel, équivalent à  $\mathbb{N}$ ) sont déclarées : **x**, **k** et **n**.

Remarquons que celles-ci ne sont pas initialisées. Ce n'est en aucun cas un problème ; rappelez-vous, l'initialisation d'une variable est optionnelle lors de sa déclaration. Cependant, chaque variable doit impérativement être initialisée avant son premier usage.

À ce stade nous disposons donc simplement de trois naturels non initialisés et manipulables via leur identificateur.

2.  
3 scanf("%u %u", &k, &n);

L'instruction utilisée ici est **scanf**. Celle-ci est incluse dans la librairie **stdio.h** et permet de saisir les données sur l'entrée standard (clavier).

La chaîne de caractères "%u %u" indique le **formatage**. En particulier, %u nous indique que les données à saisir sont de type **unsigned int**.

On retrouve également les variables **k** et **n** précédées de l'opérateur **&** (opérateur de référencement). **k** va contenir la valeur de la première donnée saisie au clavier et **n** la seconde.

Finalement, cette instruction nous a permis d'initialiser les variables **k** et **n**.

3.  
5 for(x = k; x >= n; x -= n);

L'instruction **for** permet de spécifier, de manière succincte, plusieurs opérations à exécuter. Décomposons ces opérations.

- (a) **x = k** est l'instruction qui sera exécutée avant d'entrer dans la boucle pour la première fois. Elle utilise l'opérateur d'**affectation** ; cette instruction affecte la valeur de **k** dans **x**. Il s'agit donc de l'*initialisation* de **x**. Dès lors, **x** apparaît comme le compteur de la boucle.
- (b) **x >= n** est l'expression utilisée afin décider s'il faut continuer (ou non) à itérer. Il s'agit donc du *Gardien de Boucle*. L'expression contient l'opérateur de *comparaison* **>=**. Cette expression prendra la valeur booléenne vrai (1) seulement si la valeur de **x** est supérieure ou égale à la valeur de **n**. Dans ce cas, on poursuit l'exécution des instructions de la boucle. Dans le cas contraire (**x** est strictement plus petit que **n**), l'expression sera évaluée à faux (0) et les instructions qui suivent la boucle **for** seront exécutées.
- (c) **x -= n** est l'instruction qui sera effectuée entre deux itérations (i.e., après le Corps de la Boucle et avant l'évaluation du Gardien de Boucle). Dans notre cas, il n'y a pas de Corps de la Boucle. En effet, si on regarde bien la ligne de code, le Corps de la Boucle contient l'*instruction vide* (i.e., un simple **;**), soit l'instruction qui ne fait rien. Dès lors, l'instruction **x -= n** sera exécutée directement après l'évaluation du Gardien de Boucle. On y retrouve l'opérateur **-=** qui est un

sucré syntaxique. Rappelons que cette expression est équivalente à  $x = x - n$ . On soustrait donc  $n$  de  $x$ . Le résultat obtenu sera la nouvelle valeur de  $x$ . Cette instruction décrémente le compteur ( $x$ ) à chaque itération.

Finalement, l'instruction `for` soustrait  $n$  de  $x$  tant que le résultat de cette soustraction est supérieur ou égal à  $n$ . Quant aux variables  $k$  et  $n$ , elles restent inchangées et conservent les valeurs initialement entrées par l'utilisateur au clavier.

#### Alerte : Mauvaise réponse

Se contenter, comme réponse, de la phrase :

On soustrait  $n$  de  $x$  tant que le résultat de cette soustraction est supérieur ou égal à  $n$ .

n'est pas acceptable. Ceci n'est jamais qu'une interprétation littérale du code (i.e., le comportement de la boucle `for` est réécrit en langage naturel). Cette réponse ne donne aucune information sur ce qui est calculé réellement et mène, naturellement, à une note nulle.

4.  
7 `printf("%u\n", x);`

L'instruction utilisée ici est `printf`. Celle-ci est également incluse dans la librairie `stdio.h`. Elle permet d'écrire un message sur la sortie standard (écran).

La chaîne de caractères `"%d\n"` est le message à afficher ainsi que le **formatage** de l'expression que l'on souhaite afficher. En particulier, `%u` nous indique que l'expression est de type `unsigned int` et `'\n'` est un caractère spécial permettant le retour à la ligne.

Cette chaîne de caractères est suivie par l'expression `x`; c'est donc la valeur de `x` qui sera affichée à l'écran suivie d'un retour à la ligne.

Finalement, cette instruction nous a permis d'afficher ce que le bout de code a calculé (`x`).

Essayons à présent de trouver une relation entre les données introduites et le résultat obtenu. Pour ce faire, détaillons les valeurs lors de l'exécution du code avec des entrées différentes.

Les tableaux ci-dessous donnent les valeurs de `x`, `k` et `n`, étape par étape. Le code couleur pour la colonne `x` correspond au code couleur du code de l'instruction `for`. On effectue trois tests : (i) pour une valeur générale de `x` (tableau de droite), (ii) pour une valeur générale plus petite pour `x` (tableau haut gauche) et, enfin, pour une valeur très particulière de `x` (tableau bas gauche). L'itération 0 correspond au code avant la boucle `for` et comprend donc le `scanf()` et l'instruction exécutée une seule fois avant la toute première évaluation du Gardien de Boucle.

Itération	x	k	n
0	??	7	3
	7	7	3
1	4	7	3
2	1	7	3

Itération	x	k	n
0	??	2	3
	2	2	3

Itération	x	k	n
0	??	20	3
	20	20	3
1	17	20	3
2	14	20	3
3	11	20	3
4	8	20	3
5	5	20	3
6	2	20	3

Une fois ces valeurs mises en évidence, il faut essayer de trouver une relation entre les différentes variables. C'est l'objectif du tableau 2. Sachant que `x` a été initialisé à la valeur de `k` et que `n` est répétitivement soustrait de `x`, on constate que la valeur finale de `x` est le reste de la division de `k` par `n`.

On peut donc conclure que le bout de code fourni prend en entrée deux naturels et affiche, à l'écran, le reste de la division du premier par le second. Dit autrement, ce code calcule `k modulo n`.

Itérations	x	k	n	Relation
2	1	7	3	$7 = 3 \times 2 + 1$
0	2	2	3	$2 = 3 \times 0 + 2$
6	2	20	3	$20 = 3 \times 6 + 2$

TABLE 2 – Relations entre les variables.



## 2.4 Réécriture avec une Boucle `while`

- Si vous voyez directement comment procéder, voyez la suite [2.4.3](#)
- Si vous êtes un peu perdu, voyez le rappel sur le fonctionnement de la boucle `while` [2.4.1](#)
- Si vous ne voyez pas comment démarrer, voyez l'indice [2.4.2](#)

### 2.4.1 Rappel sur la Boucle while

L'instruction **while** est une **structure itérative** qui permet :

1. de répéter une instruction (ou un bloc d'instructions);
2. tant qu'une condition (sous la forme d'une *expression booléenne*) est satisfaite;
3. la condition est évaluée **avant** chaque itération.

La syntaxe<sup>6</sup> de l'instruction **while** est la suivante :

```
1 while(expression){  
2   instructions;  
3 }
```

L'instruction commence toujours par le mot-clé **while** (*tant que*, en français). Dans un langage de programmation, un mot-clé est un mot réservé qui a une signification bien particulière dans le langage. Cela signifie donc que vous ne pouvez pas déclarer une variable (par exemple) dont l'identificateur est **while**. Cela provoquerait nécessairement une erreur de compilation.

Le mot-clé **while** est suivi par une **expression** entre parenthèses (elles sont obligatoires). Cette expression est évaluée de manière booléenne et indique si oui ou non on peut rentrer dans la boucle (i.e., exécuter le **bloc d'instructions** associé à la boucle). Après les parenthèses, nous trouvons un **bloc d'instructions**, entouré d'accolades (elles ne sont pas obligatoires si le **bloc** contient une seule instruction). Les différentes instructions du **bloc** sont séparées par un **;**.

L'instruction **while** est comprise, par le compilateur, comme une seule instruction (quelque soit la quantité d'instructions du **bloc**). Il n'est néanmoins pas nécessaire de la terminer par un **;** (les accolades jouent ce rôle).

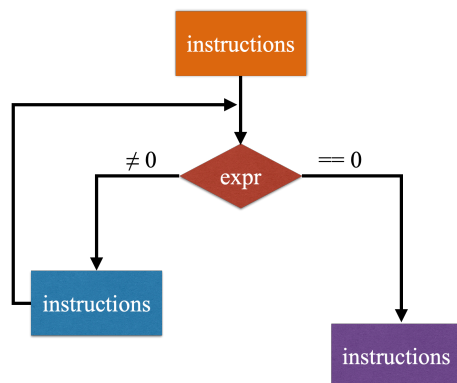


FIGURE 4 – Sémantique de l'instruction **while**. Un rectangle représente une suite d'instructions – éventuellement sous la forme d'un bloc. Un losange représente expression évaluée de manière booléenne (i.e., une condition).

La sémantique<sup>7</sup> de l'instruction **while** est donnée par la Fig. 4. Dans cette figure, l'instruction **while** reprend l'**expression** et le **bloc d'instructions**. Le **bloc orange** correspond aux instructions avant la boucle. Le

6. La syntaxe fait référence à l'ensemble des règles grammaticales d'une langue, i.e., comment les différents éléments doivent être assemblés pour former une phrase. Appliquée à la programmation (le C est un langage), c'est l'ensemble des règles permettant de construire des instructions correctes. La syntaxe des langages de programmation sera étudiée en détails dans le cours INFO0085.

7. La sémantique fait référence à l'étude du sens, de la signification des signes, dans le langage. Appliquée à la programmation, la sémantique consiste à donner une signification mathématique au programme. La sémantique permettra, entre autre, de faire des raisonnements formels – donc mathématiques – sur le programme. Ceci est intéressant pour les programmes requérant un

**bloc mauve** correspond aux instructions après la boucle. Les flèches indiquent le sens du flux du programme, i.e., l'ordre d'exécution des différentes instructions et d'évaluation des expressions.

La Fig. 4 indique que lorsque le **bloc orange** se termine, on évalue d'abord l'**expression**. L'instruction à exécuter ensuite va dépendre du résultat (booléen) de cette évaluation. Si c'est vrai ( $\neq 0$ ), alors on entre dans la boucle et on exécute le **corps** de la boucle. Après la dernière instruction du boucle, on réévalue l'**expression**. Quand l'évaluation devient fausse ( $= 0$ ), alors on exécute les **instructions** qui suivent la boucle.

#### Alerte : Vocabulaire

Le vocabulaire associé à une boucle est extrêmement important, dans le cadre de ce chapitre (bien entendu) mais, plus largement, dans le cadre des cours INFO0946 et INFO0947. Voici les éléments importants :

*Gardien de Boucle.* Il s'agit de l'expression qui est évaluée avant chaque entrée dans le Corps de la Boucle (**losange rouge** sur la Fig. 4). Le résultat de l'évaluation de l'expression est de type booléen. Si l'évaluation est vraie ( $\neq 0$ ), alors le Corps de la Boucle est exécuté. Si l'évaluation est fausse ( $= 0$ ), alors on sort de la boucle et on exécute l'instruction (ou la suite d'instructions) après la boucle (**rectangle mauve** sur la Fig. 4). Le Gardien de Boucle peut être une expression simple ou une expression plus complexe construite à l'aide des opérateurs booléens.

*Critère d'Arrêt.* Il s'agit de la condition qui est rencontrée (i.e., vraie) quand on sort de la boucle. Il y a donc un lien extrêmement fort entre Gardien de Boucle et Critère d'Arrêt : le Critère d'Arrêt est **la négation** du Gardien de Boucle. Le Critère d'Arrêt est un concept important. Toute boucle doit, un jour, rencontrer son Critère d'Arrêt. Sinon, la boucle ne se termine pas. On parle alors de *boucle infinie*. C'est l'une des erreurs les plus fréquentes et les plus mortelles.

*Corps de la Boucle.* Il s'agit de l'instruction ou du bloc d'instruction qui est exécuté chaque fois que le Gardien de Boucle est vrai (**rectangle bleu** sur la Fig. 4). Ce bloc est donc répété autant de fois que nécessaire, jusqu'à ce que le Critère d'Arrêt soit atteint.

#### Suite de l'Exercice

À vous ! réécrivez le code à l'aide d'une boucle **while** et passez à la Sec. 2.4.3.

Si vous ne voyez pas comment démarrer, voyez la Sec. 2.4.2 pour une astuce.

---

niveau qualité très important (e.g., l'informatique dans l'aéronautique, dans les véhicules, ...). La sémantique des langages de programmation sera étudiée en détails dans le cours **INFO0085**.

## 2.4.2 Indice

Il faut repartir de la **sémantique** de la boucle `while` et de la **sémantique** de la boucle `for`. Il faut ensuite essayer de faire un “mapping” entre les deux sémantiques. Normalement, la transcription devrait être assez évidente.

### Suite de l’Exercice

À vous ! réécrivez le code à l’aide d’une boucle `while` et passez à la Sec. **2.4.3**.

### 2.4.3 Mise en Commun de la Réécriture

La boucle `for` suivante est à réécrire en boucle `while` :

```
1 for(x = k; x >= n; x -= n);
```

Les boucles `for` et `while` fonctionnent de la même manière ; seul le positionnement des opérations change pour passer de l'une à l'autre. Repositionnons donc les différentes opérations sans les modifier.

La première instruction à effectuer avant d'entrer dans la boucle est `x = k`. Celle-ci sert à initialiser le compteur, soit `x`. Cette opération doit donc être placée juste avant la boucle `while`, de façon à ce qu'elle soit exécutée avant la toute première évaluation du Gardien de Boucle.

```
2 x = k;
3 while(...){
4     ...
5 }
```

On retrouve ensuite le Gardien de Boucle, soit l'expression `x >= n`. Celui-ci doit être évalué à chaque tentative d'entrée dans la boucle. Dans une boucle `while`, il est placé dans les parenthèses juste après le mot-clé `while`.

```
6 x = k;
7 while(x >= n){
8     ...
9 }
```

Après l'évaluation du Gardien de Boucle et si ce dernier est vrai, on exécute le Corps de la Boucle. Dans notre boucle `for`, aucune instruction ne doit être effectuée ; on passe donc directement à l'instruction `x -= n` qui, dans une boucle `for`, est exécutée juste avant une nouvelle évaluation du Gardien de Boucle. Cette instruction sert à décrémenter le compteur de la boucle (soit `x`) et doit être effectuée à chaque tour de boucle. Elle sera donc placée dans le Corps de la Boucle du `while`.

```
10 x = k;
11 while(x >= n){
12     x -= n;
13 }
```

Finalement, le bout de code initial est équivalent au suivant :

```
1 unsigned int x, k, n;
2
3 scanf("%u%u", &k, &n);
4
5 x = k;
6 while(x >= n){
7     x -= n;
8 }
9
10 printf("%u\n", x);
```

## 2.5 Réécriture avec une Boucle `do ... while`

- Si vous voyez directement comment procéder, voyez la suite [2.5.3](#)
- Si vous êtes un peu perdu, voyez le rappel sur le fonctionnement de la boucle `do ... while` [2.5.1](#)
- Si vous ne voyez pas comment démarrer, voyez l'indice [2.5.2](#)

### 2.5.1 Rappel sur la Boucle `do ... while`

L'instruction `do ... while` est une **structure itérative** qui permet :

1. de répéter une instruction (ou un bloc d'instructions) au moins une fois ;
2. tant qu'une condition (sous la forme d'une *expression booléenne*) est satisfaite ;
3. la condition est évaluée **après** chaque itération.

La syntaxe<sup>8</sup> de l'instruction `do ... while` est la suivante :

```
1 do{  
2   instructions;  
3 }while(expression);
```

L'instruction contient deux mots-clés : `do` et `while` (*faire ... tant que*, en français). Dans un langage de programmation, un mot-clé est un mot réservé qui a une signification bien particulière dans le langage. Cela signifie donc que vous ne pouvez pas déclarer une variable (par exemple) dont l'identificateur est `do` ou `while`. Cela provoquerait nécessairement une erreur de compilation.

Le mot-clé `do` est suivi par un **bloc d'instructions** correspondant au bloc à répéter à chaque itération. Ensuite, vient le mot-clé `while` qui contient, entre parenthèses, une **expression**. Cette expression est évaluée de manière booléenne et indique si oui ou non on peut rentrer dans la boucle (i.e., exécuter le **bloc d'instructions** associé à la boucle).

L'instruction `do ... while` est comprise, par le compilateur, comme une seule instruction (quelque soit la quantité d'instructions du **bloc**). Contrairement au `for` ou au `while`, il est nécessaire de terminer l'instruction `do ... while` par un `;` après la parenthèse fermante du `while`. L'objectif du `;` est de permettre au compilateur de distinguer la fin d'un `do ... while` par rapport à un `while`.

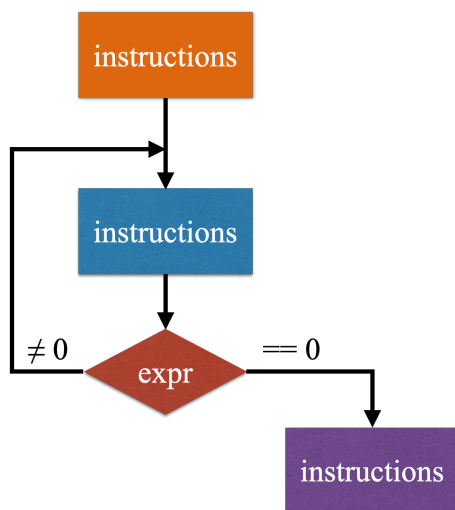


FIGURE 5 – Sémantique de l'instruction `do ... while`. Un rectangle représente une suite d'instructions – éventuellement sous la forme d'un bloc. Un losange représente expression évaluée de manière booléenne (i.e., une condition).

8. La syntaxe fait référence à l'ensemble des règles grammaticales d'une langue, i.e., comment les différents éléments doivent être assemblés pour former une phrase. Appliquée à la programmation (le C est un langage), c'est l'ensemble des règles permettant de construire des instructions correctes. La syntaxe des langages de programmation sera étudiée en détails dans le cours [INFO0085](#).

La sémantique<sup>9</sup> de l'instruction `do ... while` est donnée par la Fig. 5. Dans cette figure, l'instruction `do ... while` reprend l'expression et le bloc d'instructions. Le bloc orange correspond aux instructions avant la boucle. Le bloc mauve correspond aux instructions après la boucle. Les flèches indiquent le sens du flux du programme, i.e., l'ordre d'exécution des différentes instructions et d'évaluation des expressions.

La Fig. 5 indique que le bloc d'instructions est toujours exécuté au moins une fois car l'évaluation de l'expression se fait après. Il s'agit là de la principale différence entre le `do ... while` et les deux autres formes de boucle (`for` et `while`) où l'expression est toujours évaluée avant d'exécuter (éventuellement) le bloc associé.

Si l'expression est vraie ( $\neq 0$ ), alors on exécute encore une fois le bloc d'instructions. Sinon ( $= 0$ ), on passe aux instructions qui suivent la boucle.

#### Alerte : Vocabulaire

Le vocabulaire associé à une boucle est extrêmement important, dans le cadre de ce chapitre (bien entendu) mais, plus largement, dans le cadre des cours INFO0946 et INFO0947. Voici les éléments importants :

*Gardien de Boucle.* Il s'agit de l'expression qui est évaluée avant chaque entrée dans le Corps de la Boucle (losange rouge sur la Fig. 5). Le résultat de l'évaluation de l'expression est de type booléen. Si l'évaluation est vraie ( $\neq 0$ ), alors le Corps de la Boucle est exécuté. Si l'évaluation est fausse ( $= 0$ ), alors on sort de la boucle et on exécute l'instruction (ou la suite d'instructions) après la boucle (rectangle mauve sur la Fig. 5). Le Gardien de Boucle peut être une expression simple ou une expression plus complexe construite à l'aide des opérateurs booléens.

*Critère d'Arrêt.* Il s'agit de la condition qui est rencontrée (i.e., vraie) quand on sort de la boucle. Il y a donc un lien extrêmement fort entre Gardien de Boucle et Critère d'Arrêt : le Critère d'Arrêt est la **négation** du Gardien de Boucle. Le Critère d'Arrêt est un concept important. Toute boucle doit, un jour, rencontrer son Critère d'Arrêt. Sinon, la boucle ne se termine pas. On parle alors de *boucle infinie*. C'est l'une des erreurs les plus fréquentes et les plus mortelles.

*Corps de la Boucle.* Il s'agit de l'instruction ou du bloc d'instruction qui est exécuté chaque fois que le Gardien de Boucle est vrai (rectangle bleu sur la Fig. 5). Ce bloc est donc répété autant de fois que nécessaire, jusqu'à ce que le Critère d'Arrêt soit atteint.

#### Suite de l'Exercice

À vous ! réécrivez le code à l'aide d'une boucle `do ... while` et passez à la Sec. 2.5.3.

Si vous ne voyez pas comment démarrer, voyez la Sec. 2.5.2 pour une astuce.

9. La sémantique fait référence à l'étude du sens, de la signification des signes, dans le langage. Appliquée à la programmation, la sémantique consiste à donner une signification mathématique au programme. La sémantique permettra, entre autre, de faire des raisonnements formels – donc mathématiques – sur le programme. Ceci est intéressant pour les programmes requérant un niveau qualité très important (e.g., l'informatique dans l'aéronautique, dans les véhicules, ...). La sémantique des langages de programmation sera étudiée en détails dans le cours INFO0085.



## 2.5.2 Indice

La **sémantique** de la boucle `do ... while` indique que le Corps de la Boucle est toujours exécuté au moins une fois avant évaluation du Gardien de Boucle.

En ayant ce point en mémoire, il va falloir rédiger la boucle `do ... while` en évitant d'itérer une fois de trop.

### Suite de l'Exercice

À vous ! réécrivez le code à l'aide d'une boucle `do ... while` et passez à la Sec. **2.5.3**.

### 2.5.3 Mise en Commun de la Réécriture

La boucle `for` suivante est à réécrire en boucle `do ... while` :

```
14 for(x = k; x >= n; x -= n);
```

Commençons par réorganiser les différentes opérations afin de respecter la syntaxe des boucles `do ... while`.

La première instruction à effectuer avant d'entrer dans la boucle est `x = k`. Celle-ci sert à initialiser le compteur ; dans ce cas il s'agit de `x`. Cette opération doit donc être placée avant la boucle `do ... while`.

```
15 x = k;
16 do{
17     ...
18 }while (...);
```

On retrouve ensuite le Gardien de Boucle qui est l'expression `x >= n`. Celui-ci sera évalué à chaque tentative d'entrée dans la boucle après la première itération. Dans une boucle `do ... while`, il est placé dans les parenthèses juste après le mot-clé `while`.

```
19 x = k;
20 do{
21     ...
22 }while(x >= n);
```

Le Corps de la Boucle doit contenir les instructions à effectuer à chaque itération. Le Corps de la Boucle `for` étant vide, seule l'instruction `x -= n` est à exécuter. Elle sera donc placée dans le Corps de la Boucle `do ... while`.

```
23 x = k;
24 do{
25     x -= n;
26 }while(x >= n);
```

Penchons-nous maintenant sur la sémantique de notre nouveau bout de code. Une boucle `do ... while` commence toujours par exécuter une première fois le Corps de la Boucle sans vérifier le Gardien de Boucle. Pour notre programme, ceci est problématique dans le cas où le Gardien de Boucle est faux dès le départ (pensez au **cas particulier** qui a été discuté lors de la compréhension du code initial). Il faut donc s'assurer que le Gardien de Boucle sera vrai au moins une fois afin de neutraliser cette première itération. `x` doit donc au moins être supérieur ou égal à `n` lors de la première itération. Une manière de procéder est simplement d'ajouter `n` à `x` lors de l'initialisation.

```
27 x = k + n;
28 do{
29     x -= n;
30 }while(x >= n);
```

Finalement, le bout de code initial est équivalent au suivant :

```
1 unsigned int x, k, n;
2
3 scanf("%u%u", &k, &n);
4
5 x = k + n;
6 do{
7     x -= n;
8 }while(x >= n);
9
10 printf("%u\n", x);
```