

UNIVERSITÉ DE LIÈGE

INFO0946

INTRODUCTION À LA PROGRAMMATION

---

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

## Fichiers Séquentiels

---

Benoit DONNET

Simon LIÉNARDY

23 février 2021



# Préambule

## Exercices

Dans ce GAMECODE, nous vous proposons de suivre pas à pas la résolution d'un exercice portant sur les fichiers.

**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.

## 3.1 Rappels sur les Fichiers Séquentiels

### 3.1.1 Généralités

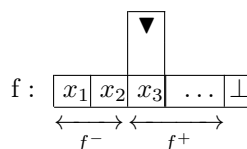


FIGURE 1 – Schéma d'un fichier séquentiel

On représente le fichier séquentiel en dessinant le ruban qui contient les données ainsi que la *tête de lecture/écriture*, symbolisée par le ▼. Un exemple de fichier est illustré par la Fig. 1.

En quoi cela diffère-t-il des tableaux ? On n'a accès qu'à un seul élément : celui sous la tête de lecture/écriture<sup>2</sup> (*élément courant*). Il faut de toute façon le lire pour avancer à l'élément suivant. On ne peut pas se déplacer de manière aléatoire dans le fichier (les ordinateurs en sont capables, évidemment, mais on n'envisage pas cette possibilité aujourd'hui). On peut tester si on est à la fin du fichier grâce à la valeur spéciale  $\perp$ <sup>3</sup> qui représente le symbole EOF (*End Of File*).

### 3.1.2 Notations sur les Fichiers

Faites en sorte de bien vous **familiariser** avec ces notations ! Si vous ne les comprenez pas, relisez les slides (Chap. 3, Slides 12 → 19).

Notion	Notation
Fichier $f$	$f = \langle x_1, x_2, \dots, x_n \rangle$
Ens. de valeur	$x_i \in V$
Fichier vide	$f = \langle \rangle$
Restriction de $f$ à l'intervalle $[i \dots j]$	$f_i^j$ , vide si $j > i$
Concaténation de fichier	$f_1    f_2$
Minorant ( $\forall i, 1 \leq i \leq n, a < x_i$ )	$a < f$
Contient valeur ( $\exists i, 1 \leq i \leq n, a = x_i$ )	$a \in f$
Ne contient pas valeur ( $\forall i, 1 \leq i \leq n, a \neq x_i$ )	$a \notin f$
Égalité des élém. à $a$ ( $\forall i, 1 \leq i \leq n, a = x_i$ )	$a = f$
¬ Égalité des élém. à $a$ ( $\exists i, 1 \leq i \leq n, a \neq x_i$ )	$a \neq f$
Fin de fichier	$\perp$
Fichier déjà parcouru	$f^-$
Fichier encore à parcourir (compris <b>head</b> <sup>4</sup> )	$f^+$

TABLE 1 – Résumé des notations pour les fichiers séquentiels.

### 3.1.3 Inventaire des Modules Disponibles

#### 3.1.3.1 Ouverture (fopen())

2. Si vous avez connu les VHS (cassettes vidéos), vous avez une idée pratique de ce dont on parle.

3. À prononcer « bottom »

4. **head** fait référence à l'élément courant (i.e., sous la tête de lecture/écriture).

```

1 /*
2  * PRÉCONDITION : mode ∈ {"r", "w"} ∧ chemin valide
3  * POSTCONDITION : mode0 = "r" ⇒ (¬eof(f) ∧ f- = <> ∧ f+ = f) ∨ mode0 = "w" ⇒ (eof(f) ∧ f = <>)
4  */
5 FILE fopen(char *chemin, char *mode);

```

Si le mode est "r" pour **R**ead, le fichier spécifié par le chemin est ouvert en lecture et la tête de lecture est positionnée sur le premier élément. Si le mode est "w" pour **W**rite, le fichier est ouvert en écriture et il est vide. S'il n'existe pas, il est créé avec le nom `chemin`.

**REMARQUE IMPORTANTE** Ici, `FILE` est une variable de type fichier. En temps normal, ce serait un type opaque (c'est d'ailleurs le cas dans `stdio.h` et il faudrait passer un pointeur, `FILE *`). **On ne s'encombre pas de ces considérations dans ce GAMECODE.** On manipule `FILE` *comme si* c'était un type primitif. Consultez les slides (Chapitre 3, Slides 24, 29, 31, 36, 39) pour connaître la traduction du pseudo-langage étudié ici et le C.

### 3.1.3.2 Fin de Fichier (eof())

```

1 /*
2  * PRÉCONDITION : /
3  * POSTCONDITION : eof(f) ⇔ f+ = < >
4  */
5 bool eof(FILE f);

```

Le prédicat est vrai si et seulement si on a atteint la fin du fichier.

### 3.1.3.3 Lecture (read())

```

1 /*
2  * PRÉCONDITION : f- = f1i-1 ∧ f+ = fin ∧ ¬eof(f)
3  * POSTCONDITION : val = xi ∧ f- = f1i ∧ f+ = fi+1n
4  */
5 void read(FILE f, type val);

```

L'élément qui est sous la tête de lecture (i.e., l'élément courant) est lu et placé dans `val`. La tête de lecture progresse ensuite vers l'élément suivant.<sup>5</sup>

### 3.1.3.4 Ecriture (write())

```

1 /*
2  * PRÉCONDITION : f = < x1, x2, ..., xi-1 > ∧ val = xi ∧ eof(f)
3  * POSTCONDITION : f = < x1, x2, ..., xi-1xi > ∧ val = val0 ∧ eof(f)
4  */
5 void write(FILE f, type val);

```

La valeur `val` est écrite en fin de fichier. La tête d'écriture est toujours positionnée en fin de fichier suite à l'appel.

### 3.1.3.5 Fermeture du Fichier (fclose())

```

1 void fclose(FILE f);

```

Ferme le fichier `f`.

---

5. Par rapport aux slides, la référence à `EOF` dans la `POSTCONDITION` a été retirée parce que c'est vrai dans tous les cas.

### 3.1.4 Remarques Finales

- Quels fichiers doit-on fermer ?  
**R** : Tous ceux qui ont été ouverts avec `fopen()`.
- Même ceux ouverts en lecture ?  
**R** : Oui, c'est une habitude à prendre !
- Même si ça ne risque rien ?  
**R** : Oui, c'est une bonne habitude à prendre !
- Quel est le risque à ne pas fermer un fichier ?  
**R** : Les fonctions d'interactions avec le système de fichier, style `fprintf()` n'écrivent pas tout de suite dans le fichier lorsqu'elles sont invoquées mais écrivent d'abord dans une mémoire tampon (*buffer* en anglais). La fermeture du fichier force l'écriture du tampon dans le fichier. Un oubli de fermeture de fichier pourrait ne pas écrire le tampon et il pourrait s'ensuivre une perte de données.

## 3.2 Énoncé

Écrivez une fonction qui délivre la somme des éléments de rang pair d'un fichier de nombres entiers.

C'est-à-dire :  $x_2 + x_4 + x_6 \dots$

### 3.2.1 Méthode de Résolution

On va procéder à la résolution de ce problème en suivant pas à pas l'approche constructive :

1. Définir de nouvelle-s notation-s (Sec. 3.3) ;
2. Spécifier le problème complètement (Sec. 3.4) ;
3. Chercher un Invariant de Boucle (Sec. 3.5) ;
4. Construire le programme complètement (Sec. 3.6) ;
5. Rédiger le programme final (Sec. 3.7).

### 3.3 Formalisation du Problème

La première étape dans la résolution du problème nécessite sa formalisation, ce qui passe par l'introduction de nouvelles notations.

- Si vous voyez de quoi on parle, rendez-vous à la Section [3.3.3](#)
- Si vous ne savez comment introduire une notation, voyez la Section [3.3.1](#)
- Si vous ne voyez pas comment faire, reportez-vous à l'indice [3.3.2](#)

### 3.3.1 Rappels sur la Formalisation d'un Problème

#### 3.3.1.1 Généralités

Voici la forme que doit prendre une notation :

$\text{NomPredicat}(\text{Liste de parametres}) \equiv \text{Détail du predicat}$

**NomPredicat** est le nom du prédicat<sup>a</sup>. Il faut souvent trouver un nom en rapport avec ce que l'on fait. Mathématiquement parlant, cela ne changera rien de proposer un nom farfelu. Par exemple, on pourrait appeler le prédicat « Gims ». Cependant, pour des raisons de lisibilité, il est toujours préférable de trouver un nom en rapport avec la définition et le problème qu'on cherche à formaliser.

$\equiv$  est le symbole mathématique signifiant « identique à ». Il sert à introduire le détail du prédicat.

**Liste de paramètres** C'est une liste de symboles qui pourront être utilisés dans la définition du prédicat. Comment cela fonctionne-t-il ? Le prédicat peut être utilisé en écrivant son nom ainsi qu'une valeur particulière de paramètre. On remplace cette valeur dans la définition du prédicat et on regarde si c'est vrai ou faux.

**Détail du prédicat** C'est une combinaison de symboles logiques qui fait intervenir, le plus souvent, les paramètres. Si nécessaire, il est aussi possible d'utiliser des **quantificateurs**.

<sup>a</sup>. Pour rappel, un prédicat est une fonction mathématique produisant une valeur booléenne.

Exemple :

$\text{Pair}(X) \equiv X \% 2 = 0$

Le nom du prédicat est « Pair » et prend un argument.  $\text{Pair}(4)$  remplace  $4 \% 2 = 0$  qui est vrai et  $\text{Pair}(5)$  signifie  $5 \% 2 = 0$  qui est faux. Le choix du nom du prédicat est en référence à sa signification : il est vrai si  $X$  est pair.

▷ **Exercice** Si vous avez compris, vous pouvez définir facilement le prédicat  $\text{Impair}(X)$ .

#### 3.3.1.2 Opérateurs Logiques

Les opérateurs logiques usuels sont les suivants :

*True* correspond la propriété “vrai” ;

*False* correspond à la propriété “faux” ;

$\neg$  est l'opérateur de négation. Dès lors,  $\neg p$  signifie “non  $p$ ” ;

$\wedge$  est l'opérateur de conjonction. Dès lors,  $p \wedge q$  signifie “ $p$  et  $q$ ” ;

$\vee$  est l'opérateur de disjonction. Dès lors,  $p \vee q$  signifie “ $p$  ou  $q$ ” ;

$\implies$  est l'opérateur d'implication. Dès lors,  $p \implies q$  signifie que si  $p$  est vraie, alors  $q$  aussi. A noter aussi que si  $p$  est fausse, alors  $p \implies q$  est vraie, quelque soit  $q$  (i.e., du faux on peut conclure ce qu'on veut).

Le Tableau 2 reprend la table de vérité pour les différents opérateurs logiques.

A noter que  $p \iff q$  est équivalent à  $(p \implies q) \wedge (q \implies p)$ .

Enfin, la priorité des opérateurs logiques est la suivante ( $<$  signifie “plus prioritaire”) :

$\neg < \wedge, \vee < \implies, \iff$



$p$	$q$	$\neg p$	$p \wedge q$	$p \vee q$	$p \implies q$	$p \iff q$
Vrai	Vrai	Faux	Vrai	Vrai	Vrai	Vrai
Vrai	Faux	Faux	Faux	Vrai	Faux	Faux
Faux	Vrai	Vrai	Faux	Vrai	Vrai	Faux
Faux	Faux	Vrai	Faux	Faux	Vrai	Vrai

TABLE 2 – Table de vérité pour les opérateurs logiques usuels.

### 3.3.1.3 Quantificateurs Logiques

#### Quantificateur Universel

Soient  $P$  et  $Q$ , deux prédicats, et  $(l)$  une liste (non vide) d'identificateurs.

$$\forall (l) \cdot (P \implies Q)$$

est un prédicat représentant la *quantification universelle*. La signification (en français) du prédicat est “pour tout  $i$  tel que  $P$  alors  $Q$  (est vrai)”.

Dans le cadre du cours, nous allons utiliser la notation (relaxée) suivante :

$$\forall (l), P, Q$$

La quantification universelle peut être vue comme un *sucre syntaxique* simplifiant une suite de conjonctions. Par exemple, le prédicat :

$$\forall i, 0 \leq i \leq N-1, T[i] > 0$$

peut se comprendre comme suit :

$$T[0] > 0 \wedge T[1] > 0 \wedge \dots \wedge T[N-1] > 0$$

Enfin, lorsque la variable prend ses valeurs sur l'ensemble vide, la quantification est vraie, quel que soit le terme  $P$ . Par exemple :

$$\forall i, 0 \leq i \leq -1, T[i] = 0 == \text{True}$$

#### Quantificateur Existentiel

Soient  $P$  et  $Q$ , deux prédicats, et  $(l)$  une liste (non vide) d'identificateurs.

$$\exists (l) \cdot (P \wedge Q)$$

est un prédicat représentant la *quantification existentielle*. La signification (en français) du prédicat est “il existe (au moins) un  $i$  tel que  $P$  pour lequel  $Q$  (est vrai)”.

Dans le cadre du cours, nous allons utiliser la notation (relaxée) suivante :

$$\exists (l), P, Q$$

La quantification existentielle peut être vue comme un *sucre syntaxique* simplifiant une suite de disjonctions. Par exemple, le prédicat :

$$\exists i, 0 \leq i \leq N-1, T[i] > 0$$

peut se comprendre comme suit :

$$T[0] > 0 \vee T[1] > 0 \vee \dots \vee T[N-1] > 0$$

### 3.3.1.4 Quantificateurs Numériques

#### Somme

La somme d'une suite de termes est représentée à l'aide de la notation  $\Sigma$ .  
Ainsi, par exemple :

$$0 + 1 + 2 + \dots + N - 1$$

sera représenté de manière beaucoup plus condensée par

$$\sum_{i=0}^{N-1} i$$

Par définition, la somme sur un intervalle vide vaudra toujours le neutre de l'addition, soit 0 (i.e., somme à zéro terme). Soit :

$$\sum_{i=0}^{-1} i = 0$$

#### Produit

Le produit d'une suite de termes est représentée à l'aide de la notation  $\Pi$ .  
Ainsi, par exemple :

$$T[0] \times T[1] \times T[2] \times \dots \times T[N - 1]$$

sera représenté de manière beaucoup plus condensée par

$$\prod_{i=0}^{N-1} T[i]$$

Par définition, le produit sur un intervalle vide vaudra toujours le neutre de la multiplication, soit 1 (i.e., produit à zéro terme)

$$\prod_{i=0}^{-1} T[i] = 1$$

#### Minimum/Maximum

Le minimum (respectivement maximum) d'une suite de termes est représenté à l'aide la notation *min* (respectivement *max*).

$$\min_P Q$$

où  $P$  et  $Q$  sont des prédicats

Par exemple :

$$\text{minimum}\{T[0], T[1], \dots, T[N - 1]\}$$

sera adéquatement représenté par :

$$\min_{0 \leq i \leq N-1} (T[i])$$

## Dénombrement

Soient  $P$  et  $Q$ , deux prédicats, et  $i$  une variable libre<sup>a</sup>

$$\#i \cdot (P|Q)$$

est un prédicat représentant le *quantificateur de dénombrement*. La signification (en français) du prédicat est “le nombre de fois où le prédicat  $Q$  est vrai lorsque  $P$  (est vrai)”.

a. Pour rappel, une variable est dite *libre* quand le nom que l’on utilise pour définir un objet a de l’importance (il s’agit typiquement d’une variable non introduite par un quantificateur). A l’inverse, une variable est dite *liée* si le nom que l’on utilise pour définir un objet n’a pas d’importance (il s’agit typiquement d’une variable introduite par un quantificateur).

Par exemple, un prédicat indiquant le nombre de valeurs à 0 dans un tableau  $T$  à  $N$  valeurs entières se présente comme suit :

$$\#i \cdot (0 \leq i \leq N - 1 | T[i] = 0)$$

## Suite de l’Exercice

À vous ! Formalisez le problème et passez à la Sec. 3.3.3.

Si vous séchez, reportez-vous à l’indice à la Sec. 3.3.2

### 3.3.2 Indice

Pour formaliser un problème, il faut répondre aux deux questions suivantes :

- De quoi parle le Problème ?
- Quel est le Sujet ?

La réponse devrait mettre en évidence les éléments essentiels à la résolution du problème.

Ensuite, une bonne idée est d'illustrer le fonctionnement de la notation sur un dessin, qui sera ensuite traduit en prédicat.

#### Suite de l'Exercice

À vous ! Définissez le problème et passez à la Sec. **3.3.3**.

### 3.3.3 Mise en Commun de la Formalisation du Problème

En relisant l'énoncé, on remarque :

Écrivez une fonction qui délivre **la somme des éléments de rang pair d'un fichier**.

On devrait avoir une notation qui représente la somme des éléments pairs dans un sous-fichier.

Nous devrions aussi être en mesure de savoir mesurer le rang d'un élément. Comment faire ? Il faudrait connaître le longueur de  $f^-$ . En effet, s'il y a  $i$  éléments dans  $f^-$ , cela veut dire que le rang de l'élément sous la tête de lecture est  $i + 1$ , comme visible sur le dessin suivant, où la longueur de  $f^-$  est 2 :

Nous devrions être en mesure de savoir mesurer le rang d'un élément. Comment faire ? Il faudrait connaître le longueur de  $f^-$ . En effet, s'il y a  $i$  éléments dans  $f^-$ , cela veut dire que le rang de l'élément sous la tête de lecture est  $i + 1$ , comme illustré à la Fig. 2, où la longueur de  $f^-$  est 2 :

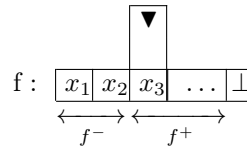


FIGURE 2 – Si  $|f^-| = 2$ , le rang de l'élément à lire est 3.

Nous devons donc définir deux notations :

- La **longueur** d'un fichier ;
- La **somme** des éléments de rang pair.

### 3.3.3.1 Longueur d'un Fichier

Cette notation permet de calculer facilement le rang d'un élément en calculant le nombre d'éléments dans  $f^-$ .

$$long(f) = \begin{cases} j - i + 1 & \text{si } f = g_i^j \wedge i \leq j \\ 0 & \text{sinon} \end{cases}$$

Si on peut décrire  $f$  comme un sous-fichier  $g$  entre les indices  $i$  et  $j$ , on a une longueur<sup>6</sup>. Sinon, celle-ci vaut 0.

### Suite de l'exercice

Vous pouvez maintenant passer à la suite de l'exercice :

- définition de la **somme** des éléments de rang pair ;
- **spécification** du module.

---

6. En fait, on peut toujours réécrire un fichier  $f^-$  comme étant un sous-fichier :

$$\exists k, k \in \{0 \dots n\}, f^- = f_1^k \Rightarrow long(f^-) = k$$

### 3.3.3.2 Somme des Eléments de Rang Pair

On s'inspire du Chapitre 3, slide 43, qui définit  $\sum f$ . On va définir  $\sum_{\%2=0}$ , qui représente la somme des éléments de rang pair d'un fichier :

$$f = \langle x_1, x_2, \dots, x_n \rangle,$$
$$\sum_{\%2=0} f = \sum_{i=1}^{n/2} x_{2*i}$$

En particulier, si  $n \leq 1$ , la somme est nulle.

### Suite de l'exercice

Vous pouvez maintenant passer à la suite de l'exercice :

- définition de la **longueur** d'un fichier ;
- **spécification** du module.

## 3.4 Spécifications

Il s'agit, maintenant, de proposer une interface pour le module.

Si vous voyez de quoi on parle, rendez-vous à la Section [3.4.3](#)

Si vous ne savez pas ce qu'est une spécification, allez à la Section [3.4.1](#)

Enfin, si vous ne voyez pas quoi faire, reportez-vous à l'indice [3.4.2](#)



### 3.4.1 Rappels sur les Spécifications

#### 3.4.1.1 Généralités

Une spécification se définit en deux temps :

**La PRÉCONDITION** implémente les suppositions. Elle caractérise donc les conditions initiales du module, i.e., les propriétés que doivent respecter les valeurs en entrée (i.e., paramètres effectifs) du module. Elle se définit donc sur les paramètres formels (qui seront initialisés avec les paramètres effectifs). La PRÉCONDITION doit être satisfaite avant l'exécution du module.

**La POSTCONDITION** implémente les certifications. Elle caractérise les conditions finales du résultat du module. Dit autrement, la POSTCONDITION décrit le résultat du module sans dire comment il a été obtenu. La POSTCONDITION sera satisfaite après l'invocation.

A l'instar de la définition d'un problème, un module doit toujours avoir une POSTCONDITION (un module qui ne fait rien ne sert à rien) mais il est possible que le module n'ait pas de PRÉCONDITION (e.g., le module ne prend pas de paramètres formels).

La spécification d'un module se met, en commentaires, avec le prototype du module. L'ensemble (i.e., spécification + prototype) forme l'*interface* du module.

Contrairement au cours INFO0946, les spécifications d'un module seront, dorénavant, exprimées de manière formelle (i.e., à l'aide de prédicats et de notations).

#### 3.4.1.2 Construction d'une Interface

Pour construire l'interface d'un module, il est souhaitable de commencer par faire un dessin. Le dessin, qui naturellement sera lié à Invariant Graphique, doit représenter des situations particulières du problème à résoudre. La situation initiale (i.e., avant la première instruction de la ZONE 1/INIT) doit aider pour trouver le prototype du module, ainsi que la PRÉCONDITION. La situation finale (i.e., après exécution de la ZONE 3/END) doit vous aider à trouver la POSTCONDITION.

En s'appuyant sur les dessins, il suffit ensuite de répondre à trois questions pour construire l'interface d'un module :

Question 1 : Quels sont les objets dont j'ai besoin pour atteindre mon objectif? Répondre à cette question permet de construire le prototype du module (éventuel type de retour, identificateur du module, paramètres formels).

Question 2 : Quel est l'objectif du module? Répondre à cette question permet d'obtenir la POSTCONDITION, représentée à l'aide d'un prédicat.

Question 3 : Quelles sont les contraintes sur les paramètres? Répondre à cette question permet d'obtenir la PRÉCONDITION, représentée à l'aide d'un prédicat.

L'avantage de faire un dessin est, bien entendu, de dresser un pont avec l'Invariant Graphique, mais aussi de faciliter la production d'un prédicat (il "suffit" de traduire le dessin en un prédicat).

#### 3.4.1.3 Exemple

A titre d'exemple, essayons de spécifier une fonction qui permet de calculer le produit de tous les entiers entre des bornes fournies, i.e.,  $a$  et  $b$  (avec  $b > a$ ).

Le problème peut s'illustrer comme indiqué à la Fig. 3. On dessine une droite (représentant les nombres qu'on va manipuler) avec les bornes de l'intervalle d'intérêt. La flèche bleue indique ce qu'on doit calculer.

##### Question 1

Pour atteindre l'objectif, nous avons simplement besoin des bornes de l'intervalle. Sur la Fig. 3, on peut clairement voir que le calcul du produit s'étale entre  $a$  et  $b$ . Ce sont les seuls éléments "extérieurs" dont on a besoin pour résoudre le problème.  $a$  et  $b$  seront donc les paramètres formels du module.

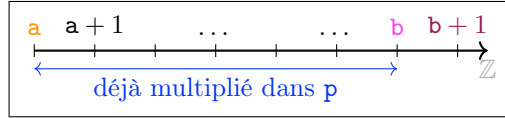


FIGURE 3 – Exemple de représentation graphique du problème à résoudre.

La Fig. 3 illustre aussi le fait qu'on manipule des entiers ( $\mathbb{Z}$ ). **a** et **b** seront donc de type `int`.

Enfin, on peut adéquatement nommer le module `produit()`, ce qui permet de naturellement décrire ce qu'il fait.

Tout ceci nous permet de dériver le prototype du module :

```
1 produit(int a, int b);
```

### Question 2

L'objectif du module est de calculer (et donc retourner) le produit de tous les entiers dans l'intervalle  $[a, b]$ . Soit :

$$a \times (a + 1) \times \cdots \times (b - 1) \times b$$

On peut représenter ce produit de manière plus condensée en utilisant le **quantificateur numérique** du produit :  $\prod$ . Soit :

$$\prod_{i=a}^b i$$

Puisque **a** et **b** sont de type `int`, le produit résultat sera aussi de type `int`. Le module `produit()` est donc une fonction retournant un `int`. On représente cela de la façon suivante :

$$produit = \prod_{i=a}^b i$$

Enfin, **a** et **b** ne sont pas modifiés par le module.

A ce stade, on obtient donc l'interface incomplète suivante :

```
1 /*
2  * POSTCONDITION : produit =  $\prod_{i=a}^b i \wedge a = a_0 \wedge b = b_0$ 
3  */
4 int produit(int a, int b);
```

### Question 3

Enfin, nous pouvons déterminer si il existe des conditions sur les paramètres. C'est bien le cas ici. L'énoncé du problème et la Fig. 3 indiquent clairement que **b** est strictement plus grand que **a**. On peut le représenter de la façon suivante :

$$b > a$$

## Interface

Au final, l'interface du module est la suivante :

```
1 /*  
2  * PRÉCONDITION :  $b > a$   
3  * POSTCONDITION :  $\text{produit} = \prod_{i=a}^b i \wedge a = a_0 \wedge b = b_0$   
4  */  
5 int produit(int a, int b);
```

## Suite de l'Exercice

À vous ! Proposez les interfaces pour les SP et passez à la Sec. [3.4.3](#).

Si vous séchez, reportez-vous à l'indice à la Sec. [3.4.2](#)

### 3.4.2 Indice

Pour le module, répondez aux **trois questions** en vous appuyant sur un schéma. Le schéma permettra d'aisément traduire la situation particulière (PRÉCONDITION ou POSTCONDITION) sous forme formelle.

Ne soyez pas ambigu dans la PRÉCONDITION, ni dans la POSTCONDITION. Au contraire, soyez le plus précis possible (rappelez-vous, un prédicat est une formulation mathématique et, dès lors, ne peut souffrir d'une quelconque ambiguïté). On n'oubliera pas de s'appuyer sur les **notations** introduites.

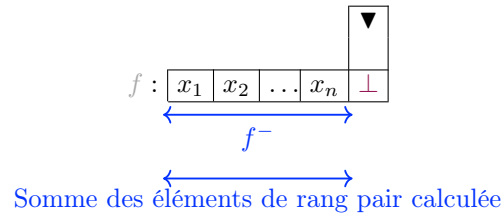
Une fois que c'est fait, rédigez l'interface du module.

### Suite de l'Exercice

À vous ! Spécifiez les SP et passez à la Sec. **3.4.3**.

### 3.4.3 Mise en Commun des Spécifications

On peut représenter le module de la façon suivante :



En entrée, on demande seulement le nom du fichier (qui servira à créer **f**) est initialisé. Soit :

PRÉCONDITION  $\equiv nom\_fichier\ init$

L'objectif du module est de calculer la somme des éléments de rang pair (on doit utiliser la notation définie précédemment) et la retourner au code appelant. On peut donc nommer la fonction `somme_pair()`. La POSTCONDITION peut donc être :

POSTCONDITION  $\equiv somme\_paire = \sum_{\%2=0} f$

Au final, l'interface de la fonction est :

```

1 /*
2  * PRÉCONDITION : nom_fichier init
3  * POSTCONDITION : somme_paire =  $\sum_{\%2=0} f$ 
4  */
5 int somme_pair(char *nom_fichier);

```

#### Suite de l'exercice

Vous pouvez maintenant proposer un **Invariant de Boucle** pour résoudre le problème.

## 3.5 Invariant de Boucle

L'écriture de code impliquant un traitement itératif (i.e., boucle) nécessite au préalable la proposition d'un Invariant de Boucle.

Si vous voyez de quoi on parle, rendez-vous à la Section [3.5.3](#)  
Si vous ne savez pas ce qu'est un Invariant de Boucle, allez la Section [3.5.1](#)  
Enfin, si vous ne voyez pas quoi faire, reportez-vous à l'indice [3.5.2](#)

## 3.5.1 Rappels sur l'Invariant de Boucle

### 3.5.1.1 Généralités

Un Invariant de Boucle est une **propriété** vérifiée à chaque évaluation du Gardien de Boucle.  
L'Invariant de Boucle présente un **résumé** de tout ce qui a déjà été calculé, jusqu'à maintenant (i.e., jusqu'à l'évaluation courante du Gardien de Boucle), par la boucle.

Le fait que l'Invariant de Boucle soit une propriété est important : ce n'est pas du code, ce n'est pas exclusivement destiné au langage C. Tout programme qui inclut une boucle doit s'appuyer sur un Invariant de Boucle.

#### Alerte : Ce que l'Invariant de Boucle n'est pas

De manière générale, un Invariant de Boucle

- n'est pas une instruction exécutée par l'ordinateur ;
- n'est pas une directive comprise par le compilateur ;
- n'est pas une preuve de correction du programme ;
- est indépendant du Gardien de Boucle ;
- ne garantit pas que la boucle se termine <sup>a</sup> ;
- n'est pas une assurance de l'efficacité du programme.

a. Seule la Fonction de Terminaison vous permet de garantir la terminaison

### 3.5.1.2 Sémantique Formelle

La *sémantique* formelle de l'Invariant de Boucle est présentée ici :

```
1 // {PRÉCONDITION}
2 INIT
3 // {INV}
4 while (B) {
5     // {INV ∧ B}
6     ITER
7     // {INV}
8 }
9 // {INV ∧ ¬B}
10 END
11 // {POSTCONDITION}
```

**INIT** correspond à la ZONE 1. Ce sont donc les instructions qui, partant de la PRÉCONDITION, doivent amener la boucle et, donc, l'Invariant de Boucle.

**B** correspond au Gardien de Boucle. B doit être vrai pour pouvoir entrer dans la boucle.

**ITER** correspond à la ZONE 2. Il s'agit donc du Corps de la Boucle. Ce sont donc les instructions qui doivent être répétées pour faire avancer le problème. Dans ITER, le point de départ est la situation particulière  $\{INV \wedge B\}$  et l'objectif, après la dernière instruction du Corps de la Boucle, est de restaurer l'Invariant de Boucle.

$\neg B$  correspond au Critère d'Arrêt. Quand  $\neg B$  est vérifié, alors on sort de la boucle.

**END** correspond à la ZONE 3. Il s'agit des instructions qui, une fois la boucle terminée (i.e.,  $\{INV \wedge \neg B\}$ ), amène la POSTCONDITION.

### 3.5.1.3 Règles pour l'Élaboration d'un Invariant Graphique

Bien que le cours INFO0947 ait pour objectif de présenter la programmation sous l'angle de ses fondements mathématiques (et, donc, par conséquence un Invariant de Boucle est présenté sous la forme d'un prédicat), il est toujours souhaitable de passer par l'étape dessin (et donc, l'Invariant Graphique).

Nous rappelons, ici, les 7 règles pour l'élaboration d'un Invariant Graphique satisfaisant (en les illustrants avec la Fig. 4) :

- Règle 1 : réaliser un dessin pertinent par rapport au problème (la droite des entiers dans la Fig. 4) et le nommer ( $\mathbb{Z}$  dans la Fig. 4) ;
- Règle 2 : placer sur le dessin les bornes de début et de fin ( $a$ ,  $b$ , et  $b + 1$  dans la Fig. 4) ;
- Règle 3 : placer une ligne de démarcation qui sépare ce qu'on a déjà calculé dans les itérations précédentes de ce qu'il reste encore à faire (la ligne rouge sur la Fig. 4). Si nécessaire, le dessin peut inclure plusieurs lignes de démarcation ;
- Règle 4 : étiqueter proprement chaque ligne de démarcation avec, e.g., une variable ( $i$  sur la Fig. 4) ;
- Règle 5 : décrire ce que les itérations précédentes ont déjà calculé. Ceci implique souvent d'introduire de nouvelles variables ("déjà multiplié dans  $p$ " dans la Fig. 4) ;
- Règle 6 : identifier ce qu'il reste à faire dans les itérations suivantes ("à multiplier" dans la Fig. 4) ;
- Règle 7 : Toutes les structures identifiées et variables sont présentes dans le code ( $a$ ,  $b$ ,  $i$ , et  $p$  sur la Fig. 4)<sup>7</sup>.

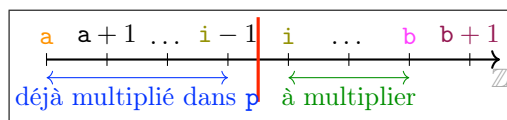


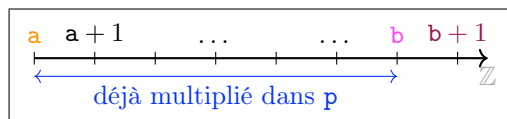
FIGURE 4 – Exemple d'Invariant Graphique pour le calcul du produit des entiers entre deux bornes.

### 3.5.1.4 Comment Trouver un Invariant Graphique ?

C'est la grande question. On peut envisager plusieurs méthodes mais, dans le cadre d'un GAMECODE et du cours, il nous semble plus pertinent d'insister sur la méthode de *l'éclatement de la POSTCONDITION*. La technique est, finalement, assez simple :

1. repartir de la représentation graphique du problème et, en particulier, de la POSTCONDITION ;
2. appliquer les règles de conception d'un Invariant Graphique sur le dessin représentant la POSTCONDITION ;

Par exemple, l'Invariant Graphique du problème de calcul du produit des entiers entre deux bornes peut facilement être inféré de la **représentation graphique du problème**. Reprenons là ici :



On voit bien qu'on dispose, ainsi, d'une situation particulière (en général, la situation à la sortie de la boucle). Naturellement, le dessin s'accommode déjà de la Règle 1 et Règle 2. Placer une ligne de démarcation (Règle 3) naturellement rétrécit la zone bleue à une situation générale (Règle 5). Il suffit d'étiqueter correctement cette ligne de démarcation avec une variable (Règle 4) et, ensuite, de rajouter l'information sur la zone encore à traiter dans les itérations suivantes (Règle 6). L'Invariant Graphique ainsi produit est alors complet et cohérent avec les règles de conception. Il ne reste plus qu'à le transformer en Invariant Formel.

7. Plus de détails sur le lien entre l'Invariant de Boucle et la construction du code dans le **rappel** associé



### 3.5.1.5 De l'Invariant Graphique à l'Invariant Formel

A des fins d'exemple, traduisons l'Invariant Graphique donné à la Fig. 4 en un Invariant Formel. Un Invariant Graphique bien formé (i.e., qui respecte les 7 règles) contient toutes les informations nécessaires pour un Invariant Formel. Passer de l'un à l'autre est donc une simple question de traduction d'un langage à l'autre (i.e., du dessin aux mathématiques).

$$\text{INV} \equiv a \leq i \leq b + 1 \quad (1)$$

$$\wedge p = \prod_{j=a}^{i-1} j \quad (2)$$

$$\wedge a = a_0 \quad (3)$$

$$\wedge b = b_0 \quad (4)$$

Ce qui signifie :

1. la variable de parcours,  $i$ , est dans l'intervalle d'intérêt  $([a, b])$ . Cela correspond à la Règle 2 et la Règle 4 de l'Invariant Graphique ;
2.  $p$  contient le produit jusqu'à maintenant. Cela correspond à Règle 5. A noter que le produit se fait entre  $a$  et  $i-1$ , ce qui implicitement positionne la variable de parcours,  $i$ , par rapport à la ligne de démarcation (Règle 3 et Règle 4) ;
3.  $a$  n'est pas modifié par le traitement itératif ;
4.  $b$  n'est pas modifié par le traitement itératif ;
5. les différentes parties de l'Invariant Formel sont connectées à l'aide de l'opérateur de conjonction ( $\wedge$ ) ;
6. la partie encore à traiter (Règle 6) est implicite dans l'Invariant Formel ;
7. toutes les variables et structures identifiées dans l'Invariant Graphique sont bien présentes dans l'Invariant Formel (Règle 7).

### Suite de l'Exercice

À vous ! Proposez les Invariants de Boucle pour les SP et passez à la Sec. 3.5.3.

Si vous séchez, reportez-vous à l'indice à la Sec. 3.5.2

### 3.5.2 Indice

Contrairement au cours INFO0946, les Invariants de Boucle sont ici proposés sous la forme de prédicats mathématiques. Il est néanmoins compliqué de proposer, directement, le prédicat.

L'approche la plus raisonnable reste donc de partir sur un Invariant Graphique et de le traduire, ensuite, sous la forme d'un Invariant Formel. Pour l'Invariant Graphique, il suffit de repartir des règles définies dans le cours INFO0946 (en les adaptant aux fichiers – par exemple la ligne de démarcation sera naturellement représentée par la tête de lecture/écriture). Dans l'état actuel des développements, le **GLI** ne vous permet pas encore de dessiner un fichier séquentiel.

Un Invariant de Boucle décrit particulièrement ce que l'on a déjà fait au fil des itérations précédentes. Dans le cadre d'un fichier séquentiel, on possède une notation des éléments déjà vus :  $f^-$ . L'Invariant de Boucle parlera donc **quasi toujours** de  $f^-$ . Ce qu'il reste à calculer correspond au traitement à appliquer à  $f^+$ .

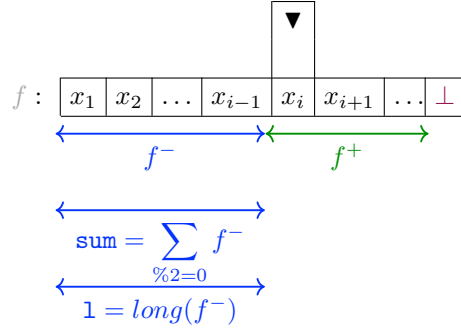
Une bonne technique consiste à retravailler la POSTCONDITION (cfr. Chapitre 2, Slide 29) et d'y faire apparaître  $f^-$  (par exemple, en plaçant la tête de lecture/écriture au « milieu » du fichier). Il faut ensuite se poser la question : « à l'itération  $i$ , de quelle-s information-s ai-je besoin pour traiter le  $i^{\text{e}}$  élément que je m'apprête à lire dans le fichier ? » (dans ce cas, on a évidemment  $\text{long}(f^-) = i - 1$ ).

### Suite de l'Exercice

À vous ! Proposez l'Invariant de Boucle et passez à la Sec. **3.5.3**.

### 3.5.3 Mise en Commun de l'Invariant de Boucle

Commençons par proposer un Invariant Graphique (construction basée sur l'éclatement de la **POSTCONDITION**) :



On commence par dessiner un fichier. On place ensuite  $f^-$  et  $f^+$  à l'aide de la tête de lecture/écriture (qui sert de ligne de démarcation). Comme suggéré, on essaie de décrire ce qu'il faut connaître sur  $f^-$ . Ensuite, il faut pouvoir déterminer le rang de l'élément  $i$  pour déterminer si  $i$  est pair. Cette information peut être calculée en connaissant le nombre d'éléments présents dans  $f^-$ , ce que l'on stocke dans la variable  $1$ . La variable **sum** accumule la somme des éléments de rang pair jusqu'à maintenant (i.e., sur  $f^-$ ).

La traduction de cet Invariant Graphique en Invariant Formel est la suivante :

$$\text{INV} \equiv 1 = \text{long}(f^-) \wedge \text{sum} = \sum_{\%2=0} f^-$$

#### Suite de l'exercice

On peut maintenant passer à la **construction** du code.

## 3.6 Construction du Code

Une fois l'Invariant Formel proposé, on peut construire le code en adoptant l'approche constructive et on précisant, à chaque étape, les assertions intermédiaires.

Si vous voyez de quoi on parle, rendez-vous à la Section	3.6.3
Si vous ne savez pas comment fonctionne l'approche constructive, allez la Section	3.6.1
Enfin, si vous ne voyez pas quoi faire, reportez-vous à l'indice	3.6.2

### 3.6.1 Rappels sur l'Approche Constructive

#### 3.6.1.1 Généralités

Voici les différentes étapes de l'Approche Constructive.

**INIT** De la PRÉCONDITION, il faut arriver dans une situation où l'Invariant de Boucle est vrai.

**B** Il faut trouver d'abord le Critère d'Arrêt  $\neg B$  et en déduire le Gardien de Boucle,  $B$ .

**ITER** Il faut faire progresser le Corps de la Boucle puis restaurer l'Invariant de Boucle.

**END** Vérifier si la POSTCONDITION est atteinte si  $\{Inv \wedge \neg B\}$ , sinon, amener la POSTCONDITION.

**Fonction de Terminaison** Il faut donner une Fonction de Terminaison pour montrer que la boucle se termine (cfr. INFO0946, Chapitre 3, Slides 97  $\rightarrow$  107).

Tout ceci doit être justifié sur base de l'Invariant de Boucle, à l'aide d'assertions intermédiaire. Notez que l'ordre dans lequel vous faites ces étapes n'a pas d'importance !

On se retrouvera donc dans la situation suivante :

```
1 // {PRÉCONDITION}
2 INIT
3 // {INV}
4 while (B) {
5     // {INV ∧ B}
6     ITER
7     // {INV}
8 }
9 // {INV ∧ ¬B}
10 END
11 // {POSTCONDITION}
```

Pour chaque étape, on démarre d'un point de départ et doit dériver les instructions (ainsi que les assertions intermédiaires) permettant d'arriver à l'objectif. Ainsi :

**{PRÉCONDITION} INIT {INV}** : INIT est donc un bloc d'instructions pour lequel  $\{PRÉCONDITION\}$  est la précondition (i.e., point de départ) et  $\{INV\}$  la postcondition (i.e., point d'arrivée). À nous de trouver comment arriver au but ( $\{INV\}$ ) en partant du point de départ ( $\{PRÉCONDITION\}$ ). À noter que INIT peut être éventuellement vide. Dans ce cas, il faut démontrer que  $\{PRÉCONDITION\} \implies \{INV\}$ .

**{INV ∧ B} ITER {INV}** : ITER est donc un bloc d'instructions pour lequel  $\{Inv \wedge B\}$  est la précondition (i.e., le point de départ) et  $\{INV\}$  la postcondition (i.e., le point d'arrivée). À nous de trouver comment arriver au but ( $\{INV\}$ ) en partant du point de départ ( $\{INV \wedge B\}$ ) tout en faisant avancer le problème. Il faut donc garantir qu'on a conservé notre Invariant de Boucle à la fin du Corps de la Boucle.

**{INV ∧ ¬B} END {POSTCONDITION}** : END est donc un bloc d'instructions pour lequel  $\{INV \wedge \neg B\}$  est la précondition (i.e., point de départ) et  $\{POSTCONDITION\}$  la postcondition (i.e., le point d'arrivée). À nous de trouver comment arriver au but ( $\{POSTCONDITION\}$ ) en partant du point de départ ( $\{INV \wedge \neg B\}$ ). À noter que END peut être éventuellement vide. Dans ce cas, il faut démontrer que  $\{INV \wedge \neg B\} \implies \{POSTCONDITION\}$ .

On voit donc bien que l'Invariant de Boucle est l'étape clé autour de laquelle s'articule non seulement la construction de la boucle mais aussi ce qui se passe avant et après la boucle. L'Invariant de Boucle est donc la colle entre les différentes parties du code.

#### Alerte : Oubli

Attention, pour chaque partie, on n'oublie pas d'indiquer les assertions intermédiaires. C'est obligatoire afin de s'assurer que les différentes parties s'enchaînent correctement.

### 3.6.1.2 Exemple

Afin d'illustrer l'approche constructive, travaillons sur un exemple. Il s'agit de rédiger une fonction qui permet de calculer le produit de tous les entiers entre des bornes fournies, i.e.,  $a$  et  $b$  (avec  $b > a$ ).

Pour rappel, l'interface de la fonction est la suivante :

```

1 /*
2  * PRÉCONDITION :  $b > a$ 
3  * POSTCONDITION :  $\text{produit} = \prod_{i=a}^b i \wedge a = a_0 \wedge b = b_0$ 
4  */
5 int produit(int a, int b);

```

L'Invariant Graphique est le suivant :

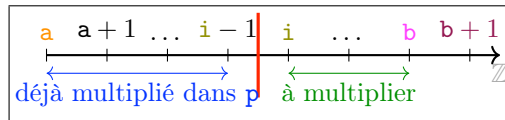


FIGURE 5 – Invariant Graphique pour le calcul du produit des entiers entre deux bornes.

L'Invariant Formel est le suivant :

$$\text{INV} \equiv a \leq i \leq b + 1$$

$$\wedge p = \prod_{j=a}^{i-1} j$$

$$\wedge a = a_0$$

$$\wedge b = b_0$$

#### INIT

Sur base de l'Invariant Formel :

$$\text{INV} : a \leq i \leq b + 1 \wedge p = \prod_{j=a}^{i-1} j \wedge a = a_0 \wedge b = b_0$$

il faut rendre ceci vrai en partant de la PRÉCONDITION (i.e.,  $b > a$ ). Cela consiste donc à :

- déclarer les variables nécessaires. Soit  $i$  (la variable d'itération) et  $p$  (l'accumulateur pour le produit cumulatif) ;
- initialiser les variables. La valeur de  $i$  est donnée par  $a \leq i \leq b + 1$ . La valeur de  $p$  en découle naturellement.

On obtient donc la séquence suivante :

```

1 // { $b > a$ }
2 int i, p;
3 // { $i = ?? \wedge p = ?? \wedge a = a_0 \wedge b = b_0 \wedge b > a$ }
4 i = a;
5 // { $i = a \wedge p = ?? \wedge a = a_0 \wedge b = b_0 \wedge b > a$ }
6 // { $a \leq i \leq b + 1 \wedge p = ?? \wedge a = a_0 \wedge b = b_0$ }
7 p = 1;
8 // { $a \leq i \leq b + 1 \wedge p = 1 \wedge a = a_0 \wedge b = b_0$ }
9 // { $\Rightarrow \text{Inv}$ }

```

On peut écrire la ligne 6 car, par la PRÉCONDITION,  $b > a$ . Soit  $a \leq b + 1$  (les deux expressions sont identiques).

La transition ligne 8  $\rightarrow$  ligne 9 est permise car

$$p = 1$$

$$p = \prod_{j=a}^{i-1}$$

$i$  est initialisé à  $a$  (ligne 4), ce qui donne  $p = \prod_{j=a}^{a-1}$ . Soit un produit à 0 termes. Par **définition**,  $p$  vaut le neutre de la multiplication, soit 1.

## B et Fonction de Terminaison

Le Critère d'Arrêt ( $\neg B$ ) est donné par le schéma suivant :

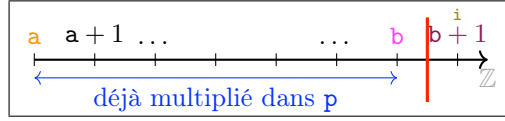


FIGURE 6 – Exemple d'Invariant Graphique pour le calcul du produit des entiers entre deux bornes.

C'est confirmé par la partie suivante de l'Invariant Formel :

$$a \leq i \leq b + 1$$

On a donc :

$$\neg B \equiv i = b + 1$$

On en dérive naturellement le Gardien de Boucle (B) :

$$B \equiv i \leq b$$

Pour la Fonction de Terminaison, il suffit d'estimer la taille de la zone "encore à traiter" dans l'**Invariant Graphique**. Soit :  $(b - a + 1) - (i - a)$ . Ce qui donne :

$$f \equiv b - i + 1$$

## ITER

```

1 | while(i <= b){
2 |   // {Inv ∧ B ≡ a ≤ i ≤ b + 1 ∧ p = ∏_{j=a}^{i-1} j ∧ a = a_0 ∧ b = b_0 ∧ i <= b}
3 |   // {a ≤ i ≤ b ∧ p = ∏_{j=a}^{i-1} j ∧ a = a_0 ∧ b = b_0}
4 |   p *= i;
5 |   // {a ≤ i ≤ b ∧ p = [∏_{j=a}^{i-1} j] × i ∧ a = a_0 ∧ b = b_0}
6 |   // {a ≤ i ≤ b ∧ p = ∏_{j=a}^i j ∧ a = a_0 ∧ b = b_0}

```

```

7   i++;
8   // {INV}
9 }

```

La transition ligne 2  $\rightarrow$  ligne 3 est naturelle dans le sens où il s'agit de réécrire le prédicat (ligne 3) de sorte qu'il intègre complètement  $\{INV \wedge B\}$  (ligne 2).

Cette situation de départ nous indique qu'il faut faire progresser le problème, soit accumuler encore une valeur dans le produit cumulatif (ligne 4). Cette instruction nous conduit à la situation décrite à la ligne 6 (après réécriture du prédicat à la ligne 5).

Pour restaurer l'Invariant Formel (qui est l'objectif à atteindre pour ITER), il suffit d'incrémenter  $i$  d'une unité, ce qui permettra de restaurer  $a \leq i \leq b + 1$  et  $p = \prod_{j=a}^{i-1} j$ .

**END**

À la sortie de la boucle, on est dans la situation  $\{INV \wedge \neg B\}$ . Soit :

$$a \leq i \leq b + 1 \wedge p = \prod_{j=a}^{i-1} j \wedge a = a_0 \wedge b = b_0 \wedge i = b + 1$$

On voit que la variable d'itération ( $i$ ) prend une valeur particulière dans l'intervalle  $a \leq i \leq b + 1 : b + 1$ . On peut donc réécrire le prédicat en remplaçant la variable  $i$  par sa valeur particulière  $b + 1$ . Soit :

$$p = \prod_{j=a}^{b+1-1} j \wedge a = a_0 \wedge b = b_0$$

Et encore :

$$p = \prod_{j=a}^b j \wedge a = a_0 \wedge b = b_0$$

Cette situation correspond à la POSTCONDITION, à la différence près que  $p$  contient le produit cumulatif au lieu de la fonction `produit()` ! Dès lors, à la sortie de la boucle, il nous suffit de retourner la valeur de  $p$ .

Ce qui nous donne :

```

1 // {INV ∧ ¬B}
2 return p;
3 // {POSTCONDITION}

```

## Code Complet

```

1 int produit(int a, int b){
2   // {PRÉCONDITION}
3   int i, p;
4   i = a;
5   p = 1;
6   // {INV}
7   while(i <= b){
8     p *= i;
9     i++;
10  } //fin while - i
11
12  return p;
13  // {POSTCONDITION}
14 } //fin produit()

```



### Suite de l'Exercice

À vous ! Construisez le code pour les SP et passez à la Sec. 3.6.3.  
Si vous séchez, reportez-vous à l'indice à la Sec. 3.6.2

### 3.6.2 Indice

L'ordre de résolution des SP n'a pas d'importance (il n'y a d'ailleurs pas de SP pour ce GameCode). Pour chaque SP, il faut cependant construire le code en suivant :

**INIT** De la PRÉCONDITION, il faut arriver dans une situation où l'Invariant de Boucle est vrai.

**B** Il faut trouver d'abord le Critère d'Arrêt  $\neg B$  et en déduire le Gardien de Boucle.

**ITER** Il faut faire progresser le programme puis restaurer l'Invariant de Boucle.

**END** Vérifier si la POSTCONDITION est atteinte si  $\{INV \wedge \neg B\}$ , sinon, amener la POSTCONDITION.

**Fonction de Terminaison** Il faut donner une Fonction de Terminaison pour montrer que la boucle se termine.

Tout ceci doit être justifié sur base de l'Invariant de Boucle, à l'aide d'assertions intermédiaires. Notez que l'ordre dans lequel vous faites ces étapes n'a pas d'importance !

#### Suite de l'Exercice

À vous ! Construisez le code pour chaque SP et passez à la Sec. [3.6.3](#).

### 3.6.3 Mise en Commun de l'Approche Constructive

Pour rappel, l'interface de la fonction est la suivante :

```

1 /*
2  * PRÉCONDITION : nom_fichier init
3  * POSTCONDITION : somme_paire =  $\sum_{\%2=0} f$ 
4  */
5 int somme_paire(char *nom_fichier);

```

L'Invariant Formel est le suivant :

$$\text{INV} \equiv 1 = \text{long}(f^-) \wedge \text{sum} = \sum_{\%2=0} f^-$$

#### INIT

Sur base de l'Invariant Formel :

$$\text{INV} \equiv 1 = \text{long}(f^-) \wedge \text{sum} = \sum_{\%2=0} f^-$$

il faut rendre ceci vrai en partant de la PRÉCONDITION. Cela consiste donc à :

- déclarer les variables nécessaires. Nous avons besoin de **sum** (pour la somme cumulative), **1** pour la longueur de  $f^-$ , **f**, le fichier et **val** pour lire les valeurs dans le fichier.
- initialiser les variables.
  - **f**. Il suffit d'ouvrir le fichier en lecture. A ce moment-là, la tête de lecture/écriture sera sur le premier élément du fichier. Par définition, alors,  $f^-$  sera vide et  $f^+$  vaudra tout le fichier **f**.
  - **1**. Puisque  $f^-$  est vide, la longueur d'un sous-fichier vide est 0 (cfr. définition de  $\text{long}()$ ).
  - **sum**. La somme des éléments de rang pair se calcule sur  $f^-$ , qui est vide à ce stade. On se retrouve donc avec une somme à 0 élément. Par définition de l'opération de sommation, le résultat d'une somme à 0 terme est 0.
  - **val**. Il ne doit pas être initialisé. Sa seule utilité est de contenir la valeur courante lue dans le fichier.

On obtient donc :

```

1 // {PRÉCONDITION}
2 FILE f = fopen(nom_fichier, "r")
3 // {f^- = < >}
4 int val;
5 int sum = 0;
6 // {f^- = < > ∧ sum = 0}
7 // {f^- = < > ∧ sum =  $\sum_{\%2=0} f^-$ }
8 int l = 0;
9 // {f^- = < > ∧ sum =  $\sum_{\%2=0} f^- \wedge l = 0$ }
10 // {l = long(f^-) ∧ sum =  $\sum_{\%2=0} f^- \wedge f^- = < >$ }
11 // {⇒ Inv}

```

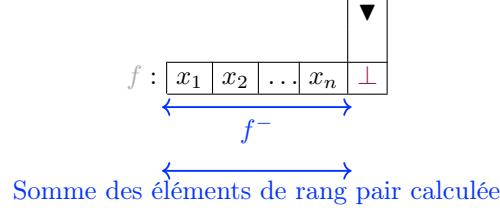


FIGURE 7 – Situation finale.

## B et Fonction de Terminaison

On commence par définir le Critère d'Arrêt en comparant la POSTCONDITION et l'Invariant de Boucle. On arrive alors à la situation décrite à la Fig. 7.

On sortira donc de la boucle quand on aura lu tout le fichier (i.e., la tête de lecture/écriture est sur  $\perp$ ). Ce qui nous donne :

$$\neg B \equiv eof(f)$$

Le Gardien de Boucle est donc naturellement :

$$B \equiv \neg eof(f)$$

La Fonction de Terminaison se calcule simplement sur la portion du fichier encore à lire. Soit :

$$f \equiv long(f^+)$$

## ITER

Nous partons de la situation particulière suivante :

$$INV \wedge B \equiv l = long(f^-) \wedge sum = \sum_{\%2=0} f^- \wedge \neg(eof)$$

Ce qui peut se réécrire :

$$INV \wedge B \equiv l = long(f^-) \wedge sum = \sum_{\%2=0} f^- \wedge f^+ \neq < >$$

Cela signifie qu'il reste, au moins, un élément à traiter dans le fichier. Soit :

```

1 while(!eof(f)){
2   // {Inv ∧ B}
3   read(f, val);
4   // {sum = ∑%2=0 f- ∧ l = long(f-) ∧ (f- = f-|| < val >)}
5   l++;
6   // {sum = ∑%2=0 f- ∧ l = long(f-) ∧ (f- = f-|| < val >) ⇒ val = xl}
7   if (!(l % 2))
8     // {sum = ∑%2=0 f- ∧ l = long(f-) ∧ val = xl ∧ l mod 2 = 0}
9     sum += val;
10    // {l = long(f-) ∧ sum = ∑%2=0 f-}
11    // {l = long(f-) ∧ sum = ∑%2=0 f-}
12    // {⇒ Inv}
13 }
```

On lit d'abord l'élément sous la tête de lecture et on calcule son rang grâce à  $l$ . Si  $l$  est pair, il faut ajouter cet élément dans la somme. Remarque :  $f^-$  est le sous-fichier  $f^-_0$  après la lecture ( $f^-|| < val > = f^-$ ).

**END**

À la sortie de la boucle, nous sommes dans la situation suivante :

$$\text{INV} \wedge \neg B \equiv l = \text{long}(f^-) \wedge \text{sum} = \sum_{\%2=0} f^- \wedge \text{eof}(f)$$

Cela signifie donc que  $f^-$  vaut tout le fichier. On peut donc réécrire de la façon suivante :

$$l = \text{long}(f) \wedge \text{sum} = \sum_{\%2=0} f$$

Partant de cette situation, pour atteindre la `PostCondition`, il suffit de retourner la valeur de `sum` et fermer le fichier. Soit :

```
1 // {INV ∧ ¬B}
2 fclose(f);
3 return sum;
4 // {PostCondition}
```

### Suite de l'exercice

On peut maintenant passer à la **mise en commun** du code.

## 3.7 Code Complet

```
1 int somme_paire(char *nom_fichier){
2     int sum = 0;
3     int l = 0;
4
5     while(!eof(f)){
6         read(f, val);
7         l++;
8         if(!(l % 2))
9             sum += val;
10    }//fin while
11
12    fclose(f);
13    return sum;
14 }//fin somme_paire()
```