

# Algorithmie en langage C

---

Semestre 3



---

# Table des matières

---

<b>I</b>	<b>Introduction à la programmation</b>	<b>5</b>
<b>1</b>	<b>Paradigmes de programmation</b>	<b>6</b>
1.1	Programmation fonctionnelles . . . . .	6
1.2	Programmation déclarative . . . . .	6
1.3	Programmation Impérative . . . . .	7
<b>2</b>	<b>Programmation impérative en C</b>	<b>8</b>
2.1	Description de l'organisation des données en mémoire . . . . .	8
2.2	Code syntaxe . . . . .	8
2.3	Structure d'une programme en C . . . . .	11
2.4	La compilation . . . . .	12
<b>3</b>	<b>Méthodologie de la programmation impérative</b>	<b>13</b>
3.1	Les différents types de problèmes . . . . .	13
3.2	Développement d'un algorithme . . . . .	13
3.3	Les différents étapes de développement d'un programme . . . . .	14
<b>II</b>	<b>Intérêt et utilisation des spécifications</b>	<b>16</b>
<b>4</b>	<b>Spécification d'un programme</b>	<b>17</b>
4.1	Mots clés à utiliser dans les prédicats . . . . .	17
4.2	Écriture de la spécification . . . . .	17
<b>5</b>	<b>Vérifications de programmes</b>	<b>19</b>
5.1	Le tableau de situation . . . . .	19

---

5.2	Vérification formelle de programmes avec les pfp . . . . .	20
<b>6</b>	<b>Ecriture d'un programme à partir de sa spécification</b>	<b>25</b>
6.1	Modèle de boucle . . . . .	25
<b>7</b>	<b>Transformation d'une spécification récursive en un programme itératif</b>	<b>28</b>
7.1	Récurtivité terminale . . . . .	28
<b>III</b>	<b>Annexes</b>	<b>30</b>
<b>A</b>	<b>Glossaire</b>	<b>31</b>
<b>B</b>	<b>Liste des codes sources</b>	<b>32</b>
<b>C</b>	<b>Exercices</b>	<b>33</b>
C.1	Initiation à la programmation . . . . .	33
C.2	Spécification . . . . .	35
C.3	Tableau de situation . . . . .	36
C.4	Plus faible précondition . . . . .	40
C.5	Écriture d'un programme à partir de sa preuve . . . . .	47
C.6	Exercices complets . . . . .	48
C.7	Transformation d'une spécification récursive en un programme itératif . . . . .	51

# Première partie

## Introduction à la programmation

# Paradigmes de programmation

## Sommaire

<b>1.1</b>	<b>Programmation fonctionnelles</b>	<b>6</b>
<b>1.2</b>	<b>Programmation déclarative</b>	<b>6</b>
<b>1.3</b>	<b>Programmation impérative</b>	<b>7</b>

Un paradigme est une manière de programmer, il en existe plusieurs :

- La programmation fonctionnelles (cf. 1.1)
- La programmation déclarative (cf. 1.2)
- La programmation impérative (cf. 1.3)

## 1.1 Programmation fonctionnelles

**Type de langage** Compilés<sup>1</sup> ou interprétés<sup>2</sup>.

**Entité de base** Appel de fonction

**Structure de contrôle** Approche récursive.

Elle est utilisée pour des systèmes critiques<sup>3</sup>. Elle à une approche très mathématiques, ce qui permet d'avoir des outils de preuves générique.

Elle possède une abstraction de l'environnement d'exécution, approche détachée de la machine, pas de notion de mémoire.

**Ex** Le Caml est un langage de programmation fonctionnelle

## 1.2 Programmation déclarative

**Type de langage** Interprété

**Entité de base** Règles de déduction logique.

**Structure de contrôle** Possède une abstraction de la machine cible.

1. Traduction du langage source vers le langage cible(compilation) + une édition de liens, qui est une instanciation sur la machine d'exécution (Recherche d'adresse, mémoire, résolution de fonctions) Elle peut être statique ou dynamique. Ex : C, Adda

2. Le langage source est traduit en langage cible à la volée par un interpréteur. Il est ainsi possible de modifier le programme pendant le fonctionnement du programme.

3. Besoin d'une sureté de fonctionnement

**Ex** Le prolog est un langage de programmation déclarative

## 1.3 Programmation Impérative

La programmation est directement liée à la machine d'exécution.

**Type de langage** Compilé ou Interprété

**Entité de base** Affectation d'une valeur à une variable, qui est une place en mémoire.

**Structure de contrôle** Séquence, sélection, répétition.

**Ex** C, Python, Ada ...

# Programmation impérative en C

## Sommaire

<b>2.1</b>	<b>Description de l'organisation des données en mémoire . . . . .</b>	<b>8</b>
<b>2.2</b>	<b>Code syntaxe . . . . .</b>	<b>8</b>
<b>2.3</b>	<b>Structure d'une programme en C . . . . .</b>	<b>11</b>
<b>2.4</b>	<b>La compilation . . . . .</b>	<b>12</b>

Énormément de langage sont fondés sur la syntaxe du langage C.

Il a été développé dans les années 1960 par Dennis Ritchie.

On trouvera toujours une partie description de l'organisation des données en mémoire<sup>1</sup>, nous aurons donc une déclaration de variables et un type de données.

```
1 type nomVariable;
```

Listing 2.1 – Syntaxe de déclaration de variable

## 2.1 Description de l'organisation des données en mémoire

Le C possède différents type de données :

**int** Entiers signés

**unsigned int** Entiers non signés

**float** Nombre réel sur 32bits.

**double** Nombre réel sur 64bits.

**char** Entier signé sur 8bits.

**pointeur** type\* ptr ; La case mémoire contient une adresse.

## 2.2 Code syntaxe

### 2.2.1 Blocs

```
1 bloc { // début du bloc
2 } //fin du bloc
```

Listing 2.2 – Syntaxe d'un bloc

1. C'est un grand tableau découpé en cases mémoire.



Toute variable est visible dans son bloc de déclaration et ses blocs imbriqués.

Un bloc transforme une séquence en action.

## 2.2.2 Séquence

```
1 action 1;  
2 action 2;  
3 action 3;
```

Listing 2.3 – Syntaxe des actions

## 2.2.3 Sélection

```
1 if(conditon) {  
2     action 1;  
3 } else {  
4     action 2;  
5 }
```

Listing 2.4 – Syntaxe d’une structure de contrôle

Condition est une expression booléenne<sup>2</sup>, si celle-ci est vrai, action 1 est exécuté, sinon action 2 est exécuté.

## 2.2.4 Répétition

```
1 while(condition) {  
2     action;  
3 }
```

Listing 2.5 – Syntaxe de répétition

Condition est une expression booléenne, tant que la condition est vrai, les actions se répètent.

## 2.2.5 Affectation

```
1 variable = expression;
```

Listing 2.6 – Syntaxe d’une affectation

variable reçoit expression, si celle-ci n’est pas du même type que variable, un cast<sup>3</sup> peut-être effectué.

---

2. Expression renvoyant vrai( $\neq 0$ ) ou faux( $= 0$ )

3. ou conversion de type, consiste à convertir un type vers un autre. (int vers double par exemple)

## 2.2.6 Opérateurs de base sur les types

= Affectation

+, -, /, \* Opérateurs arithmétiques.

&&, ||, ! Opérateurs logiques

==, !=, <, >, <=, >= Opérateurs booléens

++i, i++, --i, i-- Opérateur unaires d'incrément.

&, ^, |, <<, >> Opérateurs binaires

## 2.2.7 Opérateurs d'entrées / sorties

### Ecriture

```
1 printf('format', var1, var2);
```

Listing 2.7 – Syntaxe de l'appel de printf

La chaine format peut contenir une chaine de caractères, avec des caractères spéciaux :

'%d' Entier sous forme décimale

'%ox' Entier sous forme hexadécimale

'%f' Flottant

'%c' Caractère

'%s' Chaine de caractères

'\n' Vide le buffer et fait un retour chariot

'\t' Tabulation

'\r' Revient en début de ligne.


'...' RTFM

Les différents formats doivent être dans l'ordre des variables passés en paramètres.

### Lecture

```
1 scanf('format', &var1); // & représente l'adresse de la variable  
    dans laquelle écrire.
```

Listing 2.8 – Syntaxe de l'appel de scanf

 L'utilisation de cette fonction est risquée. En effet, un utilisateur malveillant peut écrire à des cases mémoires où il n'est pas autorisé.

## 2.2.8 Tableaux


Un tableau est une collection d'éléments de même type.

```

1 // avec tableau le nom de la variable et N la taille du tableau.
2 type tableau[N], i;
3 i = tableau[P]; //i recoit la valeur de la case P du tableau

```

Listing 2.9 – Syntaxe de déclaration d'un tableau

 Un tableau commence toujours à 0 et finit à N-1, ainsi, il faut faire très attention au dépassement de la taille d'un tableau.

## 2.2.9 Les sous-programmes

Un sous-programme est un sous-ensemble du programme dans sa hiérarchie fonctionnelle. En C, il correspond toujours à une fonction ou une procédure.

```

1 typeRetour nomFonction (typeArg1 nomArg1, typeArg2 nomArg2) {
2     /*
3         *   code
4         */
5     [return (valeur)];
6 }

```

Listing 2.10 – Syntaxe d'un sous programme

typeRetour peut posséder comme valeur les même types qu'une variable, voir 2.1 page 8. Celui-ci peut également être void, cela signifie que la fonction ne renvoie rien, c'est donc une procédure.

## 2.3 Structure d'un programme en C

### Interface

- Déclaration des fonctions (prototype)
- Constantes, types
- Comment utiliser le programme  
⇒ Écrire dans un fichier .h (header)
- Préprocesseur (define, macros, ...)

programme en C ne possède qu'une seul point d'entrée : une instruction est exécuté, c'est la fonction main.

### Implantation

- Définitions des fonctions : le code  
⇒ Écrire dans un fichier .c

```

1 int main (int argc, char **argv);
2 //le programme renvoie un entier. C'est le profil d'une fonction.

```

Listing 2.11 – Point d'entrée du programme: le main

## 2.4 La compilation

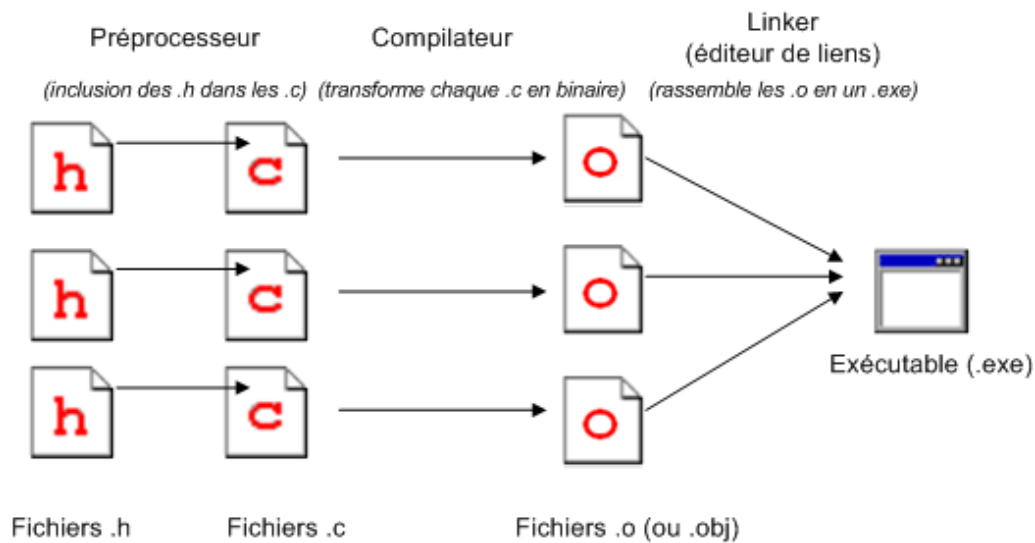


FIGURE 2.1 – La compilation

### 2.4.1 Étape 1 : Le préprocesseur

Le pré processeur sont les instructions situés en dehors d'un programme, ceux-ci sont préfixé par un dièse (#).

**Entrée** fichier.c

**Sortie** fichier obtenu une fois les modifications effectués.

```
1 #include // remplace par le contenu du fichier inclus
2 #define Arg1 Arg2 // remplace syntaxique de Arg1 par Arg2
```

Listing 2.12 – Exemple d'instructions pré-processeurs

### 2.4.2 Etape 2 : La compilation

**Entrée** fichier.c, fichier.h

**Sortie** fichier.o

```
1 gcc -c fic1.c fic2.c fic3.c # Créé les fichiers .c
2 gcc *.o nomExe #Créer l'executable.
```

**R** La compilation sera étudiée en détails lors des cours de L3 et M1

### 2.4.3 Étape 3 : L'édition de liens

Rassemble tous les fichiers binaires .o en un seul executable.

# Méthodologie de la programmation impérative

## Sommaire

<b>3.1</b>	<b>Les différents types de problèmes . . . . .</b>	<b>13</b>
<b>3.2</b>	<b>Développement d'un algorithme . . . . .</b>	<b>13</b>
<b>3.3</b>	<b>Les différents étapes de développement d'un programme . . . . .</b>	<b>14</b>

## 3.1 Les différents types de problèmes

### 3.1.1 Programmation «en petit»

**Données** celle-ci son simple, comme un tableau à N éléments.

**Problème** Petit.

**Résolution** Développement d'un algorithme afin de traiter ces données.

Cf. cours du S3.

### 3.1.2 Programmation en large

**Données** celle-ci son complexes, modélisation des données avec des types abstrait.

**Résolution** Développer de nombreux algorithmes afin de traiter le type abstrait.

Cf. cours du S4.

## 3.2 Développement d'un algorithme

C'est un processus à 4 étapes :

1. **Comprendre** le problème : identifier le «quoi».
2. **Spécification** du problème : formaliser le «quoi»
3. **Définir** un modèle de solution : identifier le «comment»
4. **Développer** et trouver l'algorithme : formaliser le «comment».

## 3.3 Les différents étapes de développement d'un programme

### 3.3.1 Étape 1 : comprendre le problème

Analyser du texte afin d'identifier les propriétés suivantes.

- Identifier les domaines du problèmes Le domaine pose les fondements scientifiques à utiliser par le programme.

**Ex** Arithmétique : se fonder sur la théorie du calcul

Topologique : se fonder sur les bases mathématiques de topologie

Il faut se poser la question «*est-ce calculable ?*» : est-ce que le problème peut être résolu par un ordinateur.

**Ex**

1. Corriger toutes les fautes d'orthographe dans un texte : Non calculable car il y a un manque d'informations sur la taille et la nature des données.
2. Calculer la factorielle d'un entier  $N \geq 0$  : calculable puisque la taille des données est fixée.

- évaluer les contraintes «*Physiques*» liées au problème.
    - Les contraintes liées à l'architecture et au fonctionnement de l'ordinateur
    - Les restrictions du problème.
  - Prendre des exemples et les traiter «manuellement»
- En sortie de cette étape, nous avons une description informelle des données et de leur traitements.

### 3.3.2 Étape 2 : Spécification du programme : spécification formelle

Utilisation du langage logique des précédents pour écrire le programme, la spécification est composée de 3 informations appelée le triplet de HOARE.

1. Prédicat d'entrée : Exprime les propriétés logiques des données en entrée.
2. nomDuProgramme (données en entrée E, données en sortie)
3. Prédicat de sortie exprime les propriétés logiques des résultats.

**Ex** Celui du facteur de  $N \geq 0$

- $N > 0 \wedge [(N \in \mathbb{N})] \wedge (N < 30)$
- `fact(N, f);`
- $f = N!$

### 3.3.3 Étape 3 : Donner un modèle de solution

Exprimer les différentes étapes de transformation des données en entrée vers les données en sortie.

- En langage naturel
- Sous forme fonctionnelle

### 3.3.4 Étape 4 : Programmer et unifier le programme

- Écriture en C du programme traduction du modèle vers le C.

### 3.3.5 Étape 5 : Vérification du programme

- Test d'exécution : Tableau de situation, vérification non exhaustive. Cf section 5.1 page 19.
- Preuve formelle par calcul de «Plus faible Pré condition» (Pfp). Cf chapitre 5.2 page 20.

## Deuxième partie

### Intérêt et utilisation des spécifications



# Spécification d'un programme

## Sommaire

4.1 Mots clés à utiliser dans les prédicats . . . . .	17
4.2 Écriture de la spécification . . . . .	17

**R** Durant ce chapitre, nous parlerons de programme, cependant cela est valable également pour les sous-programme

Un programme est spécifié par un triplet :

- Prédicat d'entrée  $P(E)$  ou précondition
- action  $(E, S)$
- Prédicat de sortie  $P(S)$  ou postcondition

Les prédicats sont écrits en utilisant le formalisme de la logique des prédicats et de opérations booléennes.

## 4.1 Mots clés à utiliser dans les prédicats

Les mots clés pouvant être utilisés :

- Les quantificateurs logiques :  $\forall$ (quelque soit),  $\exists$ (il existe),  $\nu$ (nombre de)
- Les connecteurs logiques :  $\wedge$ (et),  $\vee$ (ou),  $\rightarrow$ (implique),  $\leftrightarrow$ (équivalence),  $\neg$ (not)

## 4.2 Écriture de la spécification

C'est une traduction de l'énoncé et de l'analyse faite dans l'étape 1 de la méthodologie : c'est un **triplet** logique. La démarche pour écrire la spécification est la suivante.

- Identifier les propriétés des données d'entrée et les exprimer sous forme logique
- Identifier les propriétés sur les données en sortie et les exprimer sous forme logique.

**Ex** Écrire un programme qui trie un tableau  $T$  de  $N$  éléments.

- $N > 1$
- `trier (T, N, t);`
- $(\forall I : 0 \leq I < N - 1 \longrightarrow T[I] \leq T[I + 1]) \wedge$   
 $(\forall I : 0 \leq I < N \longrightarrow$   
 $(\nu J : 0 \leq J < N \wedge t[I] = t[J]) = (\nu J : 0 \leq J < N \wedge t[I] = T[J]))$

# Vérifications de programmes

## Sommaire

<b>5.1</b>	<b>Le tableau de situation</b>	<b>19</b>
<b>5.2</b>	<b>Vérification formelle de programmes avec les pfp</b>	<b>20</b>

Vérifier que le programme est correct revient à démontrer l'implication suivante :  
 $PE \rightarrow \text{pfp}(\text{action}(D, r, PS))$  ;

Avec pfp étant la plus faible précondition <sup>1</sup>.

**Tableau de situation** Système de réécriture des données en entrée <sup>2</sup> en fonction des actions du programme, cela permet de vérifier la cohérence du programme au niveau du contenu des variables. Cf section 5.1.

**pfp ou Plus Faible Précondition** C'est un système de réécriture permettant de transformer une formule logique en une autre selon le programme qui doit s'exécuter. C'est donc une réécriture syntaxique du prédicat de sortie en fonction des actions du programme. Cf section 5.2.

**R** Avec un tableau de situation ou des tests unitaires, il n'est pas possible de prouver qu'un programme est correct dans tous les cas, ceux-ci seront toujours basés sur un jeu d'essai. Le calcul de pfp permet de prouver qu'un programme est correct dans tous les cas, mais il est aussi plus long à effectuer, il est donc préférable de faire un tableau de situation rapidement, si le programme est incorrect, nous pourrions le voir rapidement..

## 5.1 Le tableau de situation

Le tableau de situation sert à tester le programme, la vérification n'est pas exhaustive. Il s'agit de vérifier que l'état des variables en mémoire est cohérent. Le tableau de situation peut être effectué à l'aide d'un ordinateur, via un débogueur par exemple.

**Données en entrée** Programme «instrumenté» : code source + point d'arrêt : localisation dans l'espace du programme d'une opération de photographie de l'état de l'ordinateur.

**Opération de transformation** Dénuder le programme et prendre les photos.

**Donnée en sortie** Liste de «photos» qui dévoile l'exécution de la même mémoire au cours de l'exécution.

1. wp weakest precondition

2. mémoire vers les données en sortie (mémoire)

**Ex**

```

/* (N > 0) ∧ (N > 30) */
int factorielle (int n) {
    if (0 == n) {
        return (1); // ... 1
    } else {
        int f = n; // ... 2
        while (--n > 0) {
            f *= n; // ... 3
        }
        return f; // ... 4
    }
}
/* factorielle = n! */

```

Listing 5.1 – Exercice 3

Nous choisis le jeu de données  $n = 0$  et  $n = 4$  afin de passer dans tous les cas.

	Échelle	n	f	point d'arrêt
$n = 0$	1	0	indéfini	1
	/	/	/	/
$n = 4$	indéfini	4	4	2
	indéfini	3	12	3
	indéfini	2	24	3
	indéfini	1	24	3
	24	6	24	4
	/	/	/	/

## 5.2 Vérification formelle de programmes avec les pfp

Ensemble de règles de réécriture permettant de transformer une formule logique en fonction des structures de base des langages de programmation.

Nous allons étudier le calcul d'une pfp pour un langage impératif, pour ceux-ci les structures de bases sont :

- affectation (section 5.2.3)
- Séquence (section 5.2.2)
- Sélection (section 5.2.3)
- répétition (section 5.2.4)

Règle de réécriture :  $\text{pfp}(\text{structure}, \text{formule}) = \text{formule}$

### 5.2.1 Calcul de pfp d'une affectation

La définition d'une affectation est disponible section 2.2.5.

$$pfp("x = e", Q) = Q_n^e$$

Avec la formule  $Q$  dans laquelle toutes les occurrences de « $x$ » sont remplacées par « $e$ » (remplacement textuel)

**Ex** Soit le programme suivant :

```
/* x > 0 */
x = x - 1
/* x ≥ 0 */
```

On doit se poser la question  $PE \rightarrow pfp(\text{programme}, PS)$  ?

$$\begin{aligned} (x > 0) &\rightarrow pfp("x = x - 1", x \geq 0) \\ (x > 0) &\rightarrow (x - 1 \geq 0) \\ (x > 0) &\rightarrow (x > 0) \end{aligned}$$

### 5.2.2 Calcul du pfp d'une séquence

La définition d'une séquence est disponible section 2.2.2.

$$pfp("a1; a2; a3", Q) = pfp("a1; a2;", pfp("a3;", Q));$$

**Ex**

```
/* f = i! */
i = i + 1;
f = f * i
/* f = i! */
```

$$\begin{aligned} f = i! &\rightarrow pfp("i = i + 1; f = f \times i;", f = i!) \\ f = i! &\rightarrow pfp("i = i + 1", pfp("f = f \times i", f = i!))^a \\ f = i! &\rightarrow pfp("i = i + 1", f \times i = i!) \\ f = i! &\rightarrow f \times (i + 1) = (i + 1)! \\ &\quad i! \times (i + 1) = (i + 1)! \text{ Par définition de } i! \end{aligned}$$

<sup>a</sup>. Évaluation de la "règle" la plus profonde

D'autres exercices sont disponibles annexe C.4.1 page 40.

### 5.2.3 Calcul du pfp de la sélection

La définition d'une sélection est disponible section 2.2.3.

$$\text{if}(B)\{a_1\}[\text{else}\{A_2\}]$$

$$\begin{aligned}\text{pfp}(\text{"if}(B)\{A_1\}\text{else}\{A_2\}", PS) &= B \rightarrow \text{pfp}(A_1, PS) \wedge \neg B \rightarrow \text{pfp}(A_2, PS) \\ \text{pfp}(\text{"if}(B)\{A_1\}", PS) &= B \rightarrow \text{pfp}(A_1, PS) \wedge \neg B \rightarrow PS\end{aligned}$$

**Ex** Soit le programme suivant :

```
/* x = A */
if(x < 0) {
  x = -x;
}
/* x = |A| */
```

$$\begin{aligned}x = A &\rightarrow \text{pfp}(\text{"if}(x < 0)\{x = -x\}", x = |A|) \\ x = A &\rightarrow ((x < 0) \rightarrow \text{pfp}(\text{"x = -x;", } x = |A|) \wedge (x \geq 0 \rightarrow x = |A|)) \\ x = A &\rightarrow ((x < 0) \rightarrow (-x = |A|)) \wedge ((x \geq 0) \rightarrow (x = |A|))\end{aligned}$$

$(A(A < 0) \rightarrow (-A = |A|)) \wedge (A \geq 0 \rightarrow (A = |A|))$  Définition de la valeur absolue  $|\cdot|$ .

D'autres exercices sont disponibles annexe C.4.2 page 41.

### 5.2.4 Calcul du pfp d'une répétition.

La définition d'une répétition est disponible section 2.2.4. Modélisation formelle de la répétition.

```
1 /* PE */
2 initialisation;
3 /* INVARIANT */
4 while(c) {
5   /* c ∧ INVARIANT */
6   corps de boucle;
7   /* INVARIANT */
8 }
9 /* ¬c ∧ INVARIANT */
```

Il y a cinq étapes pour la vérification de la boucle.

1.  $PE \rightarrow \text{pfp}(\text{initialisation}, \text{INVARIANT})$
2.  $c \wedge \text{INVARIANT} \rightarrow \text{pfp}(\text{corps}, \text{INVARIANT})$
3.  $\neg c \wedge \text{INVARIANT} \rightarrow PS$
4.  $\text{INVARIANT} \wedge c \rightarrow f > 0$ <sup>3</sup>
5.  $F = f \wedge \text{INVARIANT} \wedge c \rightarrow \text{pfp}(\text{corps}, F > f)$ <sup>4</sup>

**Ex**

```

/* (N ≥ 1) ∧ (B(N), ≥) */
j = 1;
p = 1;
/* INV = (1 ≤ p ≤ j ≤ N) ∧ (∃k : 0 ≤ k ≤ j - p) ∧ (B[k] = B[k + p - 1])
  ∧ (∀k(0 ≤ k ≤ j - p - 1) → (B[k] ≠ B[k + p])) */
while (j != N) {
  /* INV ∧ c */
  if (B[j-p] == B[j])
    p++;
  j++;
  /* INV */
}
/* ¬c ∧ INV */
/* R = (1 ≤ p ≤ N) ∧ (∃k : 0 ≤ k ≤ N - p) ∧
  (B[k] = B[k + p - 1]) ∧ (∀k(0 ≤ k ≤ N - p - 1) → (B[k] ≠ B[k + p])) */

```

$$\begin{aligned}
 x \geq 1 &\rightarrow \text{pfp}(j = 1, p = 1, \text{Inv}) \\
 x \geq 1 &\rightarrow (0 \leq 1 \leq 1 \leq N) \wedge \exists k(0 \leq k \leq 0(B[k] = B[k])) \wedge \\
 &\quad \forall k(0 \leq k \leq -1 \rightarrow B[k] \neq B[k + 1]) \\
 &\Rightarrow \text{Toujours vrai}
 \end{aligned}$$

$$\begin{aligned}
 C \wedge INV &\rightarrow \text{pfp}(\text{if}(B[j - p] == B[k])p ++; j ++, INV) \\
 C \wedge INV &\rightarrow \text{pfp}(\text{if}(B[j - p] == B[j])p ++, \text{pfp}(j ++, INV)) \\
 C \wedge INV &\rightarrow \text{pfp}(\dots, 1 \leq p \leq j + 1 \leq N \wedge \exists k(0 \leq k \leq j + 1 - p) \\
 &\quad \wedge B[k] = B[k + p - 1] \wedge \\
 &\quad \exists \forall k(0 \leq k < j + 1 - p - 1 \rightarrow B[k] \neq B[k + p])) = A \\
 C \wedge INV &\rightarrow B[j - p] == B[j] \rightarrow \text{pfp}(p ++, A) \wedge B[j - p] \neq B[j] \rightarrow A \\
 C \wedge INV &\rightarrow (B[j - p] \neq B[j] \rightarrow (0 \leq p \leq j + 1 \leq N) \wedge \\
 &\quad \exists k 0 \leq k \leq j + 1 - p \wedge B[k] = B[k + p - 1] \\
 &\quad \wedge \forall k 0 \neq k \leq j - p \rightarrow B[k] \neq B[k - p] \\
 &\rightarrow T \wedge \exists k 0 \leq k < j - p \wedge B[k] = B[k + 1 - p] \wedge \\
 &\quad B[j + 1 - p] = B[j + 1 - p + 1 - p] \wedge \dots
 \end{aligned}$$

3.  $f$  : fonction définie positive :  $f$  est appelée «variante», ceci afin de vérifier que la boucle se termine et n'est donc pas infinie

4.  $f$  est décroissante

**R** Il est conseillé de commencer par l'étape la plus facile, en effet, une étape et nous n'avons pas à effectuer les autres

D'autres exercices sur les répétitions sont disponible annexe [C.4.3](#) page 42.



# Ecriture d'un programme à partir de sa spécification

## Sommaire

<b>6.1</b>	<b>Modèle de boucle</b>	<b>25</b>
------------	-------------------------	-----------

Il s'agit d'écrire un programme à partir de ses spécifications..

```

/* P */
action(N,r);
/* Q */

```

Il permet de définir un modèle de solution :

- Séquence, boucle, recherche linéaire, recherche dichotomique
- Dédit de l'analyse de Q et de P

## 6.1 Modèle de boucle

Il faut déterminer l'invariant et la condition de boucle afin de construire complètement le programme. Nous allons ainsi utiliser la propriété suivante :

$$INV \wedge \neg C \rightarrow Q$$

On pose  $INV \wedge \neg C = Q$ , si cette propriété est vérifiée, alors l'implication sera vérifiée.

### 6.1.1 1<sup>ère</sup> approche : Preuve avec invariant trivial

$Q = \top \wedge Q$   $Q$  étant  $\neg$ condition

1.  $p \rightarrow \text{pfp}(\text{init}, \top) = p \rightarrow \top$  Toujours vrai
2.  $\top \wedge Q \rightarrow \text{pfp}(\text{corps}, \top) = \top \wedge Q \rightarrow \top$
3.  $\top \wedge Q \rightarrow Q$  Toujours vrai<sup>1</sup>

Toute la difficulté est de prouver la terminaison.

#### 6.1.1.1 Exemple


1.  $Q \rightarrow Q$

```

1  /* (x = A) ∧ (y=B) ∧ (z=C) */
2  init
3  /* ⊤ */
4  while ((x >= y) || (y >= z)) {
5      /* ⊤ ∧ (x ≥ y) ∧ (y ≥ z) */
6      if(x >= y)
7          echange(&x, &y);
8
9      if(y >= z)
10         echange(&y, &z);
11     /* \top */
12 }
13 /* ⊤ ∧ (x < y) ∧ (y < z) */
14 /* (x < y) ∧ (y < z)

```

Nous pouvons prendre  $x - z + |A + B + C|$  comme variante

 Prendre un invariant trivial complique la preuve de terminaison et cela réduit l'écriture du programme à la recherche de la variante.

## 6.1.2 2<sup>nd</sup> approche : Élimination de conjoint

$$Q : A \wedge B \wedge C \wedge \dots^2$$

Si on peut trouver une séquence d'initialisation qui permet de vérifier les conjoints de façon simple ; ces conjoints forment l'invariant.

$Q = A \wedge B \wedge E$  on peut écrire  $INV \wedge E$ .

### 6.1.2.1 Exemple

```

1  /* N > 0 */
2  a = N;
3  /* a² ≤ N */
4  while ( a*a > N) {
5      /* a² ≤ N ∧ cond */
6      a = a - 1;
7      /* a² ≤ N */
8  }
9
10 /*
11  * a² ≤ N ≤ (a + 1)² peut aussi être écrit a² ≤ N ∧ N ≤ (a + 1)²
12  * a² ≤ N : INV
13  * N ≤ (a + 1)² : ¬C

```

---

2. A, B et C sont des conjoints

14 `*/`

**R** Cette solution fonctionne, cependant le programme à une complexité linéaire ( $N$ ), celui-ci peut être résolu avec une complexité logarithmique.

### 6.1.3 3<sup>ème</sup> approche : introduction d'une variable dans $Q$

En général,  $Q$  s'écrit  $Q(N)$ . On va le réécrire en introduisant une variable :  $Q(i) \wedge (i = N)$

Ainsi  $Q(i)$  devient l'invariant et  $i = N$  la condition de boucle ( $\neg$ condition).

#### 6.1.3.1 Exemple

```
1  /* N > 0 */
2  f = 1;
3  i = 1;
4  /* f = i! */
5  while (i != N) {
6      ++i;
7      f *= i;
8  }
9
10 /*
11  * f = N !
12  * On le réécrit f = i! ^ i = N
13  */
```

# Transformation d'une spécification récur- sive en un programme itératif

Une approche récursive est équivalente en temps à une approche itérative, cependant la consommation mémoire sera beaucoup plus importante que l'itérative, ceci étant dû à la pile système.

**R** Certains compilateurs peuvent transformer une approche récursive terminale en itératif

Notamment le compilateur Ocaml, gcc le fait également pour les types primitifs (scalaires), pour les structures de données, il n'en est pas capable.

## 7.1 Récursivité terminale

$$G(x) \begin{cases} \text{si} & h(x) \text{ alors } a \\ \text{sinon} & f(x) \oplus G(t(x)) \end{cases}$$

**R** le  $\oplus$  est l'opérateur de combinaison intermédiaire

SI nous avons :

- $h(x)$  fonction booléenne
- $\oplus$  associatif avec un élément neutre  $e$  (à gauche)

Alors le programme suivant est correct.

```

1  /* PE: T */
2  x = X;
3  r = e;
4  /* INV: G(X) = r ⊕ G(x) */
5  while (!(h(x))) {
6      r = r ⊕ f(x);
7      x = t(x);
8  }
9  r = r ⊕ a;
10 /* PS: G(X) = r */

```

**Ex** Écrire un programme qui calcule  $Y = x^N$  avec  $N$  entier et  $N \geq 0$ .  $X$  et  $Y$  des réels.

1. Spécification récursive
2. Programme

$$\text{puissance}(X, N) = \begin{cases} \text{si } N = 0 & 1 \\ \text{sinon} & x \times \text{puissance}(X, N - 1) \end{cases}$$

- $\oplus$  : \*réels
- $e$  : 1.0
- $h(X, N) : N == 0$
- $f(X, N) : X$
- $t(X, N) : (X, N - 1)$

```
/* PE:  $\top$  */
x = X;
r = 1;
/* INV:  $G(X) = r \oplus G(x)$  */
while (!(N == 0)) {
    r = r * x;
    N = N - 1;
}
r = r  $\oplus$  a;
/* PS:  $G(X) = r$  */
```

# Troisième partie

## Annexes

---

# Glossaire

---

**Compilation** Un compilateur est un programme informatique qui transforme un code source écrit dans un langage de programmation (le langage source) en un autre langage informatique (le langage cible).

**Interprétation** Analyse, traduit et exécute un programme écrit dans un langage informatique. De tels langages sont dits langages interprétés.

L'interpréteur est capable de lire le code source d'un langage sous forme de script, habituellement un fichier texte, et d'en exécuter les instructions après une analyse syntaxique du contenu. Généralement ces langages textuels sont appelés des langages de programmation. Cette interprétation conduit à une exécution d'action ou à un stockage de contenu ordonné par la syntaxe textuelle.

**Édition de liens** Lors d'un développement informatique, l'édition des liens est un processus qui permet de créer des fichiers exécutables ou des bibliothèques dynamiques ou statiques, à partir de fichiers objets.

---

# Liste des codes sources

---

2.1	Syntaxe de déclaration de variable . . . . .	8
2.2	Syntaxe d'un bloc . . . . .	8
2.3	Syntaxe des actions . . . . .	9
2.4	Syntaxe d'une structure de contrôle . . . . .	9
2.5	Syntaxe de répétition . . . . .	9
2.6	Syntaxe d'une affectation . . . . .	9
2.7	Syntaxe de l'appel de printf . . . . .	10
2.8	Syntaxe de l'appel de scanf . . . . .	10
2.9	Syntaxe de déclaration d'un tableau . . . . .	11
2.10	Syntaxe d'un sous programme . . . . .	11
2.11	Point d'entrée du programme : le main . . . . .	11
2.12	Exemple d'instructions pré-processeurs . . . . .	12
5.1	Exercice 3 . . . . .	20
	annexes/exemplePfpAfect.c . . . . .	21
	annexes/exemplePfpSeq.c . . . . .	21
	annexes/exemplePfpSelect.c . . . . .	22
	annexes/exemplePfpBoucles.c . . . . .	23
	annexes/exo7.c . . . . .	29
C.1	Exercice 1 – Code du programme . . . . .	34
C.2	Exercice 3 . . . . .	36
C.3	Exercice 4 . . . . .	38
C.4	Exercice 5 . . . . .	39
C.5	Exercice 6 . . . . .	40
	annexes/exercices/complet.c . . . . .	49
	annexes/exercices/complet2.c . . . . .	50



---

# Exercices

---

## C.1 Initiation à la programmation

### C.1.1 Exercice 1

Écrire un programme qui lit une série de 10 valeurs et affiche la position du minimum et du maximum de la série.

#### C.1.1.1 Étape 1 : Analyser le problème

1. Lire les valeurs
2. calculer les min et max
3. afficher le résultat

#### C.1.1.2 Étape 2 : Spécifier les sous-problèmes

Identifier les entrée, les sorties et leurs propriétés.

##### **LireLesValeurs**

**Entrée** Nombre, les valeurs à lire

**Sortie** Tableau contenant les valeurs lues

##### **CalculerMinEtMax**

**Entrée** Le tableau des valeurs et le nombre de valeur

**Sortie** Position, min et max.

## C.1.1.3 Étape 3 : Le code

```
1 #include <stdlib.h>
2 #define N 100
3
4 void read (int nb, int* tab) ;
5 void calculerMinMax(int nb, int* t, int *pmin, int *vmin, int *pmax
   , int *vmax);
6 void read (int nb, int* tab) ;
7
8 int main (int argc, char** argv) {
9     int pmin, vmin, pmax, vmax;
10    int tab[N];
11    read(10, tab);
12    calculerMinMax(10, tab, &pmin, &vmin, &pmax, &vmax);
13    printf(...);
14 }
15 // un tableau est un pointeur sur le premier élément
16 // peut aussi être écrit int tab[N]
17 void read (int nb, int* tab) {
18     int i;
19     for(i=0; i < nb ; ++i) {
20         scanf('%d', tab+i);
21     }
22 }
23
24 void calculerMinMax(int nb, int *tab, int *pmin, int *vmin, int *
   pmax, int *vmax) {
25     *pmin = 0;
26     *pmax = 0;
27     *vmin = t[pmin];
28     *vmax = t[pmax];
29
30     for(; --nb = b > 0;) {
31         if(tab[nb] < *vmin) {
32             *vmin = tab[nb];
33             *pmin = nb;
34         }
35         if(tab[nb] > *vmax) {
36             *vmax = tab[nb];
37             *pmax = nb;
38         }
39     }
40 }
```

Listing C.1 – Exercice 1 – Code du programme

## C.2 Spécification

### C.2.1 Exercice 7

Écrire la spécification d'un programme qui dans un tableau  $T$  de  $N$  entiers calcul le nombre  $n$  de nombre positifs dans le tableau.

- $N > 0$
- `calculeNbPos(T, N, n)`
- $(0 \leq n \leq N) \wedge (n = \nu I : 0 \leq I < NT[I] \geq 0)$

#### C.2.1.1 Exercice 8

Soit  $T$  un tableau croissant (non strict) de  $N$  entier et  $X$  un entier.

Spécifier un programme qui calcule la position de la dernière occurrence de  $T$  inférieure ou égale à  $X$  avec  $T[0] \leq X < T[N - 1]$

- $(N > 1) \wedge (T[0] \leq X) \wedge (X < T[N - 1]) \wedge (\forall I : 0 \leq I < N - 1 \rightarrow T[I] \leq T[I + 1])$
- `searchPosition(T, N, X, p)`;
- $(0 < p < N - 1) \wedge (T[p] \leq X) \wedge (T[p + 1] > X)$

**R** Dans la suite du cours, nous pourrons utiliser un raccourci afin de savoir si un tableau est trié par ordre croissant :  $(T, N, \leq)$   
Celle-ci pourra être utilisée dans la copie à condition qu'elle soit définie au préalable.

### C.2.2 Exercice 8

Soit un tableau  $T$  non vide de  $N$  entiers. Écrire la spécifications du programme qui calculent :

- La première position de la valeur max de  $T$
- La dernière position de la valeur max de  $T$

#### C.2.2.1 Calcule de la première position

- $N > 0$
- `searchFirstPosition(T, N, f)`;
- $(\forall I : 0 \leq I < f \rightarrow T[I] < T[f]) \wedge (\forall I(f \leq I < N) \rightarrow (T[I] \leq T[f]))$

#### C.2.2.2 Calcule de la dernière position

- $N > 0$
- `searchLastPosition(T, N, l)`;
- $(\forall I : 0 \leq I < l \rightarrow T[I] \leq T[l]) \wedge (\forall I(l < I < N) \rightarrow (T[I] < T[l]))$

### C.2.3 Exercice 9

Écrire la spécification d'un programme qui, dans un tableau T de N entiers tous différents cherche la position d'une valeur X si elle existe ou retourne N si elle n'existe pas.

- $(N \geq 0) \wedge (\forall I : 0 \leq I < N \rightarrow (\forall (I, J) : 0 \leq I < N \wedge (0 \leq J < N) \rightarrow T[I] = T[J] \leftrightarrow (I = J))$ <sup>1</sup>
- `search(T, N, x)`
- $(0 \leq p < N \wedge T[p] = X) \vee (p = N) \leftrightarrow \forall I (0 \leq I < N \rightarrow T[I] \neq X)$

### C.2.4 Exercice 10

Spécifier un programme qui, dans un tableau T de N éléments trié par ordre croissant non strict retourne la longueur du plus grand plateau<sup>2</sup>.

- $(T, N, \leq) \wedge N > 0$
- `longueurPlusGrandPlateau(T, N, l);`
- $(1 \leq l \leq N) \wedge (\exists I : 0 \leq I < N - l) \wedge (T[I] = T[I + l - 1])$

### C.2.5 Exercice 11

Spécifier un programme qui, dans un tableau de N entiers calcule le nombre de doublons : un doublon est une succession de 2 nombres identiques.

- $N > 0$
- `calculeDoublons(T, N, n);`
- $n = \nu I : 0 \leq I < N \wedge T[I] = T[I + 1]$

**R** Dans le cas où deux doublons ne sont pas forcément côte à côte, le prédicat de sortie deviendrait :

$$n = \sum_{I=0}^{N-1} \nu J : I < J < N \wedge T[J] = T[I]$$

## C.3 Tableau de situation

### C.3.1 Exercice 2

```
1 typedef tab int [3];
2 tab T;
3 int j = 2;
```

1. Cela peut aussi s'écrire  $N \leq 0) \wedge (\forall I (0 \leq I < N) \rightarrow \forall J : J \neq I \wedge 0 \leq J < N \rightarrow T[I] \neq T[J])$
2. Un plateau est quand il y a plusieurs fois le même caractère

```

4
5 void modifier1(int x) {
6     j++;
7     x= 1;
8 }
9
10 void modifier2 (int *x) {
11     j++;
12     *x = 1;
13 }
14
15 void modifier3 (tab b) {
16     T[0] = b[0] + b[j] + b[2] + b[3] + b[4];
17     t[1] = b[0] + b[1] + b[2] + b[3] + b[4];
18 }
19
20 int main(int argc, char** argv) {
21     int i;
22     for(i=0; i < j; ++i) {
23         T[i] = 0;
24     }
25     modifier1(T[j]); // ... 1
26     modifier2(&T[j]) ; // ... 2
27
28     for(i = 0; i < j; ++i) {
29         T[i] = 1;
30     }
31     modifier3(T);
32
33     return 0;
34 }

```

Listing C.2 – Exercice 3

Point d'arrêt	T	j	i	n
1	0,0,0,0,0	3	5	1
2	0,0,0,1,0	4	3	@
3	5,9,1,1,1	4	5	

## C.3.2 Effets de bords

## C.3.2.1 Exercice 4

```

1  int y;
2
3  int f(int x) ;
4
5  int main(int argc, char *argv[]) {
6      int i, z;
7      y = 10;
8      i = 1; //...2
9      z = f(i) + y; //...3
10
11     return z;
12 }
13
14 int f(int x) {
15     int t = x;
16
17     ++y;
18     ++t;
19
20     return (t+y); //...1
21 }

```

Listing C.3 – Exercice 4

Point d'arrêt	y	x	t	i	z	f
2	10	/	/	1		
1	11	1	2	1	/	13
3	11	/	/	1	24	/

## C.3.2.2 Exercice 5

```

1  int i;
2  int j;
3
4  void pr (int *x, int *y, int *z) ;
5
6  int main(int argc, char *argv[]) {
7      i = 3;
8      j = 7; // ... 3
9      pr(&i, &i, &j); // ... 4
10
11     i = 3;
12     j = 7; // ... 5
13     pr(&j, &j, &i); /// ...6
14 }
15
16 void pr (int *x, int *y, int *z) {
17     *y = i + *x; // ... 1
18     *z = *x + *y; // ... 2
19 }

```

Listing C.4 – Exercice 5

Point d'arrêt	i	j	x	y	z
3	3	7	/	/	/
1			@i	@i	@j
2	6	12	@i	@i	@j
4	6	12	/	/	/
5	3	7	/	/	/
1	3	10	@j	@j	@i
2	20	10	@j	@j	@i

## C.3.2.3 Exercice 6

```

1  int i;
2
3  int f(int a, int b){
4      i = i + a;
5
6      return (a + b); // 1
7  }
8
9  int main(int argc, char *argv[]) {
10     int j, x;
11
12     i = 10;
13     j = 10; // 2
14     x = f(i, j); // 3
15 }

```

Listing C.5 – Exercice 6

Point d'arrêt	i	j	x	a	a	f
2	10	40	/	/	/	/
1	20	40	/	10	40	50
3	20	40	50	/	/	/

## C.4 Plus faible précondition

## C.4.1 Séquence

## C.4.1.1 Exercice 1

```

1  /* f = i! */
2  f = f * (i + 1);
3  i = i + 1;
4  /* f = i! */

```

$$\begin{aligned}
 f = i! &\rightarrow \text{pfp}("f = f \times (i + 1); i = i + 1;", f = i!) \\
 f = i! &\rightarrow \text{pfp}("f = f \times (i + 1);", \text{pfp}("i = i + 1", f = i!)) \\
 f = i! &\rightarrow \text{pfp}("f = f \times (i + 1);", f = (i + 1)!)
 \end{aligned}$$



## C.4.1.2 Exercice 2

```

1  /* (x = A) ∧ (y = B) ∧ (z = C) */
2  x = x + y + z;
3  z = x - y - z;
4  y = x - y - z;
5  x = x - y - z;
6  /* (x = B) ∧ (y = C) ∧ (z = A) */

```

$$PE \rightarrow \text{pfp}("x = x + y + z; z = x - y - z; y = x - y - z; x = x - y - z;", \\ (x = B) \wedge (y = C) \wedge (z = A))$$

$$PE \rightarrow \text{pfp}("x = x + y + z, z = x - y - z; y = x - y - z;", \\ \text{pfp}("x = x - y - z; (x = B) \wedge (y = C) \wedge (z = A)"))$$

$$PE \rightarrow \text{pfp}("x = x + y + z, z = x - y - z; y = x - y - z;", x - y - z = B) \wedge y = C \wedge z = A$$

$$PE \rightarrow \text{pfp}("x = x + y + z, z = x - y - z", (y = B) \wedge (x - y - z = C) \wedge z = A)$$

$$PE \rightarrow \text{pfp}("x = x + y + z", (y = B) \wedge (z = C) \wedge (x - y - z = A))$$

$$PE \rightarrow (y = B) \wedge (z = C) \wedge (x = A) \text{ Vrai parceque } p \rightarrow p = \text{vrai}$$

## C.4.2 Sélection

## C.4.2.1 Exercice 1

```

1  /* x = A ∧ y = B */
2  if (A < B) {
3      x = A;
4      y = B;
5  } else {
6      x = B;
7      y = A;
8  }
9  /* x ≤ y */

```

$$PE \rightarrow \text{pfp}("if(A < B)\{x = A; y = B\}else\{x = B; y = A\}", x \leq y)$$

$$PE \rightarrow ((A < B) \rightarrow \text{pfp}("x = A; y = B", x \leq y)) \wedge ((A \geq B) \rightarrow \text{pfp}("x = B; y = A", x \leq y))$$

$$PE \rightarrow (((A < B) \rightarrow (A \leq B)) \wedge (A \geq B) \rightarrow (B \leq A))$$

Vrai par définition. A est toujours inférieur à B.  $(A \geq B) \rightarrow (B \leq A)$  est une Tautologie.

## C.4.2.2 Exercice 2

$$\text{pfp}("if(x \geq y)\{z = x;\}else\{z = y;\}", z = \max(x, y))$$

$$\begin{aligned}
x \geq y &\rightarrow \text{pfp}(\text{"if}(x \geq y)\{z = x;\}\text{else}\{z = y\}\text{"}, z = \max(x, y)) \\
x \geq y &\rightarrow \text{pfp}(\text{"z = x"}, z = \max(x, y)) \\
x < y &\rightarrow \text{pfp}(\text{"z = y"}, z = \max(x, y)) \\
x \geq y &\rightarrow x = \max(x, y) \\
x < y &\rightarrow y = \max(x, y)
\end{aligned}$$

C'est une tautologie par définition de  $\max$ .

### C.4.2.3 Exercice 3

$$\text{pfp}(\text{"if}(x > y)\{if(x \% 2 == 0)\{x = x - 2\}\}\text{else}\{y = y - 1;\}\text{"}, y - 2 < x);$$

$$\begin{aligned}
(x > y) &\rightarrow \text{pfp}(\text{"if}(x \% 2 == 0)x = x - 2;\text{"}, y - 2 < x) \wedge ((x \leq y) \rightarrow \text{pfp}(\text{"y = y - 1;\text{"}, y - 2 < x)) \\
(x > y) &\rightarrow (((x \% 2 == 0) \rightarrow \text{pfp}(\text{"x = x - 2"}, y - 2 < x) \wedge \\
&\quad (x \% 2 \neq 0) \rightarrow (y - 2 < x)) \wedge ((x \leq y) \rightarrow (y - z < x)) \\
(x > y) &\rightarrow ((x \% 2 == 0) \rightarrow ((y - 2) < (x - 2)) \wedge ((x \% 2 \neq 0) \rightarrow (y - 2 < x))) \wedge (x \leq y \rightarrow y - z < x) \\
(x > y) &\rightarrow (x \leq y \rightarrow y - z < x)
\end{aligned}$$

## C.4.3 Boucles

### C.4.3.1 Exercice 1

```

1  /* N ≥ 0 */
2
3  /* P Tableau de polynôme
4  * N Degré du polynôme
5  * X Point ou je veux évoluer le polynôme
6  * r Résultat du polynôme
7  */
8  calculPolynome(P,N,X,r);
9  /* r = ∑k=0N A[k]Xk */

```

```

1  /* N ≥ 0 */
2
3  /* INV = (0 ≤ i ≤ N) ∧ (r = ∑k=0i A[k]xk) ∧ (y = xi) */
4  while(c) {
5  /* c \wedge INV */
6  corps;
7  /* INV */
8  }
9  /* ¬c ∧ INV */
10 /* p = ∑k=0N A[k]xk */

```

**INVARIANT**

$$(0 \leq i \leq N) \wedge (r = \sum_{k=0}^i A[k]x^k) \wedge (y = x^i)$$

**Initialisation**

## 1. INIT 0

```
i = 0;
p = 0;
y = 1;
```

$$PE \rightarrow \text{pfp}(\text{init } 0, \text{INV})$$

$$N \geq 0 \rightarrow \text{pfp}("i = 0; p = 0", p = \sum_k^i 0A[k]x^k \wedge 1 = x^i);$$

$$N \geq 0 \rightarrow \text{pfp}("i = 0", 0 = \sum_{k=0}^i A[k]x^k \wedge 1 = x^i)$$

$$N \geq 0 \rightarrow 0 = \sum_{k=0}^0 A[k]x^k \wedge 1 = x^0$$

$$(N \geq 0 \rightarrow 0 = A[0] \wedge 1 = 1) \Rightarrow \text{Faux et Vrai, donc le programme est Faux}$$

## 2. INIT 1

```
i = 0;
p = A[0];
y = 1;
```

$$PE \rightarrow \text{pfp}(\text{init } 1, \text{INV})$$

$$N \geq 0 \rightarrow \text{pfp}("i = 0; p = A[0]", p = \sum_k^i 0A[k]x^k \wedge 1 = x^i);$$

$$N \geq 0 \rightarrow \text{pfp}("i = 0", A[0] = \sum_{k=0}^i A[k]x^k \wedge 1 = x^i)$$

$$N \geq 0 \rightarrow A[0] = \sum_{k=0}^0 A[k]x^k \wedge 1 = x^0$$

$$(N \geq 0 \rightarrow A[0] = A[0] \wedge 1 = 1) \Rightarrow \text{Vrai et Vrai, donc le programme est correct}$$

L'initialisation nécessaire est donc init 1.

**Boucles**

## 1.

```
while(i < N) {
  ++i;
  p = p + A[i-1] * y;
  y = y * X;
}
```

Étape numéro 3 (Cf section 5.2.4 page 23)

$$\neg C \wedge INV \rightarrow PS$$

$$(i > N) \wedge (p = \sum_{k=0}^i A[k]x^k) \wedge (y = x^i) \rightarrow p = \sum_{k=0}^N A[k]x^k$$

$$(i > N) \wedge (p = \sum_{k=0}^i A[k]x^k) \wedge (y = x^i) \rightarrow p = \sum_{k=0}^N A[k]x^k \wedge (i = N)$$

$$(i > N) \rightarrow (i = N) \Rightarrow \text{C'est donc faux.}$$

**R** Nous sommes partis de la conclusion et avons essayé de faire apparaître notre hypothèse en partie droite.

```
while( i != N ) {
  ++i;
  p = p + a[i-1] * y;
  y = y * X
}
```

Étape numéro 3 (Cf section 5.2.4 page 23)

$$\neg C \wedge INV \rightarrow PS$$

$$(i \neq N) \wedge (p = \sum_{k=0}^i A[k]x^k) \wedge (y = x^i) \rightarrow p = \sum_{k=0}^N A[k]x^k$$

$$(i \neq N) \wedge (p = \sum_{k=0}^i A[k]x^k) \wedge (y = x^i) \rightarrow p = \sum_{k=0}^N A[k]x^k \wedge (i = N)$$

$$(i \neq N) \rightarrow (i = N) \Rightarrow \text{C'est donc vrai.}$$

Étape numéro 2 (Cf section 5.2.4 page 23)

$$C \wedge INV \rightarrow \text{pfp}(\text{corps}, INV)$$

$$C \wedge INV \rightarrow \text{pfp}("i++; p = p + A[i-1] * y;", (p = \sum_{k=0}^i A[k]x^k) \wedge (y * x = x^i) \wedge (0 \leq i \leq n))$$

$$C \wedge INV \rightarrow \text{pfp}("i++;", (p + A[i-1] * y = \sum_{k=0}^i A[k]x^k) \wedge (y * x = x^i) \wedge (0 \leq i \leq n))$$

$$(i \neq N) \wedge (0 \leq i \leq N) \wedge$$

$$(p = \sum_{k=0}^i A[k]x^k) \wedge (y = X^i) \rightarrow (p + A[i] * i = \sum_{k=0}^{i+1} A[k]x^k) \wedge (y \times x = x^{i+1}) \wedge (0 \leq i+1 \leq N)$$

$$(i \neq N) \wedge (0 \leq i \leq N) \wedge$$

$$(p = \sum_{k=0}^i A[k]x^k) \wedge (y = X^i) \rightarrow A[i] \times x^i = p + A[i+1]x^{i+1} \wedge x^i \times = x^{i+1} \wedge \text{Tautologie}$$

$$(i \neq N) \wedge (0 \leq i \leq N) \wedge$$

$$(p = \sum_{k=0}^i A[k]x^k) \wedge (y = X^i) \rightarrow \text{Faux} \wedge \text{Tautologie} \wedge \text{Tautologie} \Rightarrow \text{Le programme est donc faux}$$

3.

```

while(i == N) {
  ++i;
  p = p + (A[i] * y * X);
  y = y * X;
}

```

Ce programme effectue la correction de l'erreur détectée plus haut. Il est correct.

### C.4.3.2 Exercice 2 : Suite de Fibonacci

**R** Le rang  $n$  de la suite de Fibonacci est défini comme suit :

$F_n = 1$  si  $n = 0$  ou  $n = 1$

$F_n = F_{n-2} + F_{n-1}$  si  $n > 1$

```

/* N ≥ 0 */
i = 0;
a = 1;
b = 1;
/* INV = (F_i = a) ∧ (F_{i+1} = b) ∧ (0 ≤ i ≤ N) */
while(c) {
  /* c ∧ INV */
  i++;
  b += a;
  a = b - a;
  /* INV */
}
/* ¬c ∧ INV */
/* F_N = {...} */

```

**R** Il est conseillé de commencer par l'étape la plus facile, en effet, une étape et nous n'avons pas à effectuer les autres

#### Etape 1

$$\begin{aligned}
 PE &\rightarrow \text{pfp}(\text{"init"}, INV) \\
 N \geq 0 &\rightarrow (F_{i+1} = 1) \wedge (F_i = 1) \wedge (i = 0) \\
 N \geq 0 &\rightarrow (F_1 = 1) \wedge (F_0 = 1) \Rightarrow \text{Tautologie par définition}
 \end{aligned}$$

#### Etape 3

$$\begin{aligned}
 (i = N) \wedge (F_i = a) \wedge (F_{i+1} = b) \wedge (0 \leq i \leq N) &\rightarrow F_N = a \\
 (F_n = a) \wedge (F_{N+1}) \wedge (0 \leq N) &\rightarrow F_N = a \Rightarrow \text{Tautologie}
 \end{aligned}$$

**Etape 2**

$$\begin{aligned}
(i \neq N) \wedge (F_i = a) \wedge (F_{i+1} = b) \wedge (0 \leq i \leq N) &\rightarrow \text{pfp}(" \dots ", F_i = b - a \wedge F_{i+1} = b \wedge \dots) \\
&\rightarrow \text{pfp}(" \dots ", F_i = b + a - a \wedge F_{i+1} = b + a) \\
&\rightarrow F_i + 1 = b \wedge F_{i+2} = b + a \wedge i + 1 \leq N \\
&\rightarrow T \wedge T_{\text{par définition}} \wedge T
\end{aligned}$$

**Etape 4** Tester une existence de  $f > 0$  avant l'exécution du corps :  $c \wedge INV \rightarrow f > 0$ .

$$\begin{aligned}
(i \neq N) \wedge (F_i = a) \wedge (F_n - 1 = b) \wedge (0 \leq i \leq N) &\rightarrow f > 0 \\
f = N - i \Rightarrow N \geq 0 &\text{ donc Vrai.}
\end{aligned}$$

**Étape 5**

$$\begin{aligned}
f = F \wedge c \wedge INV &\rightarrow \text{pfp}("corps", f < F) \\
N - i = F \wedge c \wedge INV &\rightarrow \text{pfp}("corps", N - i < F) \\
N - i = F \wedge c \wedge INV &\rightarrow N - (i + 1) < F \\
&\rightarrow N - i - 1 < F \\
&\rightarrow F - 1 < F \Rightarrow \text{Tautologie}
\end{aligned}$$

## C.4.3.3 Exercice 3 : plus grand plateau

**R**  $(B(N), \geq)$  signifie le tableau est ordonné dans l'ordre croissant non strict

## C.4.3.4 Exercice 3 : Définition variante

$$\begin{aligned}
C \wedge INV &\rightarrow f > 0 \\
J \neq N \wedge INV &\rightarrow (f = N - j) > 0 \Rightarrow \text{Vrai}
\end{aligned}$$

$$\begin{aligned}
N - j = F \wedge C \wedge INV &\rightarrow \text{pfp}(corps, N - j < F) \\
N - j = F \wedge C \wedge INV &\rightarrow \text{pfp}(\text{if}(\dots)) \dots, \text{pfp}(j++, N - j < F) \\
N - j = F \wedge C \wedge INV &\rightarrow B[j - p] = B[j] \rightarrow N - j - 1 < F \wedge B[j - p] = B[j]
\end{aligned}$$

$$N - j = F \wedge C \wedge INV \rightarrow N - j - 1 < F \wedge F - 1 < F$$

## C.4.3.5 Exercice 4

```

1  /*  $A \leq 0 \wedge B \geq 0$  */  $x = A; y = B; z = 1; /* (z = x^y = A^B) \wedge y \geq 0$  */
2
3
4
5
6
7  while(y != 0) {
8      /*  $z * x^y = A^B \wedge y \geq 0$  */
9      while(y \% 2 == 0) {
10         y = y / 2 ;
11         x = x*x;
12     }
13     /*  $z = x^y = A^B \wedge y > 0 \wedge y \% 2 \neq 0$  */
14     y--;
15     z = z * x;
16     /*  $z = x^y = A^B \wedge y \geq 0$  */
17 }
18 PS : /*  $z = A^b$  */

```

## C.5 Écriture d'un programme à partir de sa preuve

### C.5.1 Exercice 1 : Dichotomie

```

1  /*  $N > 0$  */
2  a = 1;
3  b = N;
4  while(b != a + 1) {
5      m = (a+b)/2;
6      if(0 <= N-m*m) {
7          a = m;
8      } else {
9          b = m;
10     }
11 }
12 /*
13  * Réécriture :  $a^2 \leq N \wedge N \leq b^2 \wedge (b = a + 1)$ 
14  *  $a^2 \leq N \wedge N \leq (A + 1)^2$ 
15  */

```

Variante :  $b - a$  : taille de l'intervalle. Fonction décroissante.

### C.5.2 Exercice 2

```

1  /*  $N \leq 2 \wedge B(0..N-1, \leq) \wedge B[0] \leq X \wedge B[N-1] > X$  */
2  cherchep(B, N, X, p);

```

```

3  p = 0;
4  /* 0 ≤ p ≤ i-2 ∧ B(p) ≤ x ∧ B[p+1] > x */
5  while (B[p+1] <= X) {
6      ++p;
7  }
8  /*
9   * 0 ≤ p ≤ N-2 ∧ B[p] ≤ x < B[p+1]
10  */

```

Variante :  $N - i$

## C.6 Exercices complets

### C.6.1 Nombre de plateaux de tailles maximales

Écrire un programme qui calcule le nombre de plateaux de taille maximale dans un tableau de  $N$  entiers.

$T$  est croissant non strict

1. Comprendre le problème
  - Prendre des exemples
  - Identifier les entrées et les sorties
2. Spécifier
3. Écrire le programme
4. Preuve du programme

### C.6.2 Comprendre le problème

1	1	1	2	3	3	3	4
---	---	---	---	---	---	---	---

TABLE C.1 – 2 plateaux de taille 3

1	1	2	3	3	3	4
---	---	---	---	---	---	---

TABLE C.2 – 1 plateau de taille 3

1	2	3	4	5	6	7	8
---	---	---	---	---	---	---	---

TABLE C.3 – 8 plateaux de taille 1

**En entrée** : le tableau  $T$  de longueur  $N$ .

**En sortie** : le nombre de plateau de taille  $t$ .



### C.6.3 Spécification

```

1  /* (N > 0) ∧ T(0..N, ≤) */
2  nbPlateauMax(T,N,p,nb);
3  /* nb ≥ 1 ∧ nb = νI : 0 ≤ I ≤ N - p ∧ T[I] = T[I+p-1] ∧
4   * ∀ J : 0 ≤ J ≤ N-p-1 → T[J] ≠ T[J+p]
5   */

```

### C.6.4 Écrire le programme

```

1  /* PE : (N > 0) ∧ T(0..N, ≤) */
2  int i  = 1;
3  int nb = 1;
4  int p  = 1;
5
6  /* INV= 1 ≤ p ≤ i ∧ nb ≥ 1 ∧ nb = νI : 0 ≤ I ≤ i - p
7   * ∧ T[I] = T[I+p-1] ∧
8   * ∀ J : 0 ≤ J ≤ i-p-1 → T[J] ≠ T[J+p]
9   */
10 while(i != N) {
11     /* INV ∧ C*/
12     if (T[i] == T[i-p]) {
13         np = 1;
14         ++p;
15     } else if (T[i] == T[i-p+1]) {
16         ++np;
17     }
18     ++i;
19     /* INV */
20 }
21
22 /* On remplace une constante par une variable ainsi on a ¬C ∧ INV
23 * PS : (i = N) ∧ nb ≥ 1 ∧ nb = νI : 0 ≤ I ≤ i - p
24 * ∧ T[I] = T[I+p-1] ∧
25 * ∀ J : 0 ≤ J ≤ i-p-1 → T[J] ≠ T[J+p]
26 */
27
28 /* PS : 1 ≤ p ≤ i ∧ nb ≥ 1 ∧ nb = νI : 0 ≤ I ≤ N - p ∧ T[I] = T[I+p
29   -1] ∧
30 * ∀ J : 0 ≤ J ≤ N-p-1 → T[J] ≠ T[J+p]
31 */

```

## C.6.5 Écrire un programme qui calcule la somme préfixée d'un tableau

## C.6.5.1 Comprendre le problème

$\Sigma$ préfixé			i	
a	b	c	d	...

Ce qui pourrait donner :

$\Sigma$ préfixé			i	
0	a	a+b	...	$\sum_{i=0}^{N-1} T[i]$

## C.6.5.2 Spécifications

```

1  /* N > 0 */
2  prefixSum(N, T, t, s);
3  /* t[0] = 0  $\wedge$  ( $\forall I \leq I < N \rightarrow T[I] = T[I-1] + T[I-1]$ )
4     *  $\wedge$  s = t[N-1]+T[N-1]
5     */

```

## C.6.5.3 Programme

```

1  /* N > 0 */
2  s = T[0];
3  T[0] = 0;
4  i = 1;
5  while(i != N) {
6      /* INV  $\wedge$  c */
7      s += T[i];
8      T[i] = s - T[i];
9      ++i;
10     /* INV */
11 }
12 /* i = N  $\wedge$  t[0] = 0  $\wedge$  ( $\forall I \leq i < i \rightarrow T[I] = T[I-1] + T[I-1]$ )
13    *  $\wedge$  s = t[i-1]+T[i-1]
14    *
15 /* t[0] = 0  $\wedge$  ( $\forall I \leq I < N \rightarrow T[I] = T[I-1] + T[I-1]$ )
16    *  $\wedge$  s = t[N-1]+T[N-1]
17    */

```

## C.7 Transformation d'une spécification récursive en un programme itératif

### C.7.1 $N^2$

Ecrire un programme qui calcule  $N^2$  pour  $N$  entier.

$$N^2 = \sum_{i=1}^N (2 \times i) - 1$$

$$\text{carre}(N) \begin{cases} \text{Si } \overbrace{N == 0}^{h(N)} & \overbrace{0}^a \\ \text{Sinon} & \underbrace{(2 * N) - 1}_{f(N)} + \underbrace{\text{carre}(N - 1)}_{t(N)} \end{cases}$$

```

1 x= N;
2 r = 0;
3 while (N != 0) {
4     r += (N << 1) - 1;
5     N = N - 1;
6 }

```

### C.7.2 Palindrome

Écrire un programme qui détermine si un texte de  $N$  caractères est un palindrome.

$$\text{palindrome}(T, D, F) \begin{cases} \text{Si } F - D == 0 & \text{Vrai} \\ \text{Si } F - D == 1 & T[D] == T[F] \end{cases} \bigg) F - D \leq 1 T[D] == T[F]$$

$$\text{Sinon } \text{palindrome}(T, D + 1, F - 1) \underbrace{\bigwedge}_{\oplus} T[D] == T[F]$$

- e : true
- h :  $F - D \leq 1$
- f :  $T[D] == T[F]$
- t :  $(T, D+1, F-1)$

```

1 /* x = (T, D, F) */
2 r= true;
3 while (!(F-D <= 1) && r) {
4     r = r && T[D] == T[F];
5     ++D;
6     --F;
7 }
8 r = r && (T[D] == T[F]);

```