

# Algorithmie en langage C

---

Semestre 3



# Table des matières

<b>1</b>	<b>Paradigmes de programmation</b>	<b>4</b>
1.1	Programmation fonctionnelles . . . . .	4
1.2	Programmation déclarative . . . . .	4
1.3	Programmation Impérative . . . . .	5
<b>2</b>	<b>Programmation impérative en C</b>	<b>6</b>
2.1	Description de l'organisation des données en mémoire . . . . .	6
2.2	Code syntaxe . . . . .	6
2.3	Structure d'une programme en C . . . . .	9
2.4	La compilation . . . . .	10
<b>3</b>	<b>Méthodologie de la programmation impérative</b>	<b>11</b>
3.1	Programmation "en petit" . . . . .	11
3.2	Programmation en large . . . . .	11
3.3	Développement d'un algorithme . . . . .	11
3.4	Étape 1 : comprendre le problème . . . . .	11
3.5	Étape 2 : Spécification du programme : spécification formelle . . . . .	12
3.6	Étape 3 : Donner un modèle de solution . . . . .	12
3.7	Étape 4 : Programmer et unifier le programme . . . . .	12
<b>4</b>	<b>Spécification d'un programme</b>	<b>14</b>
4.1	Mots clés à utiliser dans les prédicats . . . . .	14
4.2	Écriture de la spécification . . . . .	14

<b>5</b>	<b>Vérification formelle de programmes</b>	<b>15</b>
5.1	Système de réécriture Pfp . . . . .	15
5.2	Calcul de pfp d'une affectation . . . . .	15
5.3	Calcul du pfp d'une séquence . . . . .	16
<b>A</b>	<b>Glossaire</b>	<b>18</b>
<b>B</b>	<b>Exercices</b>	<b>19</b>
B.1	Initiation . . . . .	19
B.2	Tableau de situation . . . . .	21
B.3	Effets de bords . . . . .	23
B.4	Spécification . . . . .	25
<b>C</b>	<b>Liste des codes sources</b>	<b>28</b>

# Paradigmes de programmation

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** <sup>1</sup> ou interprétés <sup>2</sup>. Ce paradigme

**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.

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

---

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

## 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

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

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

Un bloc transforme une séquence en action.

---

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

## 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 executé, sinon action 2 est executé.

## 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.

## 2.2.6 Opérateurs de base sur les types

= Affectation

---

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)



`+`, `-`, `/`, `*` Opérateurs arithmétiques.  
`&&`, `||`, `!` Opérateurs logiques  
`==`, `!=`, `<`, `>`, `<=`, `>=` Opérateurs booléens  
`++i`, `i++`, `--i`, `i--` Opérateur unaires d'incrémentement.

## 2.2.7 Opérateurs d'entrées / sorties

### Ecriture

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

Listing 2.7 – Syntaxe de l'appel de printf

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


- '**d**' Entier sous forme décimale
- '**ox**' Entier sous forme hexadécimale
- '**f**' Flottant
- '**c**' Caractère
- '**s**' Chaîne 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 6. Celui-ci peut également être `void`, cela signifie que la fonction ne renvoie rien, c'est donc une procédure.

## 2.3 Structure d'une 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

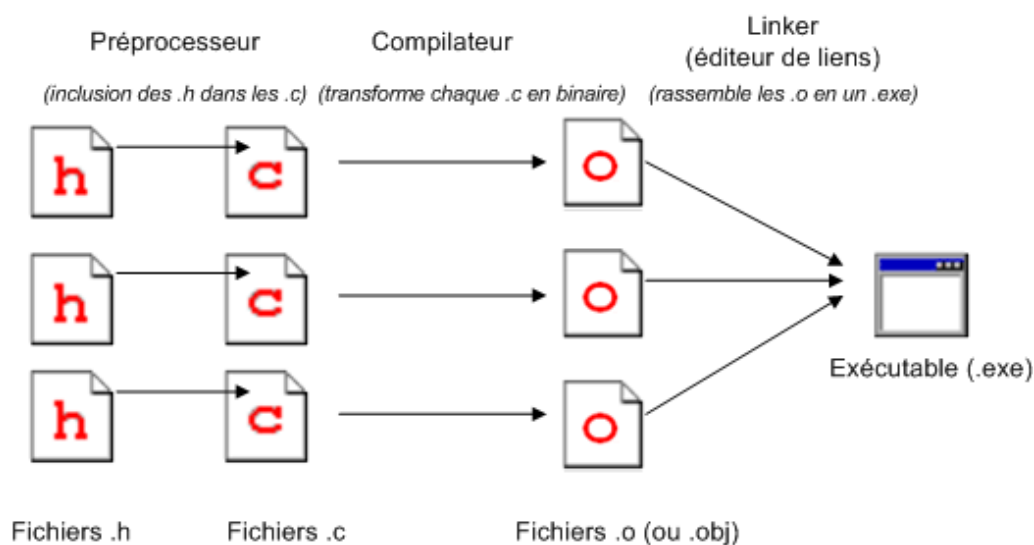


FIGURE 2.1 – La compilation

### 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'exécutable.
```

**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

## 3.1 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.

## 3.2 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.

## 3.3 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”.

## 3.4 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.

## 3.5 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);
- $f = N!$

## 3.6 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

## 3.7 Étape 4 : Programmer et unifier le programme

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

### 3.7.0.1 Vérification du programme

- Test d’exécution : Tableau de situation, vérification non exhaustive.
- Preuve formelle par calcul de “Plus faible Pré condition” (Pfp)

**Le tableau de situation**

**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.

# Spécification d'un programme

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

# Vérification formelle de programmes

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>.

**pfp** 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.

**Tableau de situation** Système de réécriture des données en entrée (mémoire à vers les données en sortie (mémoire)) en fonction des actions du programme.

## 5.1 Système de réécriture Pfp

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

Les langages impératifs :

- affectation
- Séquence
- Sélection
- répétition

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

## 5.2 Calcul de pfp d'une affectation

$$\text{pfp}("x = e", Q) = Q_n^e$$

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

---

1. wp weakest precondition



**Ex** Soit le programme suivant :

- $x > 0$
- $x = x - 1$
- $x \geq 0$

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

$$(x > 0) \rightarrow \text{pfp}("x = x - 1", x \geq 0)$$

$$(x > 0) \rightarrow (x - 1 \geq 0)$$

$$(x > 0) \rightarrow (x > 0)$$

## 5.3 Calcul du pfp d'une séquence

$$\text{pfp}("a1; a2; a3", Q) = \text{pfp}("a1; a2;", \text{pfp}("a3;", Q));$$

**Ex** Soit les programme suivant, lesquels sont justes, lesquels sont faux ?

```

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

```

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

```

- /* f = i! */
- f = f * (i + 1);
  i = i + 1;
- /* f = i! */

```

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

```

- /* (x = A) ∧ (y = B) ∧ (z = C)
- x = x + y + z;
  z = x - y - z;
  y = x - y - z;
  x = x - y - z;
- /* (x = B) ∧ (y = C) ∧ (z = A) */

```

$$\begin{aligned}
 PE &\rightarrow \text{pfp}("x = x + y + z; z = x - y - z; y = x - y - z; x = x - y - z;", (x = B) \wedge (y = C) \wedge \\
 PE &\rightarrow \text{pfp}("x = x + y + z, z = x - y - z; y = x - y - z;", \text{pfp}("x = x - y - z; (x = B) \wedge (y = \\
 PE &\rightarrow \text{pfp}("x = x + y + z, z = x - y - z; y = x - y - z;", x - y - z = B) \wedge y = C \wedge z = A \\
 PE &\rightarrow \text{pfp}("x = x + y + z, z = x - y - z", (y = B) \wedge (x - y - z = C) \wedge z = A \\
 PE &\rightarrow \text{pfp}("x = x + y + z", (y = B) \wedge (z = C) \wedge (x - y - z = A) \\
 PE &\rightarrow (y = B) \wedge (z = C) \wedge (x = A) \text{ Vrai parceque } p \rightarrow p = \text{vrai}
 \end{aligned}$$

---

a. Évaluation de la "règle" la plus profonde

---

# 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.

# Exercices

## B.1 Initiation

### B.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.

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

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

#### B.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.

## B.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 B.1 – Exercice 1 – Code du programme

## B.2 Tableau de situation

### B.2.1 Exercice 2

```

1  /* (N > 0) ∧ (N > 30) */
2  int factorielle (int n) {
3      if (0 == n) {
4          return (1); // ... 1
5      } else {
6          int f = n; // ... 2
7          while (--n > 0) {
8              f *= n; // ... 3
9          }
10         return f; // ... 4
11     }
12 }
13 /* factorielle = n! */

```

Listing B.2 – Exercice 3

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

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

## B.2.2 Exercice 3

```

1  typedef tab int[3];
2  tab T;
3  int j = 2;
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 B.3 – 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	

## B.3 Effets de bords

### B.3.1 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 B.4 – Exercice 4

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



B.3.2 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 B.5 – 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

## B.3.3 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 B.6 – Exercice 6

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

## B.4 Spécification

## B.4.1 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)`
- $(0 \leq n \leq N) \wedge (n = \nu I : 0 \leq I < NT[I] \geq 0)$

## B.4.1.1 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 pourrions 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.

## B.4.2 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  $T$

### B.4.2.1 Calcule de la première position

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

### B.4.2.2 Calcule de la dernière position

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

## B.4.3 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.

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

## B.4.4 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);`
- $(1 \leq l \leq N) \wedge (\exists I : 0 \leq I < N - l) \wedge (T[I] = T[I + l - 1])$

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

### B.4.5 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 :

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

# Liste des codes sources

2.1	Syntaxe de déclaration de variable . . . . .	6
2.2	Syntaxe d'un bloc . . . . .	6
2.3	Syntaxe des actions . . . . .	7
2.4	Syntaxe d'une structure de contrôle . . . . .	7
2.5	Syntaxe de répétition . . . . .	7
2.6	Syntaxe d'une affectation . . . . .	7
2.7	Syntaxe de l'appel de printf . . . . .	8
2.8	Syntaxe de l'appel de scanf . . . . .	8
2.9	Syntaxe de déclaration d'un tableau . . . . .	9
2.10	Syntaxe d'un sous programme . . . . .	9
2.11	Point d'entrée du programme : le main . . . . .	9
2.12	Exemple d'instructions pré-processeurs . . . . .	10
B.1	Exercice 1 – Code du programme . . . . .	20
B.2	Exercice 3 . . . . .	21
B.3	Exercice 3 . . . . .	22
B.4	Exercice 4 . . . . .	23
B.5	Exercice 5 . . . . .	24
B.6	Exercice 6 . . . . .	25