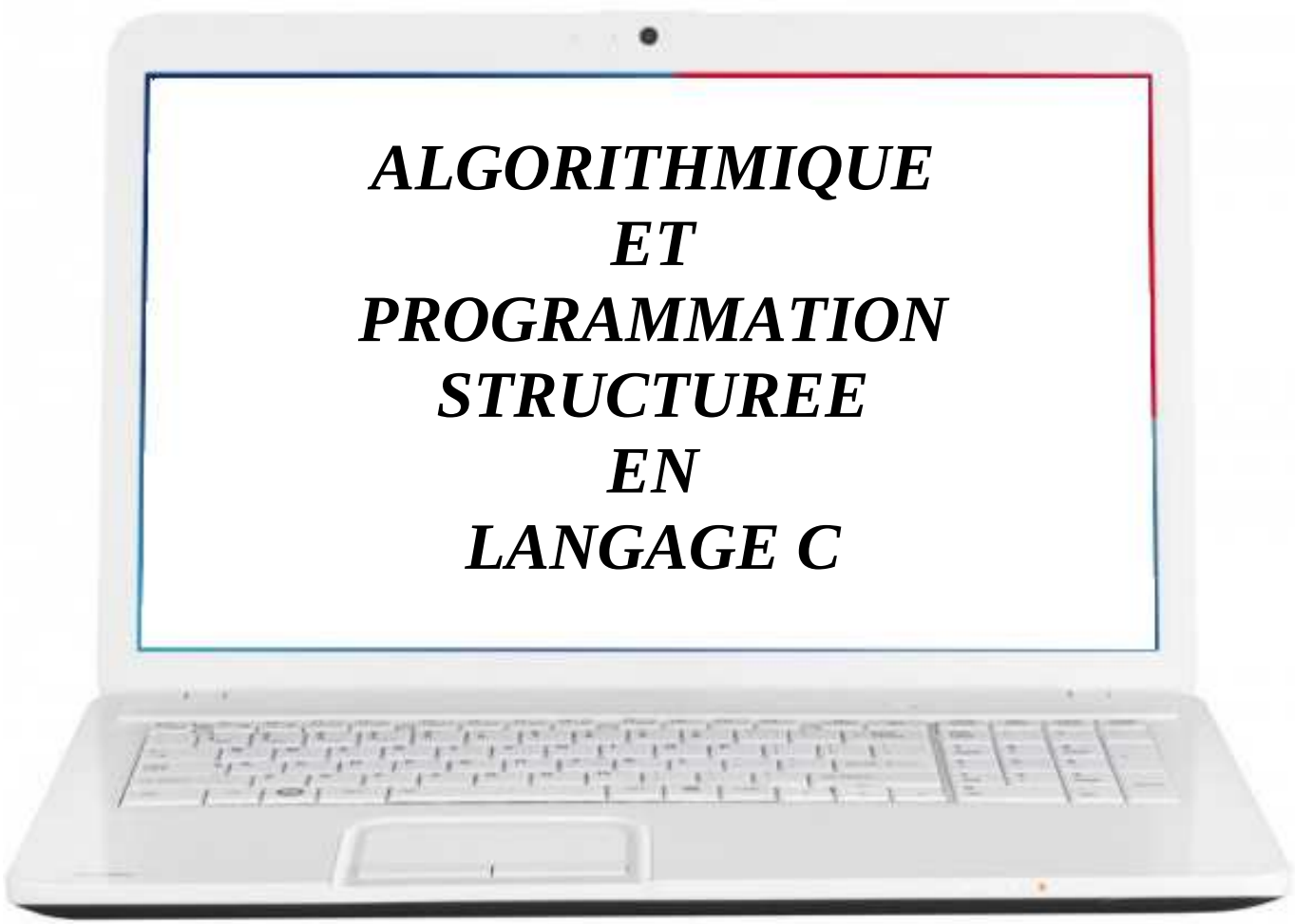
**UNIVERSITÉ DE POITIERS**

**INSTITUT UNIVERSITAIRE DE TECHNOLOGIE**
**6, allée Jean Monnet**
**BP 389**



# *ALGORITHMIQUE ET PROGRAMMATION STRUCTUREE EN LANGAGE C*

*Année 2012 - 2013*

***J.D. GABANO***
***Maître de Conférences***

# *Sommaire*

# 1. INTRODUCTION

- ✓ L'informatique est une technique de traitement auto**matique** d'**infor**mation.

- ✓ L'ordinateur est une machine conçue pour cette technique.

  Il permet d'exécuter des programmes qui effectuent l'acquisition, le stockage, le traitement et la restitution de données.

- ✓ L'exécution de ces programmes ne peut s'effectuer qu'en **langage binaire**.

# 2. CODAGE BINAIRE DE L'INFORMATION

## 2.1. Base de numération

Nombre : suite ordonnée de gauche à droite de chiffres et de signes permettant de quantifier des informations.

Dans un nombre, la valeur de chaque chiffre dépend de sa position : .....

.....

## 2.2. Base de numération – base 10

On dispose de 10 chiffres pour constituer les nombres :

0, 1, 2, 3, 4, 5, 6, 7, 8, 9

***Exemple*** : principe de la décomposition de 2354 en base 10 :

| Nombre | | | | |
|---|---|---|---|---|
| Rang | | | | |
| Poids | | | | |
| Valeur associée au chiffre | | | | |

2354 = 2000 + 300 + 50 + 4

## 2.3. Base de numération – base 2

On dispose de 2 chiffres pour constituer les nombres : 0, 1

***Exemple*** : nombre 10101 en base 2 : 10101(B) = 21(D)

| Nombre | | | | | |
|---|---|---|---|---|---|
| Rang | | | | | |
| Poids | | | | | |
| Valeur associée au chiffre | | | | | |

21 = 16 + 0 + 4 + 0 + 1

## 2.4. Bit, octet

BIT (Binary digIT) : Constituant élémentaire des informations binaires : 2 valeurs possibles : 1 ou 0

Octet (Byte) :---------- | **0** | **0** | **0** | **0** | **1** | **0** | **1** | **0** |

Avec 8 bits, il est possible de représenter ----------

Si par exemple, on souhaite coder un entier non signé sur un octet, on peut représenter les valeurs comprises entre :

0 : | **0** | **0** | **0** | **0** | **0** | **0** | **0** | **0** |

255 : | **1** | **1** | **1** | **1** | **1** | **1** | **1** | **1** |

## 2.5. Base de numération – base 16 (hexadécimale)

On dispose de 16 chiffres pour constituer les nombres :

0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F

avec : $A_{(H)} = 10_{(D)}$ , $B_{(H)} = 11_{(D)}$ , $C_{(H)} = 12_{(D)}$,

$D_{(H)} = 13_{(D)}$ , $E_{(H)} = 14_{(D)}$ , $F_{(H)} = 15_{(D)}$

***Exemple :*** nombre 2AC6 en base hexa :

| Nombre | | | | |
|---|---|---|---|---|
| Rang | | | | |
| Poids | | | | |
| Valeur associée au chiffre | | | | |

8192 + 2560 + 192 + 6

⇩

$2AC6_{(H)} = 10950_{(D)}$

# 3. STRUCTURE ELEMENTAIRE D'UN SYSTEME INFORMATIQUE

## 3.1. L'ordinateur (aspect matériel *harware*)

✓ **Microprocesseur (µP) :**

Le gestionnaire du système !

✓ **Mémoire vive (R.A.M) :**

Zone de rangement des programmes en cours d'exécution et des données manipulées par ces programmes.

✓ **Mémoire morte (R.O.M) :**

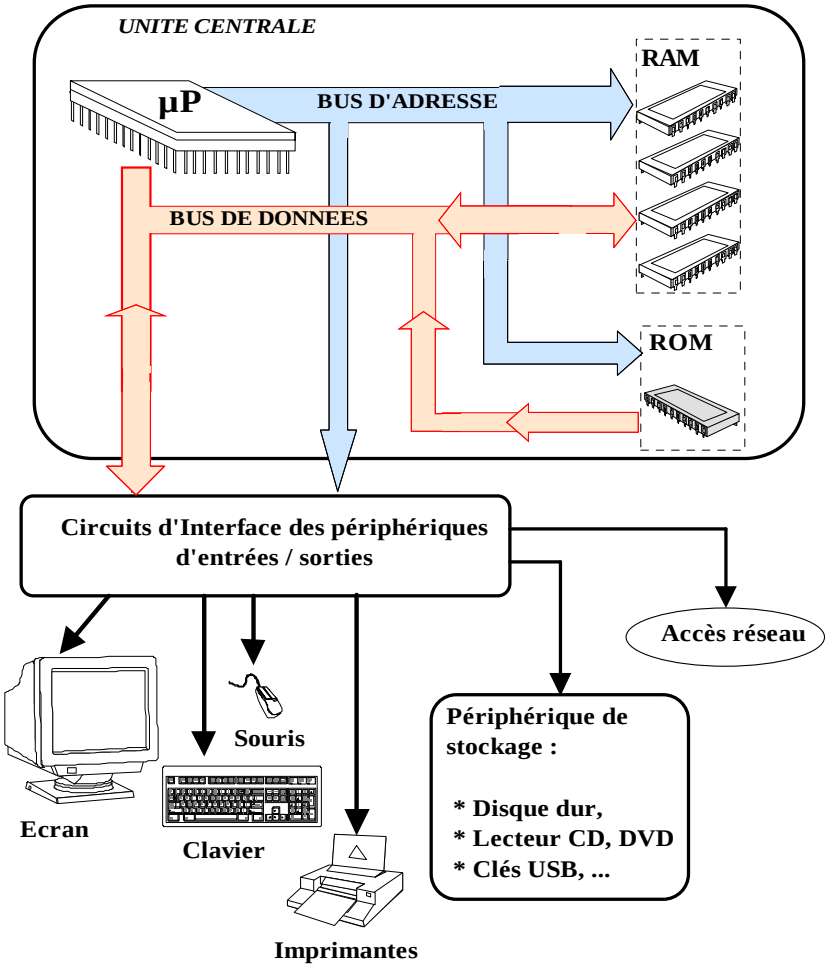
Mémoire non volatile contenant les programmes de base indispensable au démarrage de l'ordinateur (B.I.O.S : Basic Input/Output System).

✓ **Périphériques d'entrées/sorties :**

Ecran, clavier, souris, imprimante

Mémoire de masse (stockage)

Accès à un réseau




✓ **Dialogue du µP avec les composants du système :**

Langage binaire …0010010101...

- Bus de données : échange de valeurs binaires écrites par le µP ou lues par le µP.
- Bus d'adresse : permet d'indiquer la zone de mémoire (ou le périphérique de sortie) où le µP souhaite écrire une valeur présente sur le bus de donnée.

- Taille du bus interne de données : 32 bits (Pentium 1), 64 bits (Pentium 4)
  - ➔ Traitement possible de 4 octets (µP 32 bits) ou 8 octets (µP 64 bits) en une seule opération.
- Taille du bus d'adresse : 32 bits : nombre d'octets adressables : $2^{32}$ = 4.29496 109 = 4 Go.

## 3.2. Les logiciels (aspect *software*)

### 3.2.1. Système d'exploitation

Pour gérer les dialogues homme / machine de base, il faut un logiciel spécialisé, **le système d'exploitation** :

DOS, Windows 95, 98, Windows 2000, Windows NT, Windows XP,

Windows 7 (*Microsoft*), Linux, …

### 3.2.2. Programmes dédiés à différents domaines d'application :

- ✓ Gestion de données,
- ✓ Informatique scientifique (modélisation, simulateurs)
- ✓ Bureautique : Traitement de texte, tableur,...
- ✓ Systèmes industriels (gestion d'une instrumentation programmable, commandes numériques, robotique,…)
- ✓ Logiciels de C.A.O (**C**onception **A**ssistée par **O**rdinateur)

### 3.2.3. Programmes dédiés au développement d'applications

#### 3.2.3.1. Langage machine

- A chaque commande binaire du jeu d'instructions du microprocesseur (suite d'octets) correspond un code (**mnémonique**) facilement mémorisable par le programmeur.
- Ce langage est aussi appelé langage **assembleur**.
- .....

.....

#### 3.2.3.2. Langages évolués : PASCAL, C…

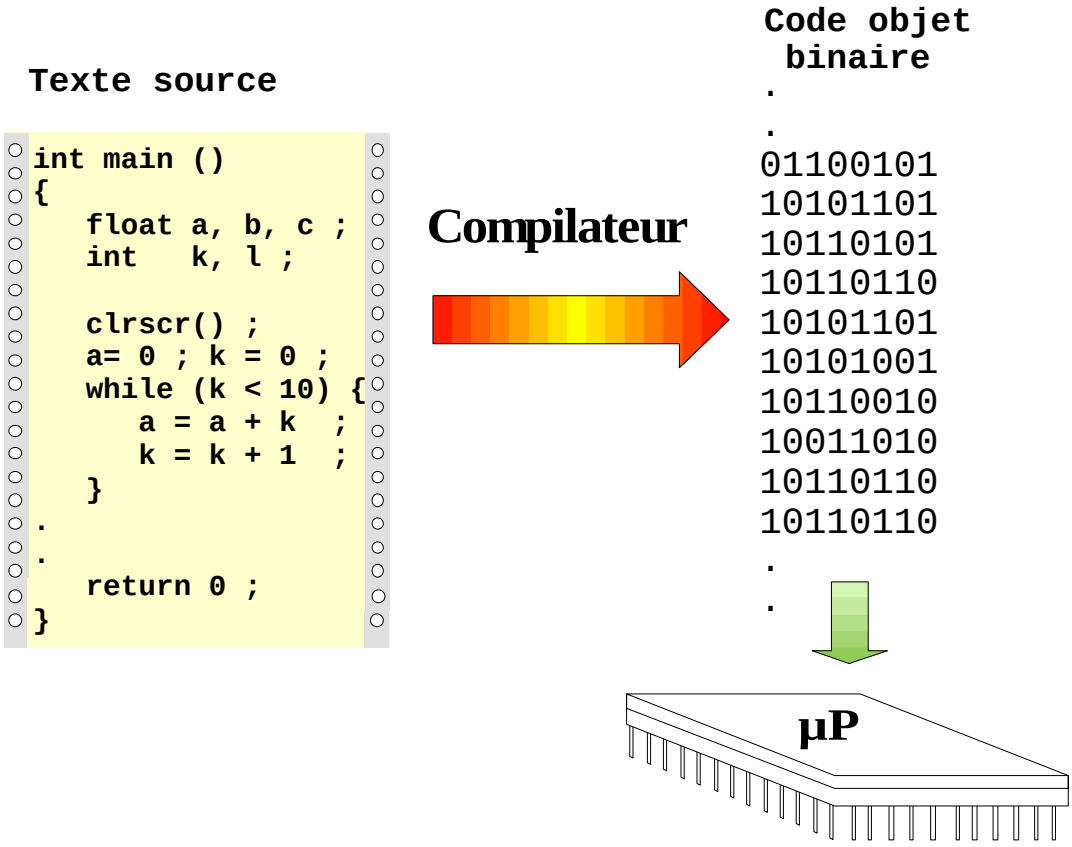
- Ces langages utilisent une sémantique qui définit le sens des mots (anglo-saxons) et une syntaxe qui définit la façon d'enchaîner les mots pour construire une phrase compréhensible par un compilateur.
- Ils ne dépendent pas du microprocesseur utilisé !
- Le texte source .....

.....

.....

- Un programme appelé **compilateur** .....

.....




### 3.2.4. Notion de fichier

Un fichier permet de sauvegarder sur un périphérique de stockage un ensemble d'informations reliées entre elles de façon logique.

La désignation d'un tel fichier a la structure :

*Nom_fichier.ext*

L'extension renseigne sur la nature des informations contenues dans le fichier (3 caractères maximum).

*Exemple :*
.c : fichier source écrit en C
.o : fichier "objet" contenant du code binaire
.exe : fichier binaire exécutable
.doc : fichier créé avec le logiciel word
.xls : fichier créé avec le tableur excel
*etc ...*

La longueur du nom peut varier suivant le système d'exploitation :

8 caractères pour le nom sous DOS et Windows 3.xx

256 caractères sous Windows 95, 98 et NT

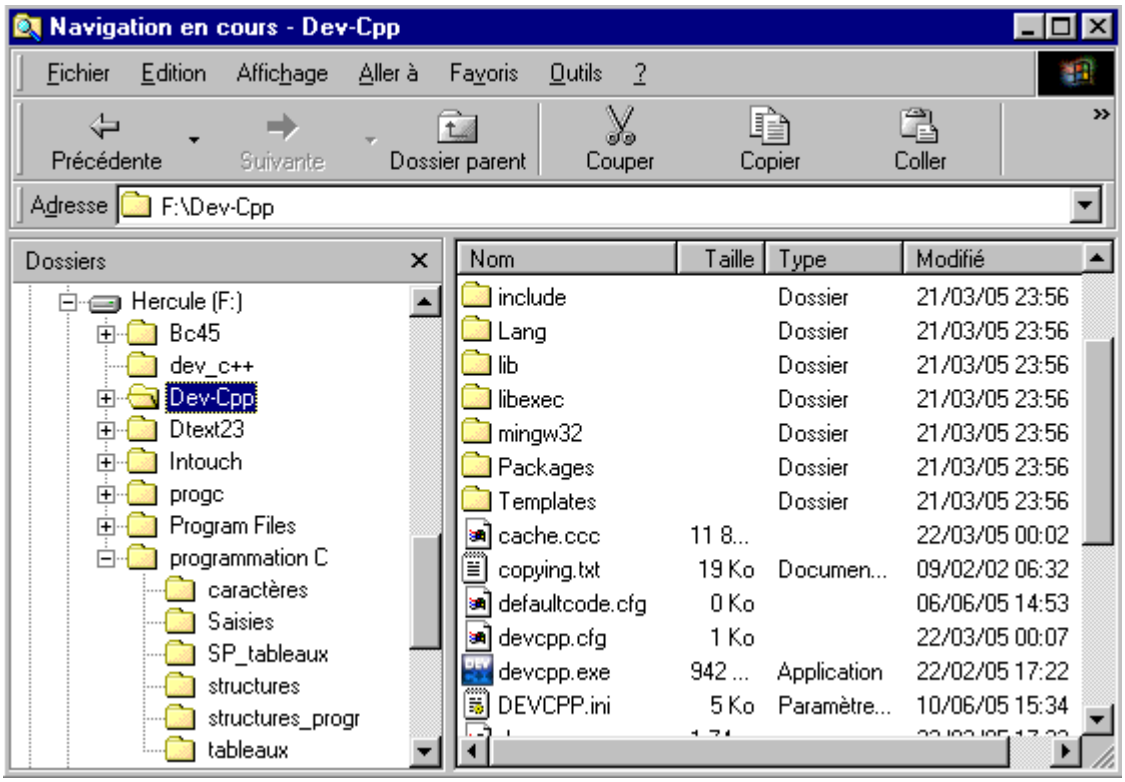
**Un fichier binaire** est une suite d'informations binaires, c'est-à-dire une suite de 0 et de 1. Celui-ci peut correspondre à des données ou un programme en code binaire que peut alors exécuter le µP après chargement en mémoire.

**Un fichier texte** est un fichier composé de caractères stockés sous la forme d'octets selon le **code A.S.CI.I.**

### 3.2.5. Notion de répertoire

Le système d'exploitation permet de ranger les fichiers sur les périphérique de stockage (disque dur, clé USB, CD, disquette …) en gérant des répertoires hiérarchisés.

*Exemple possible d'organisation de disque dur* (Ici de nom logique F:) :



# 4. ALGORITHME – GENERALITES

## 4.1. Définition

Un algorithme est un ensemble de règles opératoires dont l'enchaînement permet de résoudre un problème au moyen d'un nombre fini d'opérations (*Définition Larousse*).

Quelques points importants :

- Un algorithme décrit un traitement sur un ensemble fini de données de nature simple (nombres ou caractères) ou plus complexes (données structurées).
- Un algorithme est constitué d'un ensemble fini d'actions composées d'opérations ou actions élémentaires. Ces actions élémentaires doivent être réalisables par la machine.
- L'expression d'un algorithme nécessite un langage clair (compréhension) structuré (enchaînements d'opérations, implémentation de structures de contrôle), universel (**indépendant du langage de programmation choisi**).

⇒ ***Langage algorithmique***

## 4.2. Analyse descendante

Établir l'algorithme d'un programme, c'est :

- effectuer une analyse descendante du cahier des charges en décomposant ce dernier en tâches de plus en plus simples à réaliser,
- recenser les variables et constantes nécessaires,
- écrire en langage algorithmique (ou à l'aide d'organigrammes) et d'après l'analyse descendante précédente, les algorithmes des différentes tâches.

**En principe, un algorithme ne doit pas se référer à un langage particulier**

## 4.3. Structure du bloc exécutable d'un programme

Le bloc exécutable d'un programme **complet** comporte **trois parties** qu'il convient de bien distinguer :

| |
|---|
| **1.** ..... |
| **2.** ..... |
| **3.** ..... |

# 5. VARIABLES INFORMATIQUES DE TYPE SIMPLE

## 5.1. Les variables informatiques

Pour qu'un programme puisse traiter des informations, il est nécessaire que ces dernières soient mémorisées, ainsi que le résultat du traitement.

Les variables informatiques permettent ces mémorisations.

Une variable informatique est caractérisée par **3 éléments** :

- .....
- .....
  .....
  .....
  .....
- .....

## 5.2. Identificateur de variable

Règle de création (***empruntée au langage C***) :

- Un identificateur de variable doit obligatoirement commencer par une lettre non accentuée ou le caractère de soulignement '_'.
- Les caractères suivants doivent être des lettres non accentuées, minuscules ou majuscules, des chiffres ou le caractère de soulignement.
- Tous les autres caractères sont strictement interdits.

| |
|---|
| Le C distingue les minuscules des majuscules :<br>A est donc une variable distincte de a. |

*On empruntera la même convention en langage algorithmique.*

*Exemples :*

Cote_carre, hauteur_1 : Identificateurs valides

Cot**é**_carre, **1**_rayon : Identificateurs invalides

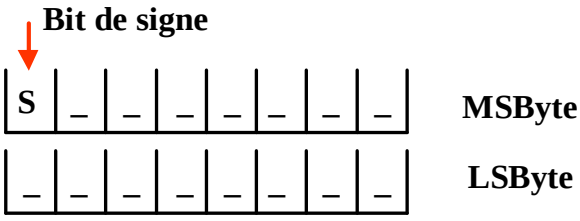
## 5.3. Les différents types simples

### 5.3.1. Type Entier

**Domaine** :

[–Nmax; Nmax [ Nmax dépend du nombre d'octets

Entier **signé** codé sur 2 octets :




**Primitives :**

- Arithmétique : +, –, *, div , mod

  div : .....

  mod : .....
- Comparaison : <, >, >=, <=, =, ≠
- Fonctions mathématiques

**Types entier du langage C**

| TYPE | DESCRIPTION | TAILLE MEMOIRE |
|---|---|---|
| **`int`** | Entier standard signé | 4 octets : $-2^{31} \leq n \leq 2^{31}-1$ |
| **`unsigned int`** | Entier standard positif | 4 octets : $0 \leq n \leq 2^{32}-1$ |
| **`short`** | Entier court signé | 2 octets : $-2^{15} \leq n \leq 2^{15}-1$ |
| **`unsigned short`** | Entier court positif | 2 octets : $0 \leq n \leq 2^{16}-1$ |

### 5.3.2. Type Réel

**Domaine** :

[-Vmax; -ε[ U [0] U [ε ; +Vmax]

- ε .....
- Vmax .....
- Réel standard codé sur **4 octets** en C.

**Primitives :**

- Arithmétique : +, –, *, /
- Comparaison : <, >, >=, <=, =, ≠
- Fonctions mathématiques

**Types réel du langage C**

| TYPE | DESCRIPTION | TAILLE MEMOIRE |
|---|---|---|
| `float` | Réel standard | 4 octets : $\varepsilon \cong 1.41\ 10^{-45}$<br>Vmax $\cong 3.39\ 10^{38}$ |
| `double` | Réel double précision | 8 octets : $\varepsilon \cong 5.01\ 10^{-324}$<br>Vmax $\cong 2.00\ 10^{308}$ |

### 5.3.3. Type Caractère

**Domaine** :

Codé sur 7 bits selon le code A.S.C.I.I. (American Standard Code for International Interchange).

**7 bits** $\Rightarrow$ .....

Ce code attribue aux **caractères imprimables** alphanumériques de l'alphabet latin des nombres décimaux entiers de **32** à **126** (numéro d'ordre dans la table des codes ASCII). Ces codes sont aussi fréquemment fournis en valeur hexadécimale (de $20_{(H)}$ à $7E_{(H)}$ ).

Le code ASCII **étendu**, utilisé par les PC est codé sur un octet (**8 bits**) : il permet donc d'encoder **256** caractères et présente 3 parties :

- 0 à 31 et 127 : $00_{(H)}$ à $1F_{(H)}$ et $7F_{(H)}$ — Codes de caractères non imprimables (directives pour certains périphériques).
  Le code **0** correspond au caractère baptisé **NUL** (Il est utilisé pour la gestion des chaînes de caractères).
- 32 à 126 : $20_{(H)}$ à $7E_{(H)}$ — Codes de caractères imprimables standards non accentués.
- 128 à 255 : $80_{(H)}$ à $FF_{(H)}$ — Codes de caractères imprimables non standards (il dépend du système d'exploitation).

**Primitives** :

- ORD : .....

  *Exemple* : ORD('A') = ..... ORD('B') = .....

  ORD('a') = ..... ORD('b') = .....
- CHR : .....

  *Exemple* : CHR(65) = ..... CHR(66) = .....

- Comparaison : <, >, >=, <=, =, ≠

*Remarque* : 'A' < 'B', mais 'a' > 'B'.

Les relations d'ordre sont définies à partir du code ASCII (entier)

Une valeur de type caractère est toujours délimitée par deux **quotes** ' '.

**Table des codes ASCII Standard (en valeur décimale)**

| N° | CAR | N° | CAR | N° | CAR | N° | CAR | N° | CAR |
|---|---|---|---|---|---|---|---|---|---|
| 32 | ' ' | 52 | '4' | 72 | 'H' | 92 | '\' | 112 | 'p' |
| 33 | '!' | 53 | '5' | 73 | 'I' | 93 | ']' | 113 | 'q' |
| 34 | '"' | 54 | '6' | 74 | 'J' | 94 | '^' | 114 | 'r' |
| 35 | '#' | 55 | '7' | 75 | 'K' | 95 | '_' | 115 | 's' |
| 36 | '$' | 56 | '8' | 76 | 'L' | 96 | '`' | 116 | 't' |
| 37 | '%' | 57 | '9' | 77 | 'M' | 97 | 'a' | 117 | 'u' |
| 38 | '&' | 58 | ':' | 78 | 'N' | 98 | 'b' | 118 | 'v' |
| 39 | ''' | 59 | ';' | 79 | 'O' | 99 | 'c' | 119 | 'w' |
| 40 | ' (' | 60 | '<' | 80 | 'P' | 100 | 'd' | 120 | 'x' |
| 41 | ')' | 61 | '=' | 81 | 'Q' | 101 | 'e' | 121 | 'y' |
| 42 | '*' | 62 | '>' | 82 | 'R' | 102 | 'f' | 122 | 'z' |
| 43 | '+' | 63 | '?' | 83 | 'S' | 103 | 'g' | 123 | '{' |
| 44 | ',' | 64 | '@' | 84 | 'T' | 104 | 'h' | 124 | '\|' |
| 45 | '-' | 65 | 'A' | 85 | 'U' | 105 | 'i' | 125 | '}' |
| 46 | '. ' | 66 | 'B' | 86 | 'V' | 106 | 'j' | 126 | '~' |
| 47 | '/' | 67 | 'C' | 87 | 'W' | 107 | 'k' | | |
| 48 | '0' | 68 | 'D' | 88 | 'X' | 108 | 'l' | | |
| 49 | '1' | 69 | 'E' | 89 | 'Y' | 109 | 'm' | | |
| 50 | '2' | 70 | 'F' | 90 | 'Z' | 110 | 'n' | | |
| 51 | '3' | 71 | 'G' | 91 | '[' | 111 | 'o' | | |

En langage C, les fonctions **ORD** et **CHR** n'existent pas : le type **`char`** permet de stocker des caractères mais peut aussi être considéré comme **un cas particulier d'entier codé sur 1 octet**.

On peut donc envisager le contenu d'une variable de type **`char`** comme une **valeur numérique entière** ou bien la **valeur caractère** correspondante parmi les valeurs répertoriées dans la table des codes ASCII.

**Types caractère du langage C**

| TYPE | DESCRIPTION | TAILLE MEMOIRE |
|---|---|---|
| **`char`** | Caractère signé | 1 octet : $-128 \leq n \leq 127$ |
| **`unsigned char`** | Caractère non signé | 1 octet : $0 \leq n \leq 255$ |

### 5.3.4. Type Booléen

**Domaine** : **VRAI**, **FAUX**

**Primitives :**

**Opérateurs booléens :**

| | **Traduction en C** |
|---|---|
| • ET | **&&** |
| • OU | **\|\|** |
| • NON | **!** |

Soit P1 et P2 deux variables booléennes :

| P1 | P2 | P1 ET P2 | P1 OU P2 |
|---|---|---|---|
| FAUX | FAUX | | |
| FAUX | VRAI | | |
| VRAI | FAUX | | |
| VRAI | VRAI | | |

| P1 | NON P1 |
|---|---|
| FAUX | |
| VRAI | |

**Gestion du type booléen en C**

Un test logique, appelé aussi prédicat, fournit un résultat de type booléen **VRAI** ou **FAUX**.

**Il n'existe pas de type prédéfini booléen en C** :

La valeur booléenne **FAUX** de tout prédicat .....

.....

La valeur booléenne **VRAI** de tout prédicat .....

.....

***Remarque :*** *toute valeur scalaire différente de 0 peut être interprétée en valeur booléenne* **VRAI**.

**Traduction en C de prédicats**

| *(Langage algorithmique)* : A, B variables de type réel | | | **Traduction C** |
|---|---|---|---|
| A = B | test d'égalité | : | `A == B` |
| A ≠ B | test de différence | : | `A != B` |
| A < B | test inférieur strictement | : | `A < B` |
| A <= B | test inférieur ou égal | : | `A <= B` |
| A > B | test supérieur strictement | : | `A > B` |
| A >= B | test supérieur ou égal | : | `A >= B` |

## 5.4. Déclaration de variables de type simple

.....

.....

***Langage algorithmique***

**VAR**

A, B : Réel

i, j : Entier

Lettre : Caractère

***Langage C***

```
float A, B ;
int i, j ;
char Lettre ;
```

**En langage C, toute instruction simple (et en particulier de déclaration) se termine par le caractère ;**

## 5.5. Déclaration de constantes

- On peut définir des constantes à l'aide d'identificateurs qui suivent les mêmes règles de création que celles utilisées pour les identificateurs de variable.
- On les écrit en général avec des lettres **MAJUSCULES**.
- .....

.....

*Exemple de déclaration en langage algorithmique :*

**CONST**

N_MAX = 100

PI = 3.14159

## Déclaration de constantes en langage C

- ***1ère méthode :*** déclaration d'une **variable** dont la valeur sera **constante** pour tout le programme (à l'aide du mot-clé **`const`**) :

  ***Exemple :***
  ```
  const int N_MAX = 100 ;
  const float PI = 3.14159 ;
  ```

  Dans ce cas, le compilateur réserve de la place en mémoire (ici 4 octets pour la variable **`N_MAX`** et 4 octets pour la variable **`PI`**. Mais les valeurs de **`N_MAX`** et **`PI`** ne peuvent être changées.

- ***2ème méthode :*** définition d'un **nom de constante** à l'aide de la directive de compilation **`#define`** :

  ***Exemple :***
  ```
  #define N_MAX 100
  #define PI 3.14159
  ```

  Dans ce cas, le compilateur ne réserve pas de place en mémoire. Il interprète le nom de constante comme la valeur définie par la directive de compilation **`#define`**. **On privilégiera donc ce type de déclaration de constante.**

  *Remarque* : les directives de compilation ne se terminent pas par un point-virgule.

# 6. ACTIONS ELEMENTAIRES

Tous les langages informatiques proposent des mots ou des symboles permettant aux programmes d'effectuer **trois** actions de bases.

## 6.1. Affectation

.....

.....

L'affectation est symbolisée par ←

*Exemple :*

**VAR**

A, B, C : Entier

**Début**

A ← 10 .....

B ← 2 + 3 .....

C ← (A + B)*2 .....

**Fin**

## L'affectation n'est pas une égalité

| **Résolution d'une équation mathématique :** |
|---|
| A = 2*A – 5 |
| Une solution unique : A = 5 |

| **Exécution d'un algorithme :** | |
|---|---|
| A ← 10 | A contient alors la valeur 10 |
| A ← 2*A – 5 | A contient alors la valeur 15 |

**Traduction de l'affectation en C :**

```
A = 10 ;          (A ← 10)
A = 2*A - 5 ;     (A ← 2*A – 5)
```

- **Ecrire les séquences permettant de permuter le contenu de deux variables A, B de type réel :**
- Combien de variables sont-elles nécessaires ?

.....

.....

.....

.....

## 6.2. Les entrées : instruction Lire

.....

.....

*Syntaxe en langage algorithmique (exemple) :* **Lire** (A)

Cette lecture peut s'opérer à partir :

- .....
- .....
- .....

## 6.3. Les sorties : instruction Ecrire

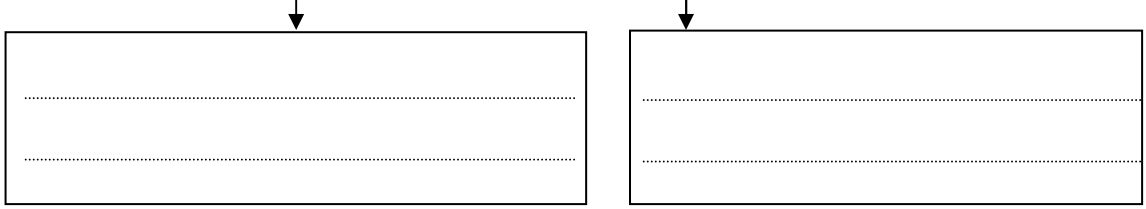
.....

.....

- .....
- .....
- .....

*Syntaxe en langage algorithmique (exemple)*

**Ecrire** ("Le contenu de la variable A est : ", A )



## 6.4. Evolution du contenu des variables dans un algorithme

Lors de la déclaration des variables, le compilateur attribue des adresses aux variables :

Il est donc indispensable de préciser le **type** des variables à mémoriser afin que le compilateur puisse réserver **la place nécessaire** en mémoire.

*Exemple d'algorithme*

**VAR**

R : Réel

N : Entier

**Début**

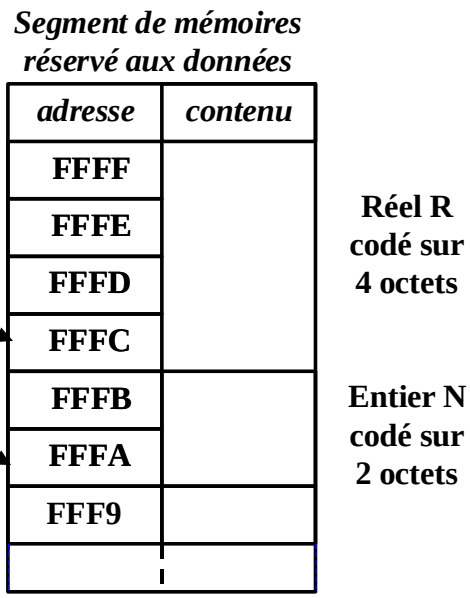
Ecrire ("Entrer un réel :")

Lire (R)

N ← 5

R ← R*N

**Fin**




Au cours de cet algorithme, le contenu de R change.

# 7. STRUCTURES DE CONTROLE

## 7.1. Introduction

Tout **algorithme** peut s'écrire à l'aide de **trois structures de contrôle** (principe de la programmation structurée) :

**1. La séquence :**

.....

.....

**2. L'alternative :**

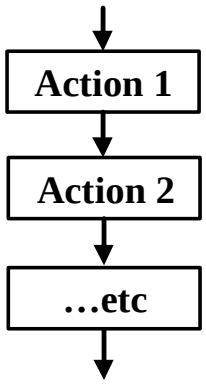
.....

.....

**3. L'itération :**

.....

## 7.2. Séquence

| Organigramme | Langage algorithmique |
|---|---|
|  | Action 1 /* Commentaires */<br>Action 2<br>… etc |



On convient de mettre entre les symboles /* et */ les commentaires qui ne correspondent pas aux actions algorithmiques que peut exécuter la machine. (Convention empruntée à la syntaxe du langage C)

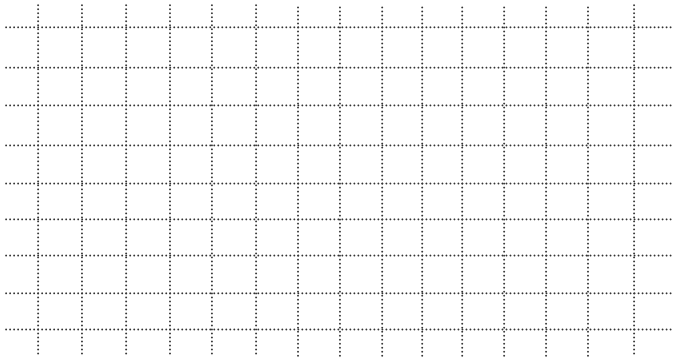
**Instructions simples et commentaires en C :**

- Toute instruction simple se termine par le caractère **';'**
- Il existe deux manières d'écrire des commentaires (qui ne doivent donc pas être compilés) dans un texte source :
  - Les commentaires écrits sur plus d'une ligne sont encadrés par les symboles **/*** et ***/ ,**
  - Les commentaires écrits sur une seule ligne sont précédés des symboles **//.**

## 7.3. Alternatives

### 7.3.1. Alternative

**Organigramme**



**Langage algorithmique**

**Si (**Prédicat VRAI**)**

**Alors** Séquence 1

**Sinon** Séquence 2

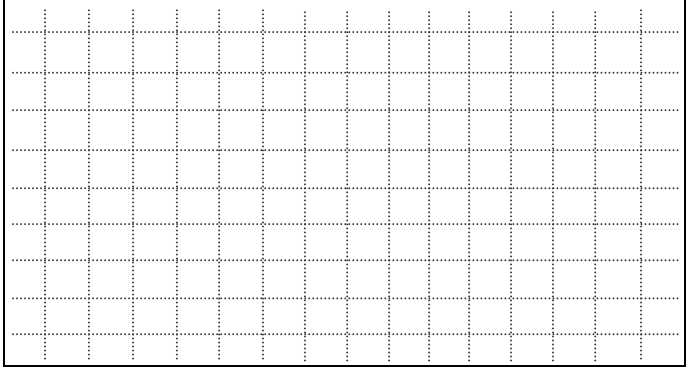
**Fin si**

**Langage C :**

```
if ( Prédicat VRAI )
{
    ... ;  // Séquence 1
    ... ;
}
else
{
    ... ;   // Séquence 2
    ... ;
}
```

### 7.3.2. Alternative simple

**Organigramme**



**Langage algorithmique**

**Si (**Prédicat VRAI**)**

**Alors** Séquence

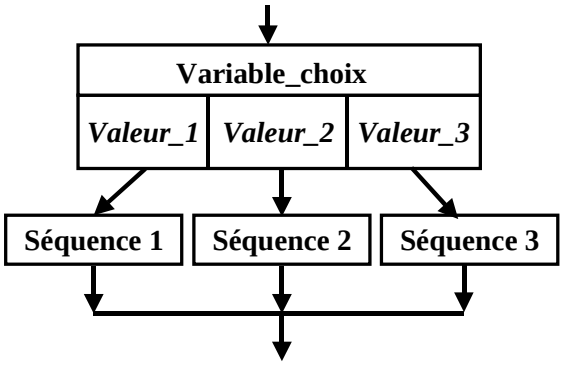
**Fin si**

**Langage C :**

```
if ( Prédicat VRAI )
{
    ... ;  // Séquence 1
    ... ;
}
```

### 7.3.3. Alternative multiple

**Organigramme**




**Langage algorithmique**

**Cas de** (Variable_choix)

***Valeur_1* :** Séquence 1

***Valeur_2* :** Séquence 2

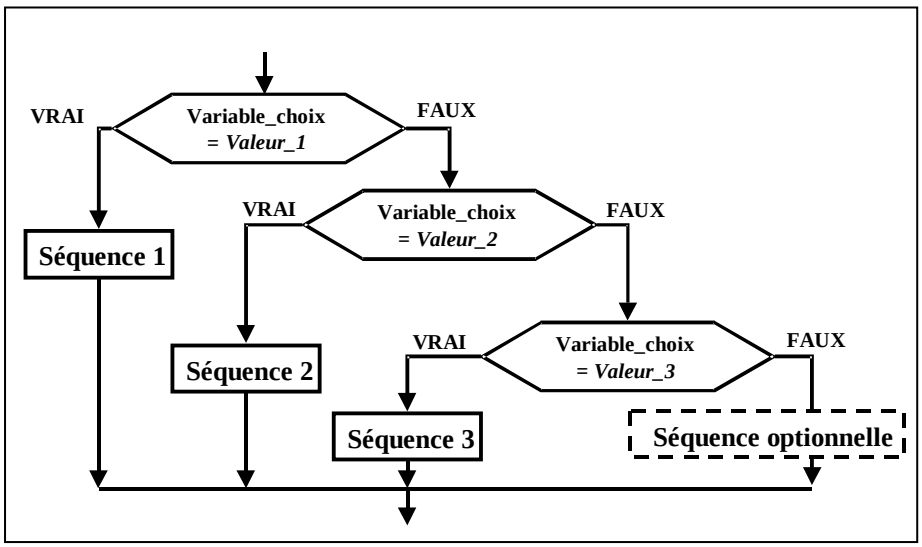
***Valeur_3* :** Séquence 3

**Fin cas**

*L'alternative multiple est équivalente à une séquence composée d'alternatives en cascade.*

***Remarque :***

si **Variable_choix** ne contient aucune des valeurs envisagées dans les divers branchements, la séquence suivante est **automatiquement exécutée**.



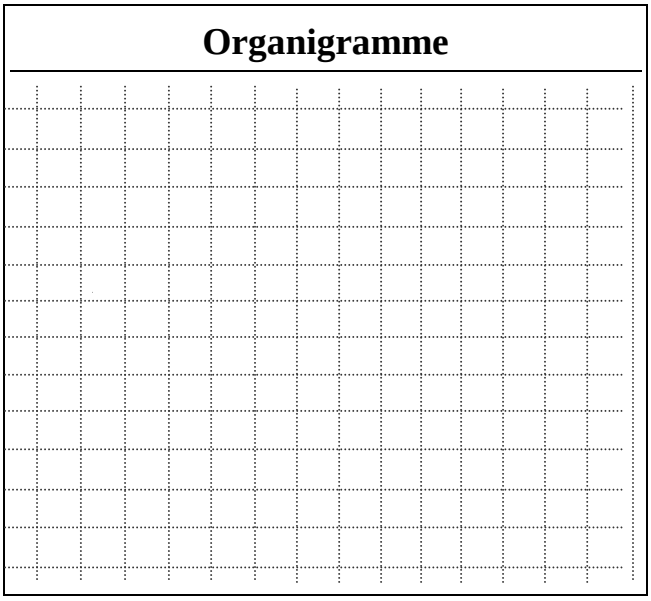
**Langage C :**

```
switch (Variable_choix)
{
    case Valeur_1 :  ...   ;  // Séquence 1
                     ...   ;
                     break ;
    case Valeur_2 :  ...   ;  // Séquence 2
                     ...   ;
                     break ;
    case Valeur_3 :  ...   ; // Séquence 3
                     ...   ;
                     break ;
    default       :  ...   ; // Séquence
                     ...   ; // optionnelle
}
```

## 7.4. Itérations

### 7.4.1. Itération universelle Tant que

**Organigramme**



**Langage algorithmique**

**Tant que (**Prédicat VRAI**) Faire**

Séquence

**Fin tant que**

**Langage C :**

```
while ( Prédicat VRAI )
{
   ... ;
   ... ;  // Séquence
   ... ;
}
```

Il se peut que la séquence ne soit **jamais exécutée** .....

.....

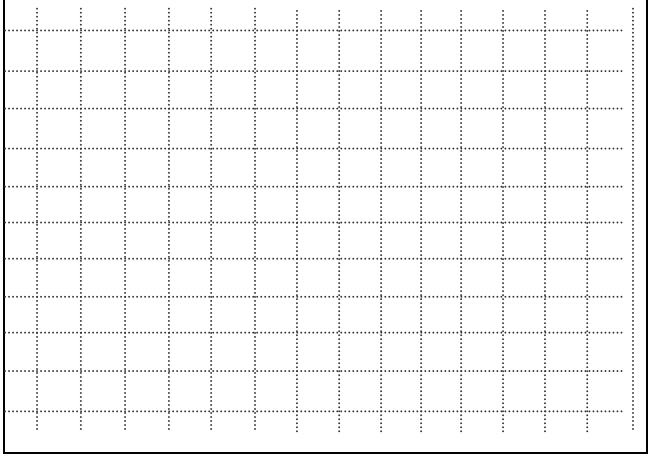
Il faut que.....

.....

.....

### 7.4.2. Itération Faire … Tant que

**Organigramme**



**Langage algorithmique**

**Faire**

Séquence

**Tant que (**Prédicat VRAI**)**

**Langage C :**

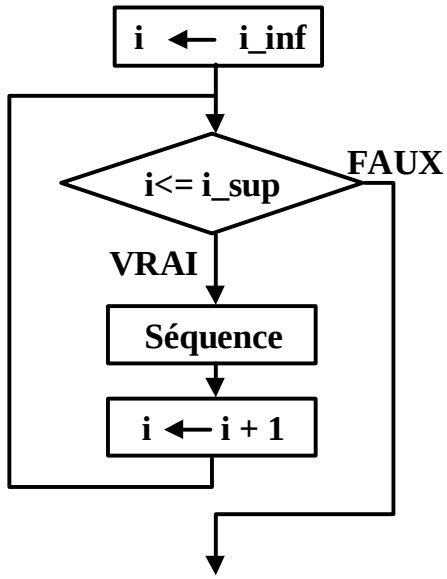
```
do
{
   ... ;
   ... ;  // Séquence
   ... ;
} while ( Prédicat VRAI );
```

La séquence est obligatoirement exécutée une fois, .....

Il faut que la séquence puisse rendre faux le prédicat sinon, la boucle sera répétée indéfiniment.

### 7.4.3. Itération Pour

**Organigramme**




**Langage algorithmique**

**Pour** i **de** i_inf **à** i_sup par pas de **1 Faire**

Séquence

**Fin pour**

- Structure intéressante lorsqu'existe une notion d'incrémentation d'un compteur i parcourant un intervalle de valeurs entières [i_inf;i_sup] connu à l'avance. **La gestion du compteur est implicite**.

- Cette structure fonctionne comme une itération universelle **Tant que**.

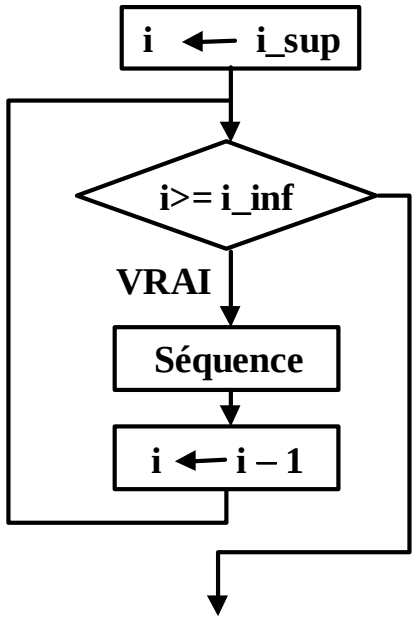
Il se peut que la séquence ne soit jamais exécutée si le i_inf > i_sup !

**Langage C :**

```
for (i=i_inf ; i <= i_sup ; i = i+1)
{
   ... ;
   ... ;      // Séquence
   ... ;
}
```

**Itération Pour (boucle descendante)**

**Organigramme**




**Langage algorithmique**

**Pour** i **de** i_inf **à** i_sup par pas de -1 **Faire**

Séquence

**Fin pour**

- Le compteur i est ici **décrémenté** de manière implicite d'une unité depuis la valeur i_sup jusqu'à la valeur i_inf.

**Langage C :**

```
for (i=i_sup ; i <= i_inf ; i = i-1)
{
   ... ;
   ... ;      // Séquence
   ... ;
}
```

**Remarque : Le langage C autorise des écritures simplifiées pour l'incrémentation et la décrémentation de variables :**

`i = i+1` est équivalent à `i++`

`i = i-1` est équivalent à `i--`

## 7.5. Exercice

Ecrire en langage algorithmique un programme **calculatrice** permettant d'effectuer :

- la saisie de deux opérandes **A**, **B** permettant de stocker 2 valeurs réelles,
- La saisie d'un caractère **Operateur** ne pouvant prendre exclusivement que les valeurs '**+**' , '**-**', '*****', '/'
- Une opération (somme, différence, multiplication et division) entre les 2 opérandes correspondant à l'opérateur choisi. Le résultat du traitement correspondant sera stocké dans une variable **C** dont le contenu sera affiché.

On tiendra compte du fait que l'opération '/' est impossible si **B = 0**.

On complètera les organigrammes permettant l'analyse du problème.

**VAR**

.....

.....

.....

.....

**Début**

/* Opérations de saisie * /

.....

.....

.....

.....

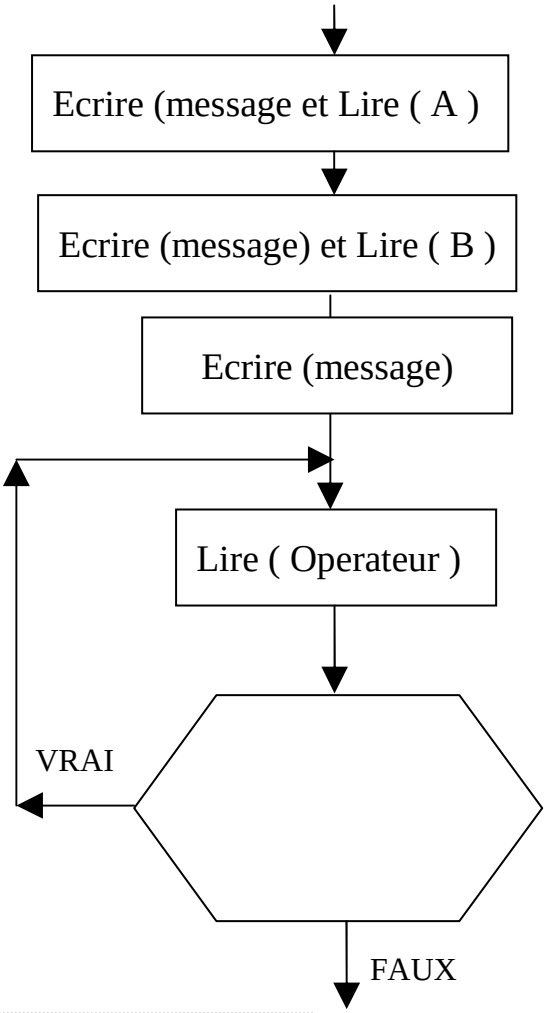
.....

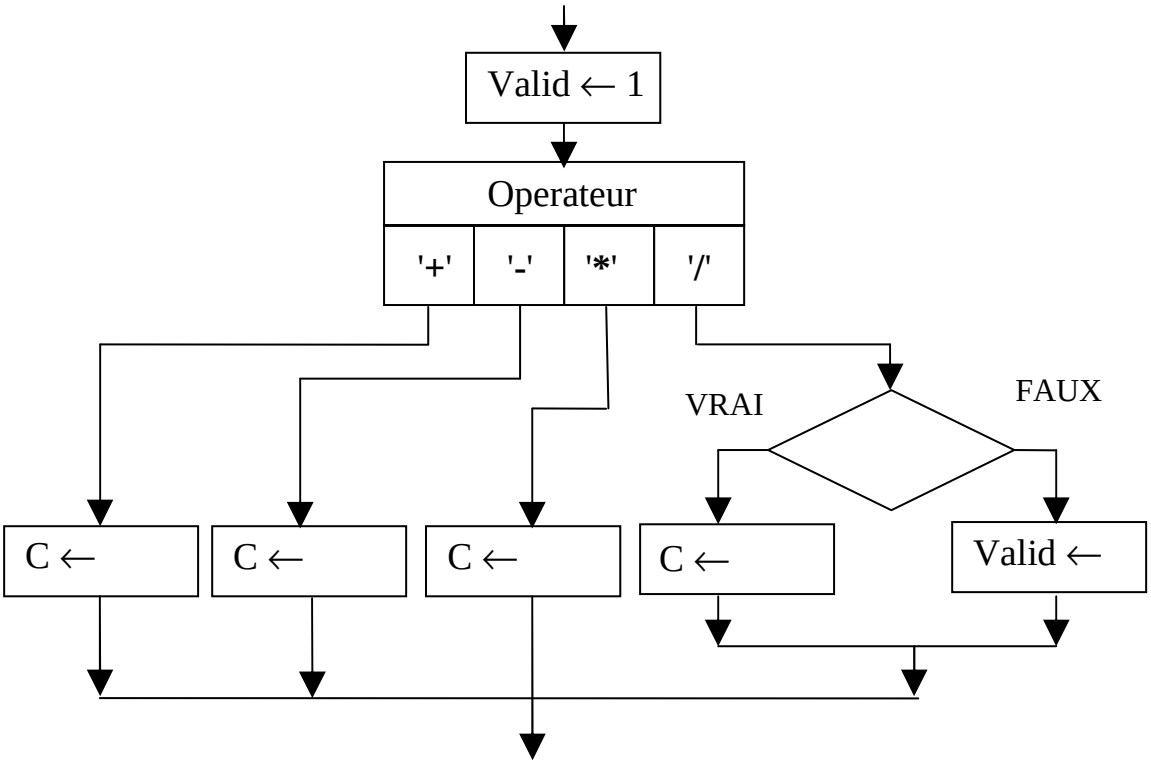
.....

.....

.....

.....






/* Traitement */

/* Edition des résultats */

# 8. INFORMATION STRUCTUREE

## 8.1. Introduction

- Les nombres (entiers ou réels), les caractères, sont des informations scalaires.
- Les variables de type simple recevant un scalaire ne permettent pas le traitement d'une grande quantité d'informations, car il faut autant de variables qu'il y a de valeurs à traiter, ce qui devient très rapidement impossible à programmer.
- Il faut donc définir des variables de type structuré à partir de variables de type simple. L'utilisateur pourra construire ses propres types par des déclarations de type.
- On se limitera dans ce cours aux variables de type Tableau.

## 8.2. Tableau de nombres (à une dimension)

### 8.2.1. Principe

*Exemple d'introduction :*

Soit un programme devant déterminer et stocker dans V_min la plus petite valeur contenue au sein d'une liste de **10** variables réelles.

Une première solution consiste à déclarer, en plus de la variable V_min, **10** variables réelles :

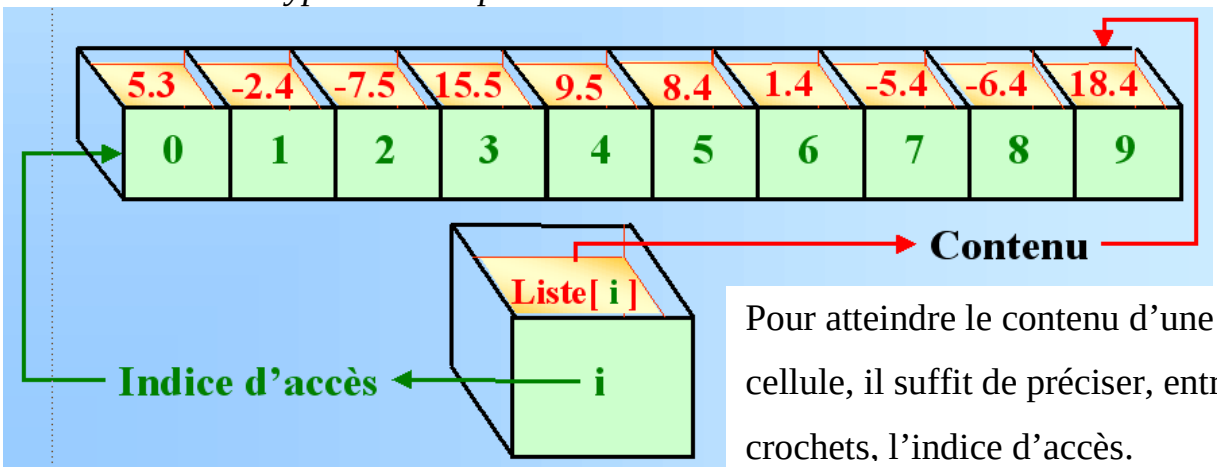
```
VAR
   V_min : Réel
   V1, V2, V3, V4, V5, V6, V7, V8, V9, V10 : Réel
```

*Remarque* : on conçoit bien que cette solution devient irréaliste si l'on souhaite effectuer la même tâches avec **100** variables réelles !

**Principe d'une variable Tableau à une dimension**

Une variable de type **tableau** à une dimension permet de ranger en mémoire **sous un même nom**, mais avec un indice d'accès de type entier, plusieurs valeur **du même type**.

*Variable* **Liste** *de type Tableau pouvant contenir 10 réels.*




Pour atteindre le contenu d'une cellule, il suffit de préciser, entre crochets, l'indice d'accès.

Dans l'exemple ci-dessus, la **1ère** valeur de la variable est

**Liste** [ **i** ] .....

.....

**Liste [ 0 ]** = ……. **Liste [ 1 ]** = …… **Liste [ 9 ]** = ……..

La déclaration directe d'une variable Tableau de nombres à une dimension s'écrit tel qu'indiqué dans l'exemple ci-dessous :

**CONST**

N_MAX = 10

**VAR**

Liste : .....

i : .....

**Liste** est un tableau **statique** : .....

.....

**Utilisation de la déclaration de type :**

**CONST**
N_MAX = 10
**TYPE** .....
**VAR**
Liste : t_reels /* Tableau d'au plus 10 réels */

t_reels *est maintenant un nouveau type d'information qui va s'utiliser comme un type simple pour déclarer des variables de type* Tableau d'au plus N_MAX réels.

**Déclaration de tableau (à une dimension) de nombres en C :**

```
#define N_MAX 10  // définition d'une constante
```

*Déclaration directe de la variable* **Liste** *:*

```
float Liste [N_MAX]  ; /* Liste : Tableau d'au plus
                          N_MAX réels */
```

*Utilisation de la déclaration de type synonyme :* **typedef**

```
typedef float t_reels [N_MAX] ;
              /* On définit le nouveau type t_reels
                 synonyme de tableau de N_MAX réels */
```

*Utilisation du type* **t_reels** *pour déclarer la variable* **Liste**

```
 t_reels Liste ; /*Tableau d'au plus 10 réels */
```

### 8.2.2. Gestion du nombres de valeurs significatives

- Un tableau de nombres n'est pas obligatoirement rempli totalement de valeurs significatives.....

.....

.....

- Pour un tableau de nombres, le programmeur est donc obligé d'utiliser une **variable**, de type **entier**, qui contiendra la **quantité de valeurs significatives** que l'on désire stocker dans ce tableau.

.....

.....

.....

.....

*Exemple : soit la déclaration :*

```
CONST
  N_MAX = 10
TYPE .....
VAR
   Liste : t_reels  /* Tableau d'au plus 10 réels */
    i      : Entier
           ;
```

Le tableau **Liste** ne sera pas obligatoirement rempli totalement de valeurs significatives (les 10 places ne seront pas obligatoirement utilisées).

.....

.....

| Indice i | 0 | 1 | 2 | 3 | | | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|
| Liste [ i ] | 5.3 | -15.7 | -0.02 | 1502 | | | 17.1 | 8.8 | 0.9 |

.....

.....

### 8.2.3. Gestion d'un tableau de nombres

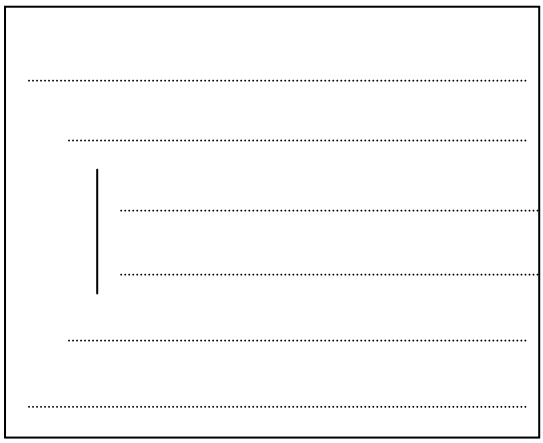
Les variables tableau .....

.....

**CONST**
N_MAX = 10
**TYPE**
t_reels = Tableau[N_MAX] de Réels

**VAR**
Liste : t_reels /* Tableau d'au plus 10 réels */
i : Entier /* Indice d'accès aux valeurs du tableau */
N : Entier /* Nombres de valeurs significatives (N <=N_MAX) */

L'algorithme général de gestion d'une variable tableau à une dimension utilise une seule itération :



Si le traitement est la saisie des valeurs du tableau, cet algorithme doit être précédé de la demande et de la lecture sécurisée de la valeur de N tel que :

N > 0 ET N <= N_MAX.

## 8.3. Tableau de caractères (chaîne de caractères)

Les chaînes de caractères .....

.....

*Exemple :*
**CONST**
NB_CAR_MAX = 9
**TYPE** t_chaine = Tableau[NB_CAR_MAX] de Caractères
**VAR**
Phrase : t_chaine /* Tableau d'au plus 9 caractères pouvant stocker une chaîne d'au plus 8 caractères*/

Afin de signaler le dernier caractère significatif d'une chaîne contenue dans un tableau de caractères, le langage C place, après celui-ci, le caractère NUL (n° 0 du code A.S.C.I.I.) lors de la saisie au clavier.

Exemple de contenu possible de la variable Phrase :

| 'H' | 'E' | 'L' | 'L' | 'O' | ' ! ' | NUL | ? | ? |
|---|---|---|---|---|---|---|---|---|
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |

.....

.....

Le caractère NUL de fin de chaîne (caractère sentinelle) permet d'adapter l'algorithme de traitement de chaînes de caractères pour ne gérer que les caractères significatifs du tableau :

**Début**

.....

.....

.....

.....

.....

**Fin**

**Déclaration d'une chaîne de caractères en C :**

```
#define NB_CAR_MAX 9 // définition de constantes
```

*Déclaration directe de la variable* **phrase** *:*

```
char Phrase [NB_CAR_MAX] ; /* Phrase : Tableau
                              d'au plus NB_CAR_MAX caractères */
```

*Utilisation de la déclaration de type synonyme :* **typedef**

```
typedef char t_chaine [NB_CAR_MAX] ;
       /* On définit le nouveau type t_chaine synonyme
          de tableau de NB_CAR_MAX caractères */
```

*Utilisation du type* **t_chaine** *pour déclarer la variable* **phrase**

```
t_chaine Phrase ; /* Chaîne d'au plus NB_CAR_MAX
                     caractères */
```

## 8.4. Remarque sur la manipulation de tableaux en C

Le langage C ne permet pas de manipuler directement l'ensemble des valeurs stockées dans des variables tableau.

.....

.....

Si **Liste1** et **Liste2** sont 2 variables **tableaux** de nombres (**entiers** ou **réels**), l'instruction algorithmique **Liste1** ← **Liste2** est donc **interdite**.

**L'affectation** ne peut se faire **qu'élément par élément**. Donc, la séquence algorithmique permettant de recopier les **N** valeurs significatives du tableau **Liste1** dans le tableau **Liste2** s'écrit :

.....

.....

.....

## 8.5. Tableau à deux dimensions

### 8.5.1. Principe

Le principe est le suivant : chaque cellule d'une variable tableau est repérée par deux indices entiers (*Il s'agit d'un tableau à double entrée*).

*Exemple* : tableau **M** de réels de 4 lignes et 4 colonnes.

| i \ j | 0 | 1 | 2 | 3 |
|---|---|---|---|---|
| 0 | 1.0 | –2.0 | –4.5 | –5.6 |
| 1 | 2.0 | 1.1 | –3.8 | –5.2 |
| 2 | 4.5 | 3.8 | 1.2 | –15.2 |
| 3 | 5.6 | 5.2 | 15.2 | 1.3 |

Pour atteindre le **contenu** d'une cellule, il suffit de préciser, entre **crochets**, le **n° de ligne** et le **n° de colonne**. Ainsi, pour l'exemple ci dessus :

M [0] [0] = 1.0, M [0] [1] = –2.0, M [2] [2] = 1.2 M [2] [3] = –15.2

### 8.5.2. Déclaration

**Utilisation de la déclaration de type :**

**CONST**

N_LIGN_MAX = 4

N_COLON_MAX = 4

**TYPE** t_matrice = Tableau [N_LIGN_MAX][N_COLON_MAX] de Réels

**VAR**

M : t_matrice /* Tableau d'au plus 4 x 5 entiers */

i , j : Entier /* Indice d'accès aux valeurs du tableau */

Nlign, Ncolon : Entier /* Nombre de lignes et de colonnes effectives */

M [i] [j] permet d'accéder à l'ensembles des 16 valeurs stockées dans la variable M : 0 <= i < 4, 0 <= j < 4.

Remarque : De même que pour les tableaux à une dimension, il faudra assurer, pour éviter tout débordement de zone allouée en mémoire, que :

1 <= Nlign < = N_LIGN_MAX et 1 <= Ncolon < = N_COLON_MAX

**Déclaration de tableau (à deux dimensions) de nombres en C :**

```
#define NB_LIGN 4 // définition de constantes
#define NB_COL  5
```

*Déclaration directe de la variable* **M** *:*

```
int M [NB_LIGN][NB_COL]  ; /* M : Tableau
               d'au plus NB_LIGN x NB_COL entiers */
```

*Utilisation de la déclaration de type synonyme :* **typedef**

```
typedef int t_matrice [NB_LIGN][NB_COL] ;
          /* On définit le nouveau type t_ent synonyme
             de tableau de NB_LIGN x NB_COL entiers */
```

*Utilisation du type* **t_matrice** *pour déclarer la variable* **M**

```
t_matrice M ; /*Tableau d'au plus NB_LIGN x NB_COL entiers */
```

## 8.6. Exercice

On considère une variable tableau **Liste** contenant **N** valeurs réelles **distinctes** significatives. Ecrire en langage algorithmique les séquences permettant de déterminer si une valeur appartient à ce tableau. On indiquera en plus le **rang** de cette valeur si elle existe.

*Exemple d'écran d'exécution souhaité*

```
Valeur recherchée ?     10.6
10.6 n'existe pas

Valeur recherchée ?     -51.5
-51.5 est la 7 eme valeur contenue dans Liste.
```

**CONST** N_MAX = 10

**TYPE** t_reels = Tableau[N_MAX] de Réels

**VAR**

Liste : t_reels /*Tableau d'au plus N_MAX réels */

i : Entier /* Indice d'accès aux valeurs du tableau */

N : Entier /* Nombres de valeurs significatives (N <= N_MAX) */

Valeur : Réel /*valeur recherchée */

Pos : Entier /* rang de la valeur recherchée */

**Début**

**/* Les contenus de N et de Liste sont supposés connus */**

**/* Saisie de la valeur recherchée */**

.....

/* Traitement */

.....

.....

.....

.....

.....

.....

/* Edition */

.....

.....

.....

.....

.....

.....

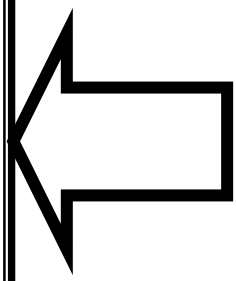
.....

.....

**Fin**

Afin de bien distinguer les variables du programmes principal (paramètres effectifs) des paramètres formels (interface de communication d'un sous programme), on adoptera la convention suivante concernant leurs identificateurs :

- **Paramètres effectifs** : la première lettre est systématiquement **en majuscule** (à l'exception des variables compteurs d'itération i, j, k, de type **Entier**)
- **Paramètres formels** : toutes les lettres sont en minuscule



# 9. SOUS-PROGRAMMES

## 9.1. Intérêt des sous-programmes

.....

.....

.....

*Il est alors possible de créer des bibliothèques de sous-programmes*.

- .....

.....

- Un programme peut alors être composé d'un ensemble fini de sous-programmes.

## 9.2. Procédures

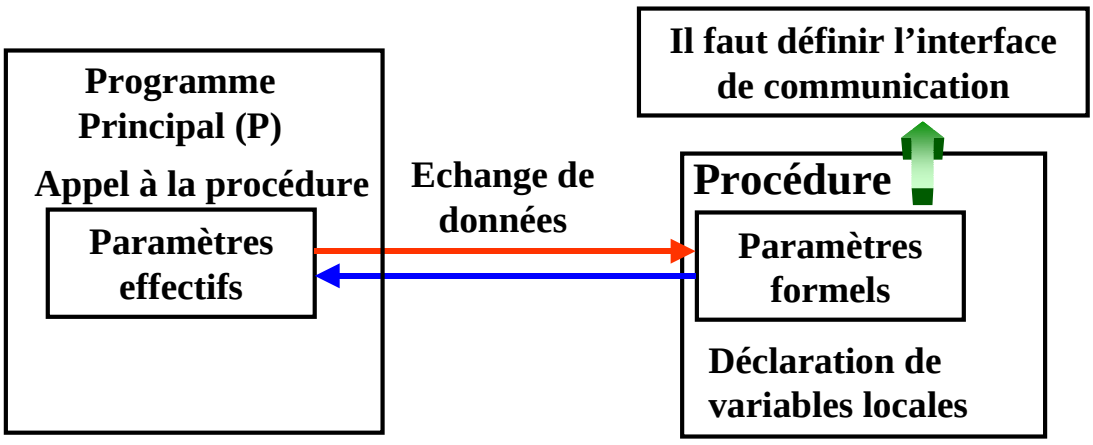
### 9.2.1. Définition des paramètres formels, effectifs, locaux

- .....

.....

.....

.....




Lorsque cela est nécessaire, les sous-programmes (SP) utilisent des variables supplémentaires qui leur sont propres pour exécuter les tâches désirées :

.....

.....

- Le partage et l'échange de données entre le programme principal et les sous-programmes s'effectuent à l'aide des **paramètres formels** du sous-programme (aussi appelé arguments).

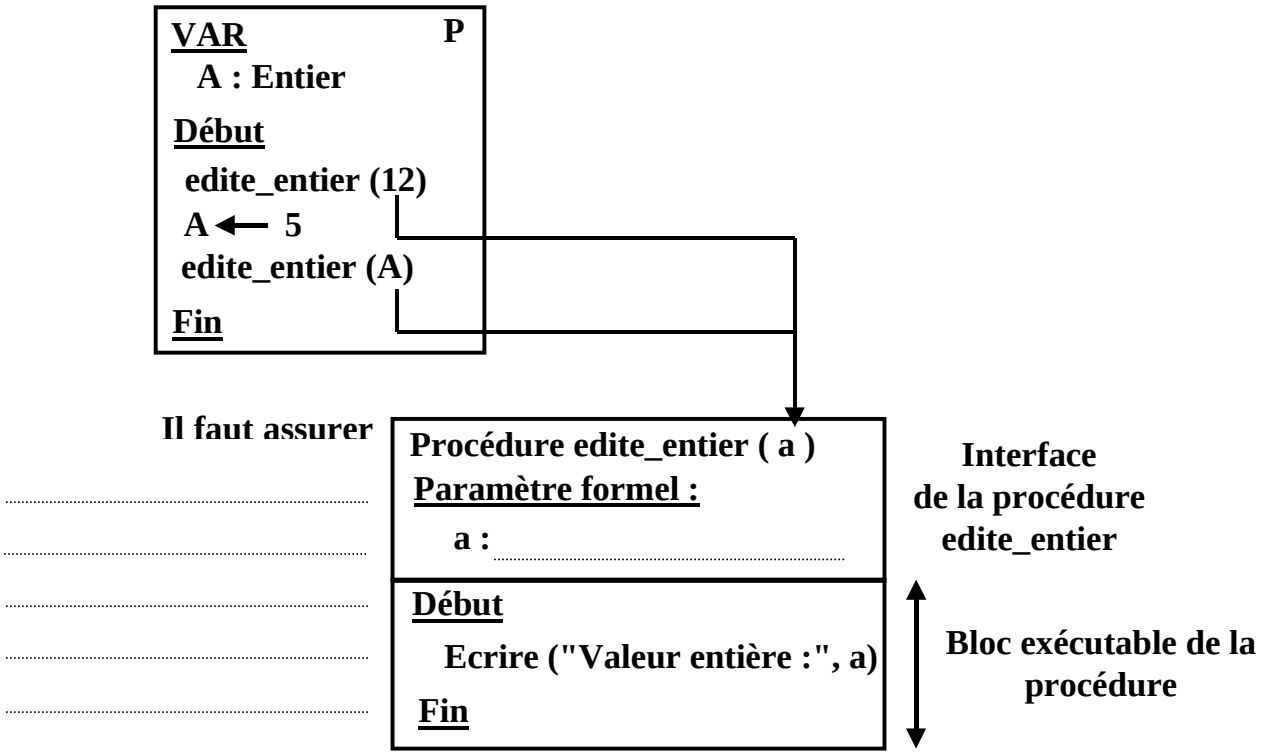
  Il existe 3 catégories de **paramètres formels** :

  1. .....
  2. .....
  3. .....

## 9.2.2. Paramètres formels d'entrée

- Un paramètre d'entrée { E } permet de .....

.....




- Que se passe-t-il en C lors du passage des paramètres effectifs pour les paramètres d'entrée { E } ?

.....

.....

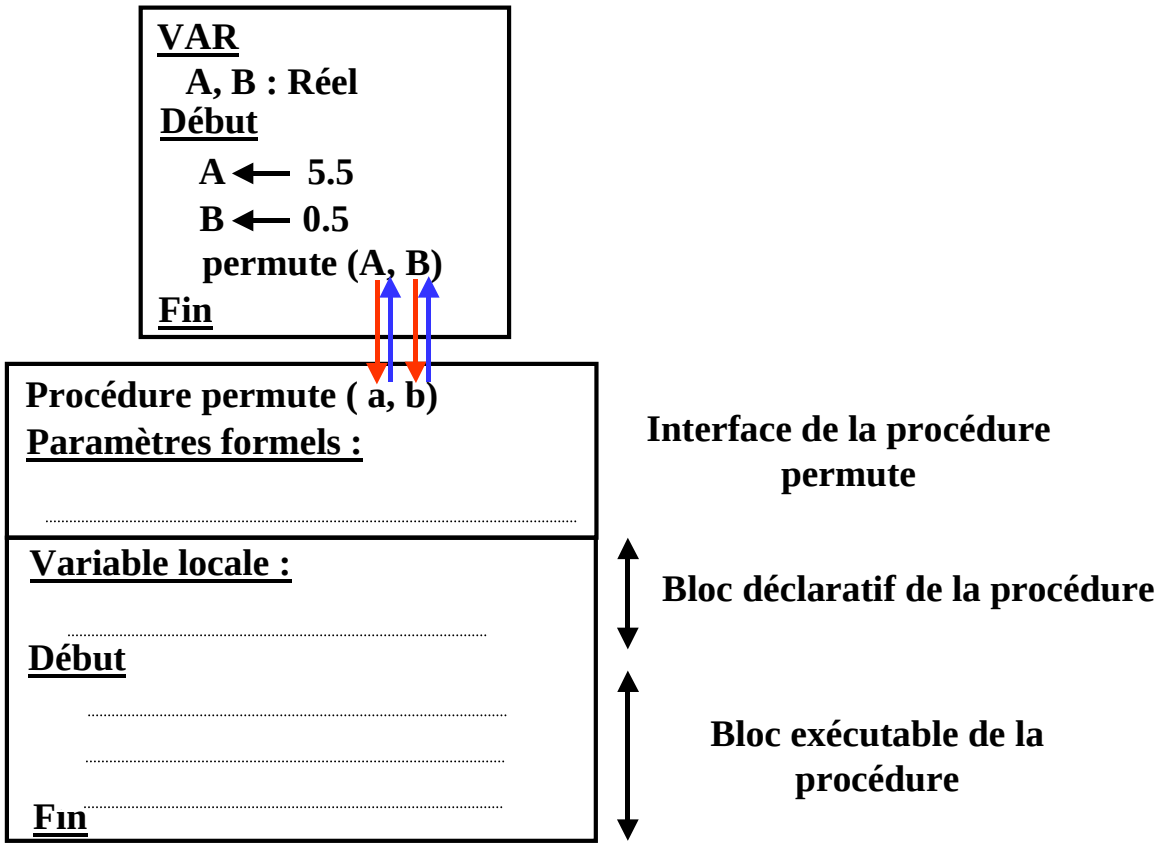
.....

### 9.2.3. Paramètres formels d'entrée/sortie

- Un paramètre d'entrée/sortie { ES } permet de .....

.....

.....




- Que se passe-t-il en C lors du passage des paramètres effectifs pour les paramètres d'entrée/sortie { ES } ?

.....

.....

Le sous-programme appelé peut donc s'exécuter en travaillant **directement** sur la zone mémoire des **paramètres effectifs**.

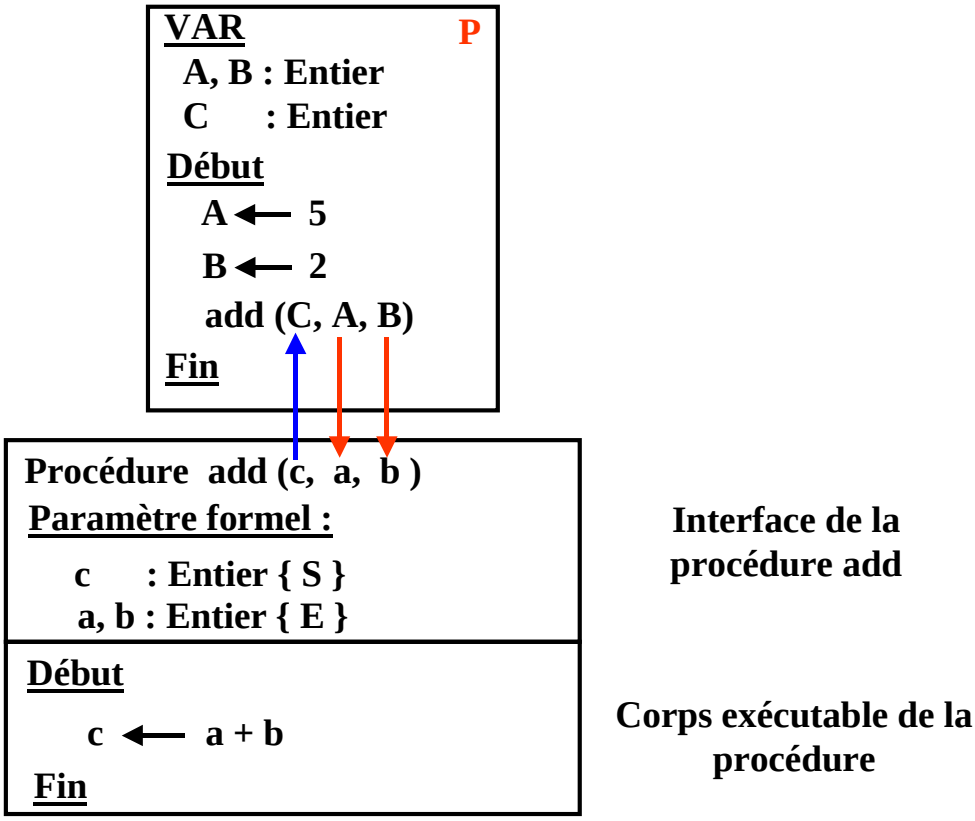
Dans l'exemple ci-dessus, cela signifie que toutes les actions décrites par les séquences effectuées sur **a** et **b** sont équivalentes à des actions menées sur **A** et **B**.

Il faudra en C spécifier par une syntaxe particulière que l'on transmet non pas les valeurs de **A** et **B** mais **leurs adresses** et que **a** et **b** ne sont pas exactement des variables réelles mais des variables capables de recevoir les **adresses** des variables **A** et **B** (pointeurs de réel) permettant d'accéder aux contenus de celles-ci.

### 9.2.4. Paramètres formels de sortie

Un paramètre de sortie { S } .....

.....




- Que se passe-t-il en C lors du passage des paramètres effectifs pour les paramètres de sortie { S } ?

.....

.....

Un paramètre de sortie peut être vu comme un paramètre d'entrée/sortie **dont on n'utilise pas la valeur d'entrée**.

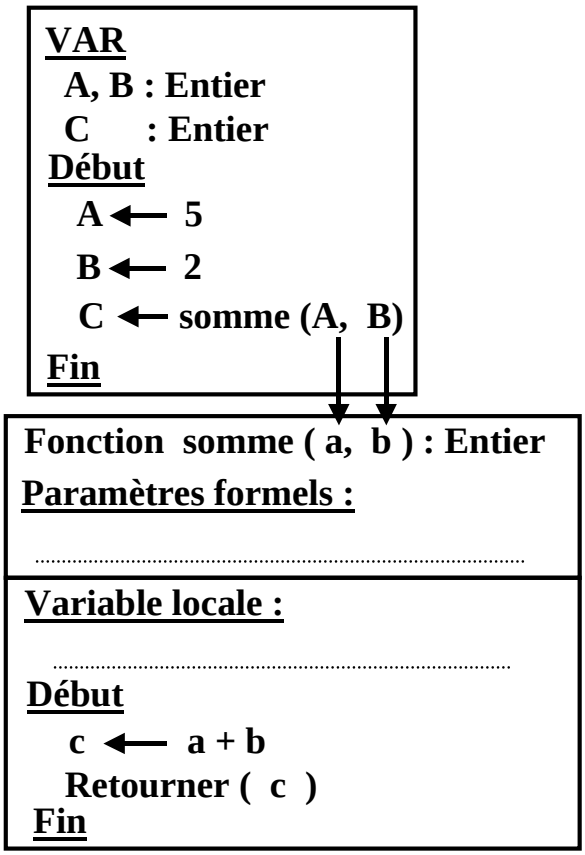
## 9.3. Fonction

- On peut choisir qu'un **sous-programme retourne un résultat** en remplaçant, par exemple, l'échange de donnée réalisé par un paramètre de sortie { S } par le **retour d'une valeur équivalente**.

### On parle alors de fonction

.....

.....




La variable locale **c** est utilisée pour retourner la valeur qu'elle contiendra.

Cette valeur retournée peut être directement affectée (à la fin de l'exécution de la fonction) à une variable du Programme principal (P) appelant.

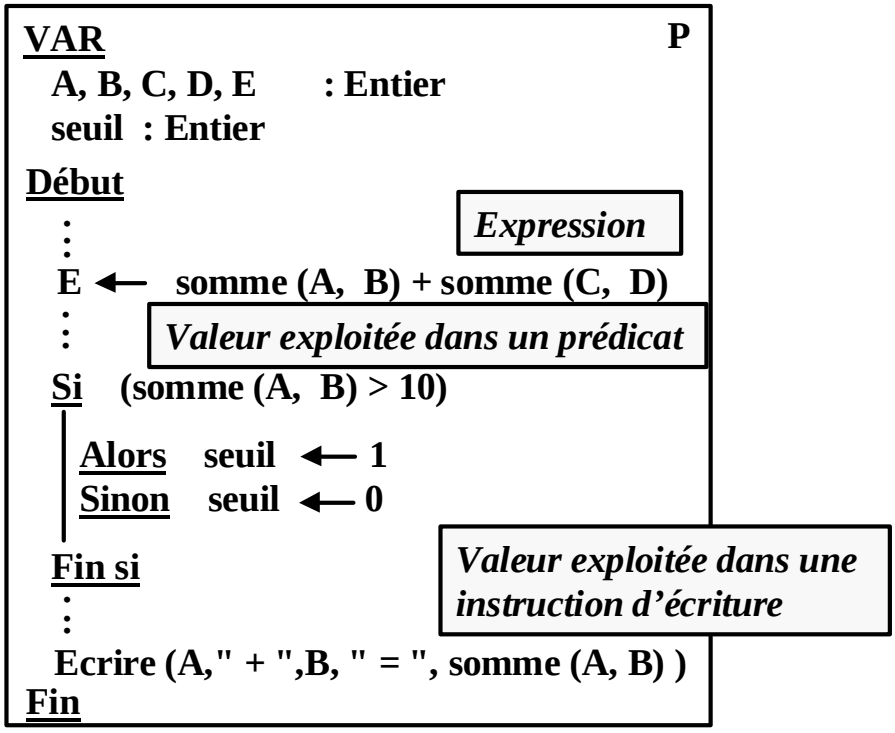
- Que se passe-t-il en langage C lors du retour d'une valeur ?

.....

.....

.....

- L'intérêt de la fonction est que l'on peut utiliser directement la valeur retournée dans toute expression ou instruction d'écriture.




## 9.4. Elaboration de l'en-tête (interface de communication) d'un SP

Pour pouvoir développer correctement l'interface de communication d'un SP, il faut avoir répondu à un certain nombre d'interrogations :

-1- quel nom donner au SP ?

*Donner un nom en respectant les règles de création d'un identificateur.*

-2- de quelle(s) donnée(s) du programme principal le SP a-t-il besoin ?

*Préciser l'utilité de cette (ces) donnée(s), en préciser le type.*

-3- le SP doit-il modifier cette (ces) donnée(s) ?

*Si c'est NON, utiliser des paramètres formels d'entrée* { E }

*Si c'est OUI, utiliser des paramètres formels d'entrée/sortie* { ES }

-4- quelle(s) donnée(s) le SP doit-il créer et transmettre au programme principal ?

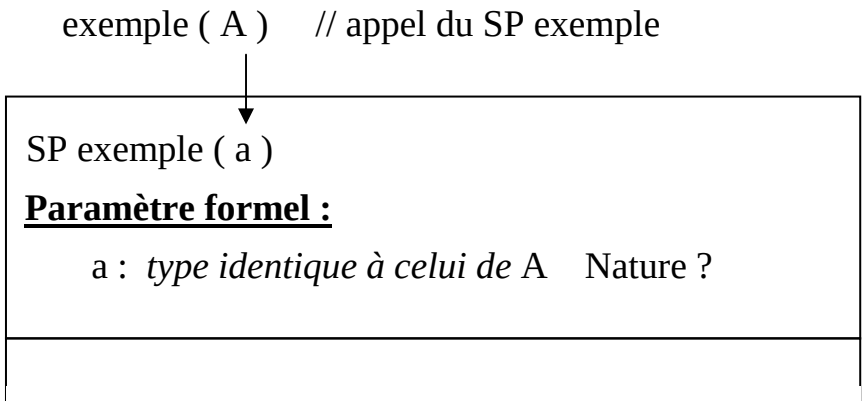
*Décrire cette (ces) donnée(s), en préciser le type*

*Utiliser des paramètres formels de sortie* { S }.

-5- le SP doit-il retourner **directement** une valeur ?

*Si c'est* OUI, *création* **d'une fonction**

## 9.5. Comment déterminer facilement la nature d'un paramètre formel d'un sous-programme ?




Un moyen simple de déterminer la **nature** d'un paramètre formel **a** d'un sous-programme chargé de gérer le paramètre effectif **A** d'un programme est de répondre à deux questions :

1°) Le paramètre formel **a** a-t-il besoin de la **valeur** du paramètre effectif **A** pour effectuer le traitement ?

Oui ➔ { E }

2°) Le SP **modifiera-t-il** la valeur du paramètre formel **a** **retransmis** au paramètre effectif **A** ?

Oui ➔ { S }

Si la réponse est **Oui** aux 2 questions, on a donc un paramètre formel qui est à la fois d'entrée et sortie {ES}

## 9.6. Exercice

Ecrire une procédure qui renvoie la partie entière et la partie décimale d'un réel positif (*on incrémente à partir de* 0 *un entier dans une boucle*).

Proposition d'appel :

partie_entiere ( Nombre, Partie_ent, Partie_dec)

Si Nombre contient 15.658,
après l'appel de la procédure
Partie_ent = 15 et
Partie_dec = 0.658.

| | |
|---|---|
| ..... | Interface de la procédure |
| ..... | Corps exécutable de la procédure |

# 10. LES POINTEURS EN C

## 10.1. Introduction

Pour gérer les paramètres formels d'entrée/sortie { ES } et de sortie { S }, le langage C.....

- On utilise pour cela des variables particulières permettant de stocker une **adresse** d'**entier**, une **adresse** de **réel**, une **adresse** de **caractère**.
- Ces variables particulières sont appelées variables pointeurs.

## 10.2. Adresse d'une variable

.....

.....

.....

L'adresse de la variable correspond à .....

.....

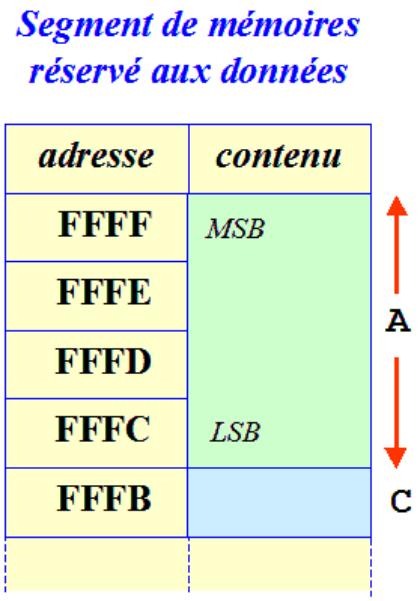
**En langage C, l'opérateur adresse & fournit l'adresse d'une variable en mémoire.**

*Exemple :*

```
float A; // Bloc déclaratif
char C ;
```

`&A` désigne l'adresse de `A` soit : .....

`&C` désigne l'adresse de `C` soit : .....




- Un pointeur est l'adresse mémoire d'une variable d'un type déterminé. On dit que le pointeur pointe sur cette variable.

  `&A` est donc un pointeur de réel de valeur .....

  `&A` pointe sur la variable `A`

  `&C` est donc un pointeur de réel de valeur .....

  `&C` pointe sur la variable `C`

## 10.3. Variable pointeur

### 10.3.1. Définition

- .....

.....

La déclaration d'une variable pointeur pointant sur une variable de type donné s'effectue à l'aide du symbole ***** :

*Nom_de_type* *******Identificateur_de_variable*

*Exemple :*

```
float *p_reel; /* Variable pouvant contenir
                  l'adresse d'une variable de type float */
char *p_car  ; /* Variable pouvant contenir
                  l'adresse d'une variable de type char */
```

### 10.3.2. Accès au contenu d'une adresse pointée par une variable pointeur

L'opérateur ***** désigne le contenu de l'adresse pointée par une variable pointeur :

*Exemple :*

```
float A ;        // Bloc déclaratif
char C  ;
float *p_reel;
char  *p_car ;

p_reel = &A ;    // Instructions
p_car = &C ;
*p_reel = 10.5;
```

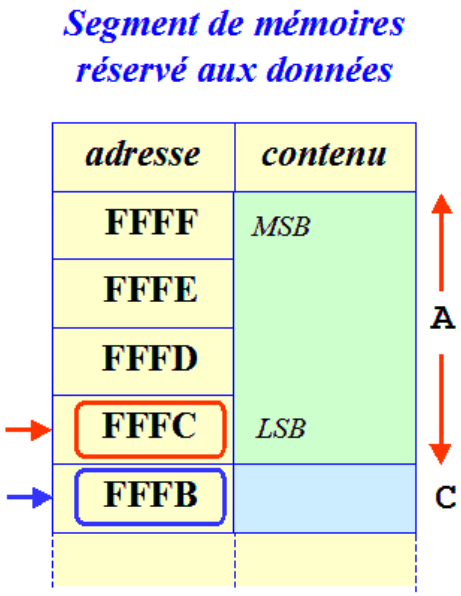
Instructions équivalentes

.....

```
*p_car =  'Z';
```

Instructions équivalentes

.....




La variable pointeur **p_reel** permet donc de manipuler la variable **A**.

La variable pointeur **p_car** permet donc de manipuler la variable **C**.

### 10.3.3. Arithmétique des pointeurs

Lorsqu'on manipule une variable pointeur, il est **impératif** que celle-ci contienne l'adresse d'une variable parfaitement définie.

Il est alors possible d'ajouter ou de soustraire une valeur entière **i** à une variable pointeur **p** pointant sur une variable occupant ***n*** octets.

- p + i désigne alors l'adresse évaluée de la manière suivante : p + *n* x i
- p – i désigne alors l'adresse évaluée de la manière suivante : p – *n* x i

***Remarque*** : Il est très dangereux de laisser une valeur aléatoire dans une variable pointeur, car toute manipulation à partir de cette variable pointeur risque de modifier de manière intempestive le contenu des octets auxquels fait référence cette valeur d'adresse aléatoire.

*Exemple :*

```
// Bloc déclaratif
char Car1, Car2, Car3;
short V1, V2 ;
// Entiers codés sur 2 octets
char *p_car ;
short *p_sh;
// Instructions
p_car = &Car2 ;        p_car =
p_car = p_car-1;       p_car =
*p_car = 'I' ;
p_car = p_car+2 ;      p_car =
*p_car = 'T' ;
p_sh = &v1 ;           p_sh =
p_sh = p_sh – 1 ;      p_sh =
*p_sh = -15 ;
p_sh = p_sh + 2 ;      p_sh =
```

***Segment de mémoires réservé aux données***

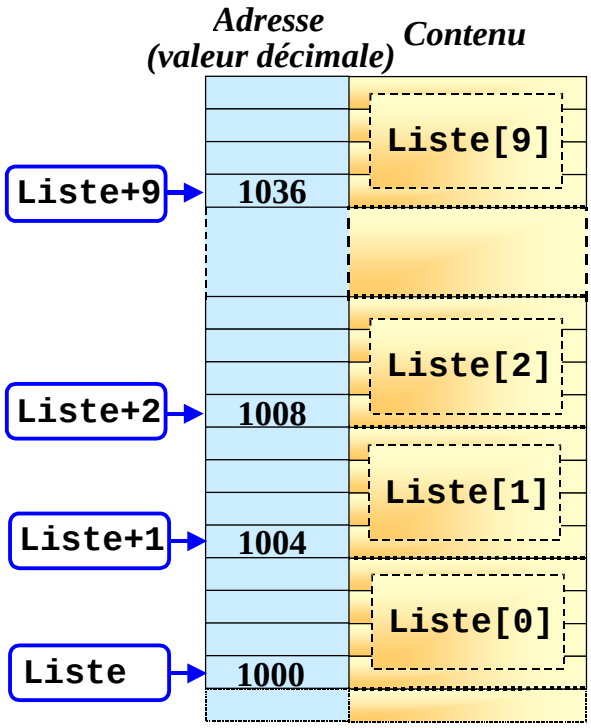
| *adresse* | *contenu* | |
|---|---|---|
| FFFF | 'I' | Car1 |
| FFFE | 'U' | Car2 |
| FFFD | 'T' | Car3 |
| FFFC | 86 | V1 |
| FFFB | | |
| FFFA | -15 | V2 |
| FFF9 | | |

**Attention : opération périlleuse !**
**La variable pointeur ne pointe plus sur une variable de type `short`**

## 10.4. Tableaux à une dimension et pointeurs

- **En déclarant un tableau en C, on définit automatiquement un pointeur** :

```
float Liste[10] ;
/* Tableau d'au plus 10 réels */
```




Cette déclaration .....

.....

.....

**`Liste`** désigne un pointeur de réel .....

.....

.....

.....

**`Liste`** contient donc l'adresse de **`Liste[0]`** soit : **`&Liste[0]`** .

Dans l'exemple ci contre, **`Liste`** = .....

- **Arithmétique des pointeurs appliquée aux tableaux**

Il est possible d'additionner un entier **i** à un pointeur ou une variable pointeur. On utilise cette possibilité pour accéder aux différents éléments d'une variable de type tableau :

**`Liste+i`** désigne l'adresse du (**i+1**)$^{éme}$ élément. Cette adresse est évaluée de la manière suivante :

**`Liste`** + **`i`***__taille(élément)__

où **taille(élément)** représente la taille (en octets) de chaque élément du tableau.

Dans l'exemple présenté, les valeurs **décimales** des adresses suivantes sont : Liste + 1 = .....

Liste + 2 = .....

Liste + 9 = .....

**`*(Liste+i)`** *désigne le contenu de l'adresse pointée par* **`Liste+i`** *et donc la notation* **`*(Liste+i)`** *est équivalente à* **`Liste[i]`**.

***Ecritures équivalentes :***

| | | | | | |
|---|---|---|---|---|---|
| **`*Liste`** | ⇔ | **`Liste[0]`** | **`Liste`** | ⇔ | **`&Liste[0]`** |
| **`*(Liste+i)`** | ⇔ | **`Liste[i]`** | **`Liste+i`** | ⇔ | **`&Liste[i]`** |

On peut accéder aux éléments de Liste en déclarant une autre variable **pointeur de réel** :

```
float *t ; // variable pointeur de réels
t = Liste ; // preel reçoit l'adresse du 1er
                // élément du tableau Liste
```

**`t[i]`** *ou* **`*(t+i`**) *désigne le contenu de l'adresse pointée par* **`t+i`** *c'est à dire :* **`Liste[i].`**

# 11. PROCEDURES ET FONCTIONS EN C

## 11.1. Structure d'un programme écrit en C

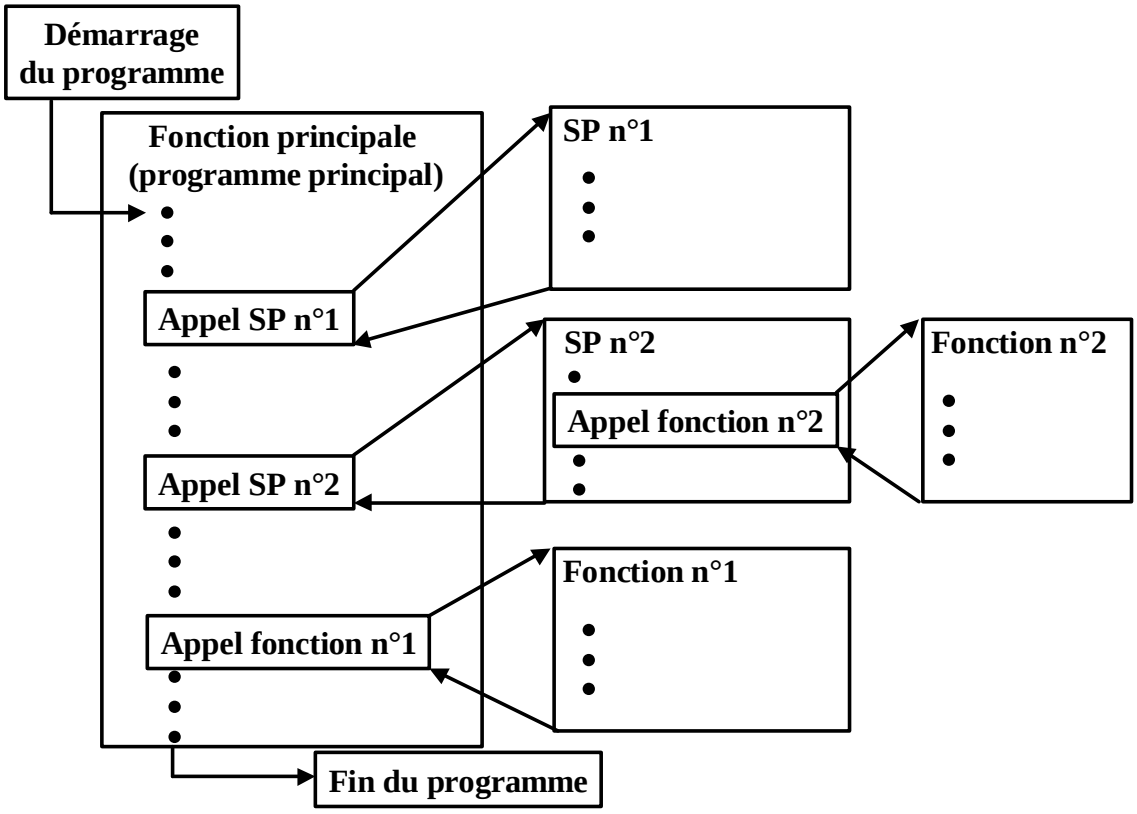
Un programme écrit en C est structuré en modules constitués de :

- la fonction principale de nom **`main`** : (programme principal P). C'est la **première fonction** qui est exécutée lors du lancement du programme. Par convention, celle-ci doit **retourner la valeur 0** à la fin de son exécution qui marque **la fin du programme**.
- sous-programmes (procédures et fonctions) :

  Ces modules peuvent être appelés soit par la fonction principale, soit par d'autres procédures ou fonctions.

  - Procédure : module ne retournant rien.
    *Exemple d'appel de SP (procédure) :*
    ```
    edite_entier (A) ; // Edite la valeur de A
    ```
  - Fonction : module retournant une valeur, un résultat :
    *Exemple d'appel de fonction :*
    ```
    C = somme (A, B) ; // Calcule C = A + B
    ```

***Déroulement d'un programme :***




## 11.2. Ecriture de procédures en C

Les procédures ne retournent pas de résultat au module appelant. Pour l'indiquer, on utilise le mot clé **`void`** dans l'écriture de son en-tête.

***Remarque*** : On peut interpréter une **procédure** comme une **fonction** qui ne **renvoie rien**.

Nom du module

```
void ma_procedure ( Liste de paramètres formels )
{
// Variables locales
   float a ;
   int   n ;

// Corps exécutable
   .
   .
   .
}
```

*Bloc déclaratif*

*Bloc exécutable*

En-tête de la procédure

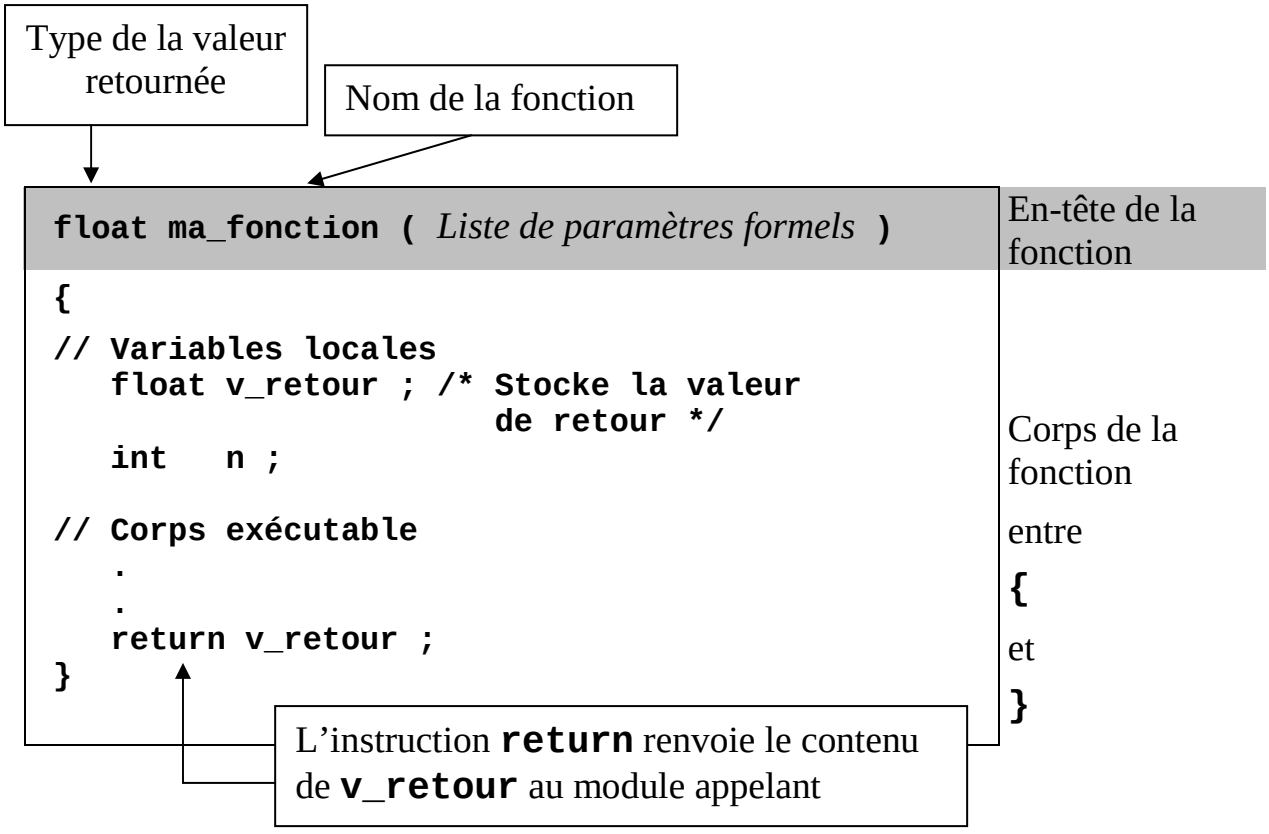
Corps de la procédure entre **{** et **}**

## 11.3. Ecriture de fonctions

Les fonctions retournent un résultat au module appelant.

Pour l'indiquer, on précise le **type** de la valeur retournée dans l'écriture de son en-tête.

*Exemple de fonction renvoyant un réel*



## 11.4. Passage de paramètres par valeur

Un exemple en langage algorithmique :

```
Fonction somme ( a, b ) : Entier
Paramètres formels :
   a, b : Entier { E }
Variable locale :
   c : Entier
Début
   c ← a + b
   Retourner ( c )
Fin
```

**Traduction en langage C :**

*Liste de paramètres formels*

```
int somme(int a, int b)
{
// Variable locale
   int c ; // pour le retour
// Corps exécutable
   c = a + b ;
   return c ;
}
```

```
int main() // prog. ppal
{
 // variables du prog.ppal
    int A, B, C ;
    ...
 // appel de la fonction
    C = somme (A, B) ;
    ...
   return 0 ;
}
```

## 11.5. Passage de paramètres par adresse

Un exemple en langage algorithmique :

**Procédure permute ( a, b)**
**Paramètres formels :**
**a,b : Réel { ES }**

**Variable locale :**
**aux : Réel**
**Début**
**aux ← a**
**a ← b**
**b ← aux**
**Fin**

**Traduction en langage C :**

*Liste de paramètres formels*

```
void permute(float *a, float *b)
{
 // Variable locale
    float aux ;

 // Corps exécutable
    aux = *a ;
    *a = *b  ;
    *b = aux ;
}
```

```
int main() // prog. ppal
{
// variables du prog. ppal
   float A, B;
    .
 // appel de la procédure
   permute (&A, &B) ;
    .
    .
    .
   return 0 ;
}
```

**a** et **b** doivent être des variables capables de recevoir les adresses de variables réelles (variables pointeurs de réel)

Lors de l'appel, **l'adresse** de **A** est recopiée en **a**, **l'adresse** de **B** est recopiée en **b**.

La procédure inverse les contenus ***a**, ***b** des deux adresses.

Les contenus de **A** et de **B** sont donc inversés à la fin de l'exécution de **permute**.

## 11.6. Cas particulier : aucun passage de paramètres

Lorsqu'aucun échange n'est réalisé par passage de paramètres, la liste des paramètres formels est remplacé par le mot-clé **`void`**.

```
void procedure_sans_par(void)
{
 .
 .
 .
}

float fonction_sans_par(void)
{
   float v_retour ;
   .
   .
   .
   return v_retour ;
}
```

```
int main() // prog. ppal
{
// variables du prog. ppal
   float A;
   .
   .
   procedure_sans_par();
   .
   .
   A = fonction_sans_par();
   .
   .
   return 0 ;
}
```

Attention ! Lors de l'appel d'un sous-programme ou d'une fonction sans paramètres, il ne faut pas oublier les parenthèses !

## 11.7. Notion de prototype

- Le prototype d'une fonction qui retourne un résultat ou qui ne renvoie rien (procédure) est **son en-tête**.

*Exemple :*

```
int somme(int a, int b) ;
void permute(float *a, float *b) ;
```

- Les prototypes permettent donc de préciser la syntaxe d'utilisation d'une fonction lors de son appel (son nom, la liste et le nombre des paramètres effectifs à passer, la valeur éventuellement retournée), mais pas les traitements qui sont effectués.
- Il est possible de déclarer le prototype d'une fonction avant d'écrire son développement complet (en-tête + corps dans lequel on décrit ce que fait la fonction).

  Cela permet au compilateur de vérifier que la syntaxe d'appel est correcte dans la fonction principale !

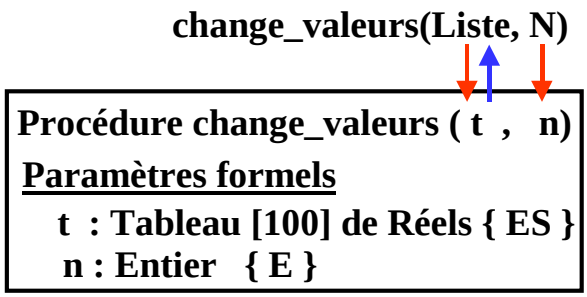
```
int somme(int a, int b) ;
void permute(float *a, float *b) ;
int main () // prog. ppal
{
   float V1, V2 ;
   int A, B, C ;
   .
   .
   C = somme (A, B)  ;        Appel correct
   permute (&A, B, C) ;       Appel incorrect
   permute (&A, &B )  ;       Appel correct
   .
   .
   return 0 ;
}
```

## 11.8. Passage de paramètres tableau

### 11.8.1. Passage par adresse

Supposons que l'on désire écrire une procédure qui **change** les N valeurs significatives (**en doublant chaque valeur**) d'un tableau à une dimension de réels **Liste** défini dans un programme principal.

L'appel et l'écriture de l'en-tête du sous-programme ont la forme ci-contre :




**Utilisation directe d'un pointeur :**

```
#define TAILLE 10
typedef float t_reels[TAILLE] ;
void change_valeurs(float *t, int n) ; // prototype
int main () // prog.ppal
{
   t_reels Liste ;
   int  N;
   .
   change_valeurs(Liste, N) ;
   .
   return 0 ;
}
void change_valeurs(float *t, int n) {
// variable locale
   int i;
 // Corps exécutable
   for (i = 0; i < n; i++) {
      t[i] = 2*t[i] ;
   }
}
```

désigne l'adresse du premier élément **`&Liste[0]`**

On accède aux contenus des adresses pointées par **`t`** donc aux valeurs **`Liste[i]`**

**Utilisation de la référence à un type synonyme :**

```
#define TAILLE 10
typedef float t_reels[TAILLE] ;
void change_valeurs(t_reels t, int n) ; // prototype
int main () // prog.ppal
{
   t_reels Liste ;
   int  N;
   .
   change_valeurs(Liste, N) ;
   .
   return 0 ;
}
void change_valeurs(t_reels t, int n) {
// variable locale
   int i;
 // Corps exécutable
   for (i = 0; i < n; i++) {
      t[i] = 2*t[i] ;
   }
}
```

désigne l'adresse du premier élément **`&Liste[0]`**

On accède aux contenus des adresses pointées par **`t`** donc aux valeurs **`Liste[i]`**

***Commentaires :***

**`t`** désigne alors une variable pointeur adaptée permettant de recevoir l'adresse fixe référencée par **`Liste`**.

Faire référence à un type synonyme est la méthode la plus simple pour transmettre un paramètre tableau à **plusieurs dimensions**.

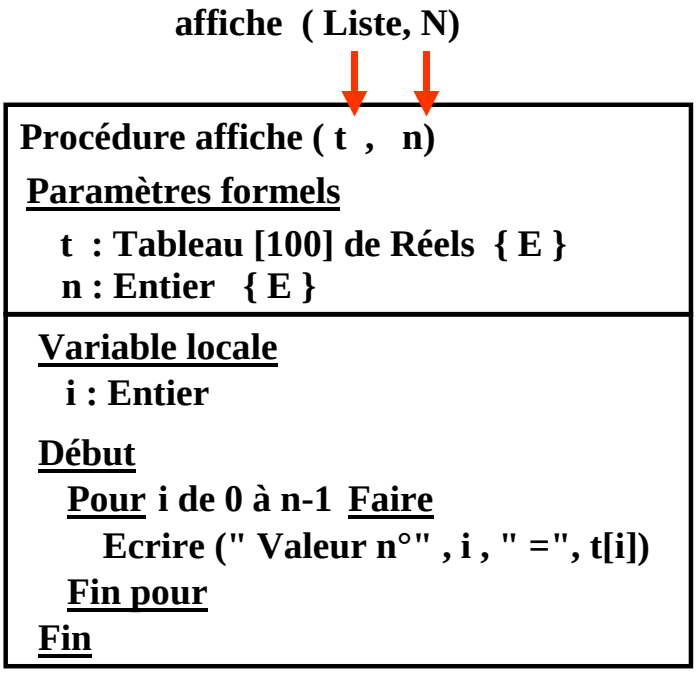
Pour gérer des paramètres formels de type tableau à une dimension, on privilégiera l'utilisation de **pointeurs** plutôt que la référence à un type synonyme, car c'est la méthode la plus universelle (elle ne nécessite pas la définition du type synonyme dans le programme).

### 11.8.2. Passage par valeur

En langage C, le nom d'un tableau désignant l'**adresse** de son premier élément, les variables tableaux sont donc **systématiquement transmis par adresse**.

Or, en **langage algorithmique**, il se peut que l'on désire transmettre un tableau par un paramètre d'entrée.

Supposons que l'on désire écrire une procédure qui affiche les **N** valeurs significatives d'un tableau à une dimension de réels **Liste** défini dans un programme principal.




La transmission systématique d'un tableau par adresse en C ne pose pas de réel problème. Il suffit simplement d'assurer que les instructions du SP **ne puissent pas modifier** le contenu du paramètre formel tableau, ici **t**.

La transmission de la variable tableau **`Liste`** étant systématiquement effectuée par adresse, il est toutefois possible d'indiquer au compilateur que toute modification du contenu des valeurs de ce tableau est interdite en plaçant le mot-clé **`const`** devant le paramètre formel correspondant (**`t`**) .

**Traduction en C :**

```
   #define TAILLE 10
   typedef float t_reels[TAILLE] ;
   void affiche(const float *t, int n) ;  // prototype
ou void affiche(const t_reels t, int n) ; // prototype
   int main () // prog.ppal
   {
      t_reels Liste ;
      int  N;
      .
      affiche(Liste, N) ;
      .
      return 0 ;
   }
```

# 12. PROGRAMMATION MODULAIRE EN C

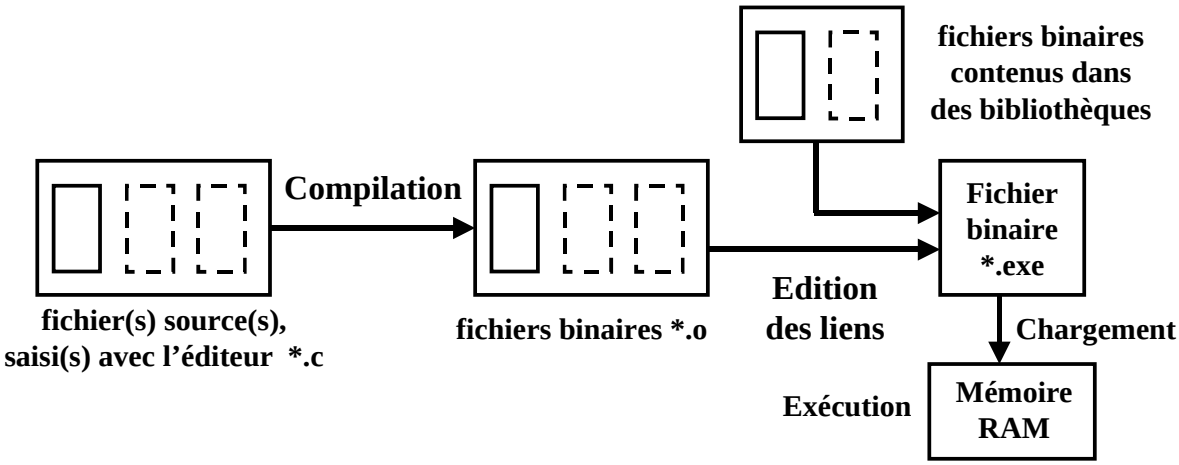
## 12.1. Utilitaires de développement

Lorsqu'on utilise un environnement de développement intégré (tel que Dev-C++), on dispose :

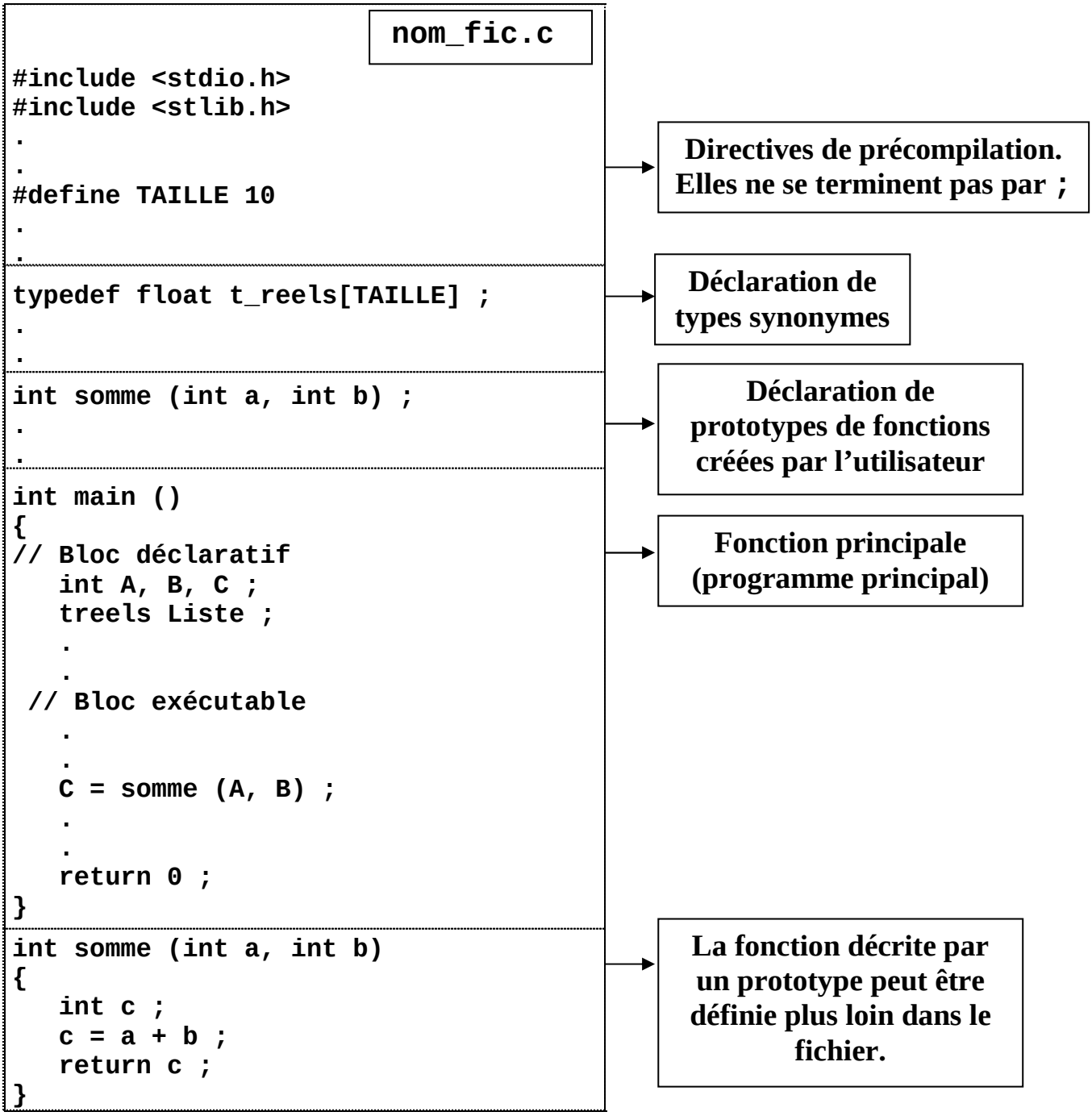
- d'un éditeur de texte, permettant de saisir en langage C les fichiers de texte source (*.**c**),
- d'un **compilateur**, permettant de traduire en code binaire chaque fonction (dont le programme principal : fonction main) écrite dans les fichiers sources, en créant des fichiers binaires ou objets (*.**o**),
- d'un **éditeur de lien** (**linker**), permettant d'ajouter aux fichiers objet des modules de code binaire correspondant à la compilation de fonctions de bibliothèques **appelées** dans les textes sources : fonctions standards, d'entrées-sorties, mathématiques ...

  L'éditeur de lien assure ensuite l'établissement des **connexions** entre le code objet de la fonction principale (**`main`**) avec l 'ensemble des codes objets des différentes fonctions appelées dans la fonction principale pour assurer correctement leur enchaînement.

  Celui-ci fournit alors un fichier binaire **exécutable** (*.**exe**) par la machine après chargement en mémoire.



## 12.2. Structure d'un texte source écrit en C



**Directives de précompilation :**

- La directive **`#include`** permet d'inclure des fichiers interfaces appelés **header** (d'où l'extension **`.h`**) Ceux-ci contiennent la définition de constantes, de types et les prototypes de fonctions appartenant à des bibliothèques alors utilisables dans le programme.
- La directive **`#define`** permet de définir des constantes.

**Compilation :**

- Le compilateur ne lit le fichier source qu'une seule fois du début à la fin pour fabriquer du code binaire (**code objet** : **`nom_fic.o`**).

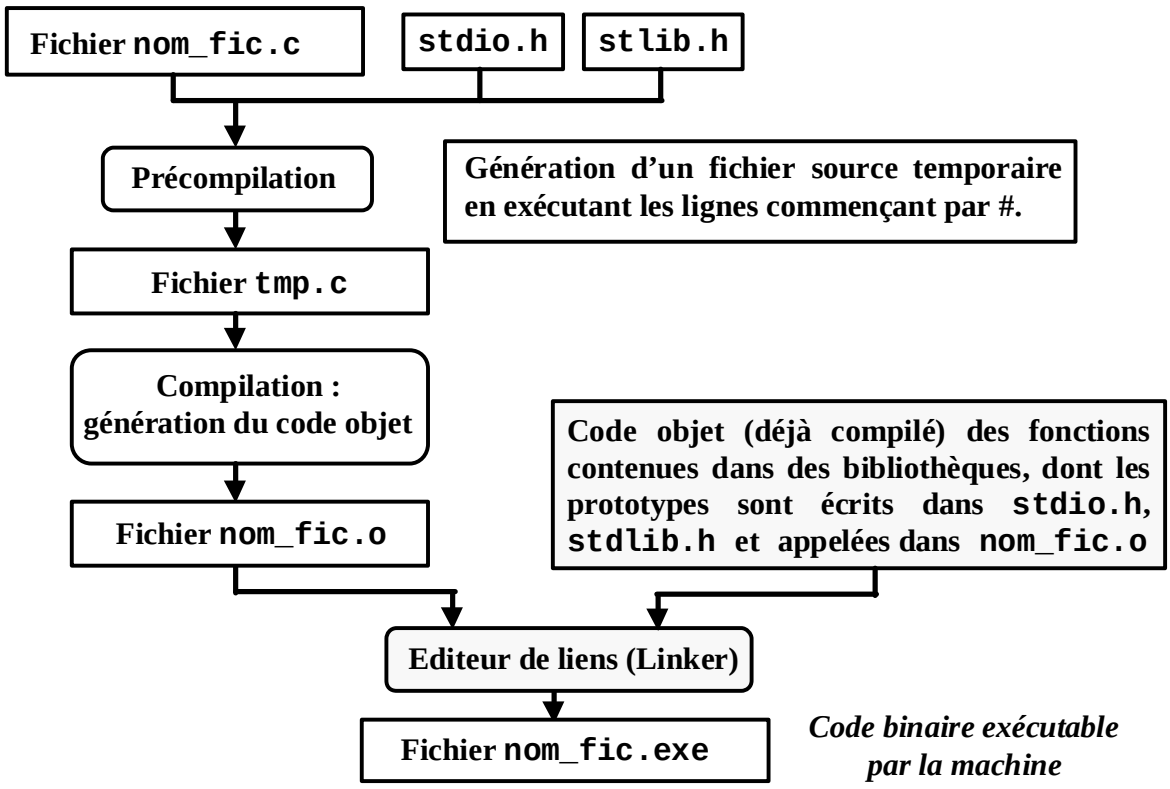
  Lorsqu'il rencontre l'appel à une fonction, il contrôle la cohérence sur le nombre et le type des paramètres transmis avec le **prototype**.

> **REGLE FONDAMENTALE :**
> Toute variable, fonction doit être déclarée avant d'être utilisée.
> Tout symbole ou type synonyme doit être défini avant d'être utilisé.

**Edition de liens :**

- L'éditeur de liens (Linker) permet ensuite de relier le code objet de la fonction principale avec le code objet de la fonction appelée écrit dans le fichier **nom_fic.o**. Il associe également le code objet de fonctions d'usage courant contenu dans des bibliothèques dont les prototypes sont écrits dans les fichiers interfaces de ces bibliothèques : fonctions standards (**`stdlib.h`**), d'entrées-sorties (**`stdio.h`**), etc….
- L'ensemble de ces opérations permet la création d'un code exécutable par la machine (fichier **`nom_fic.exe`**)

## 12.3. Compilation / édition des liens d'un seul fichier source




## 12.4. Principe de la programmation modulaire

### 12.4.1. Méthode

- La programmation modulaire permet de diviser un programme en plusieurs fichiers sources (d'extension .c). Un des fichiers doit obligatoirement contenir la fonction **`main`**.
- Ces différents fichiers constituent chacun un module pouvant être compilé séparément.
- Chaque fichier module peut alors être utilisé par différents programmes.
- On aura intérêt à regrouper dans un fichier module les fonctions de base ainsi que les types utilisés par les différents programmes manipulant les mêmes entités de base.

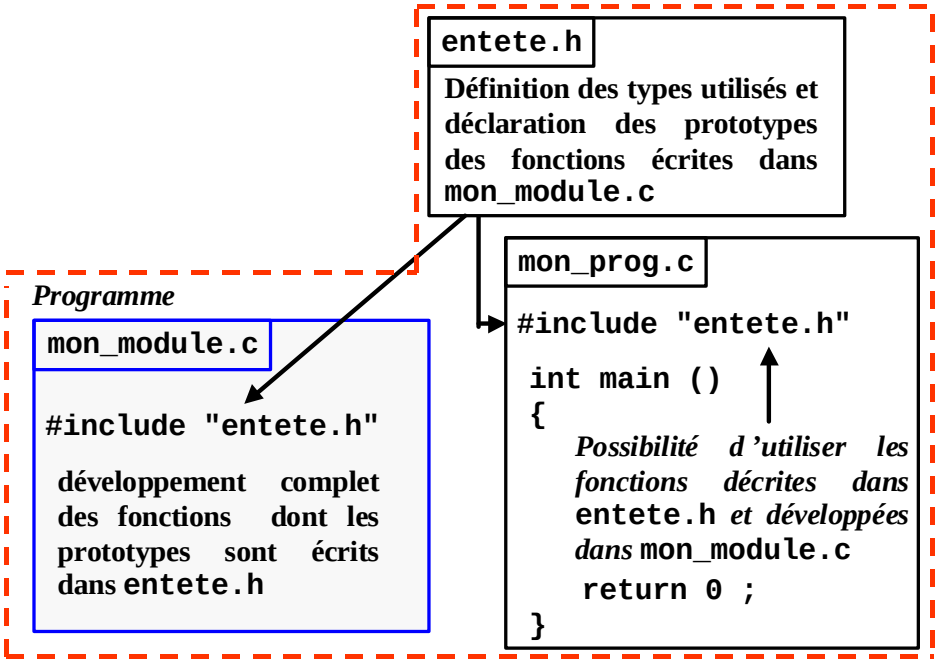
### 12.4.2. Composants d'un programme divisé en modules

Un programme est constitué d'au moins 3 fichiers :

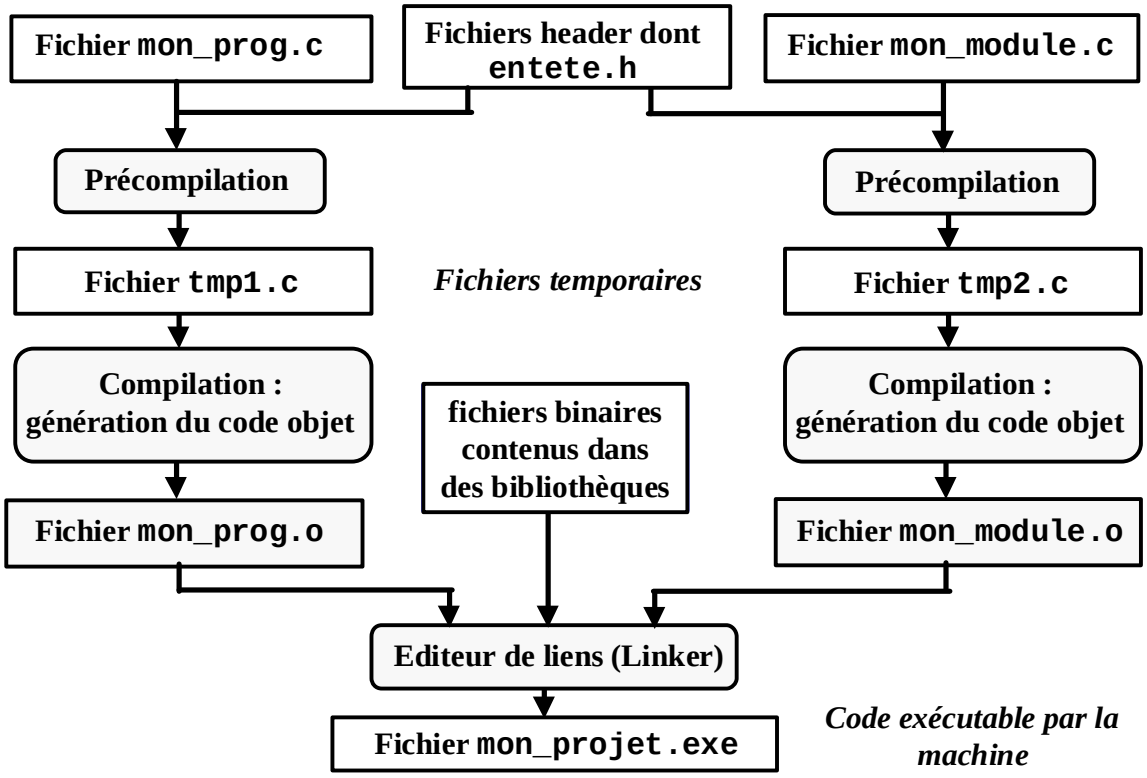
- un fichier interface (d'extension .h) contenant la définition des types et la déclaration des prototypes des fonctions que l'on désire développer;

le fichier interface sera inclus dans tous les fichiers sources concernés,

- un (ou plusieurs) fichier(s) module(s) contenant le développement complet des fonctions que l'on désire utiliser,
- un fichier utilisateur qui contient la fonction **`main`**

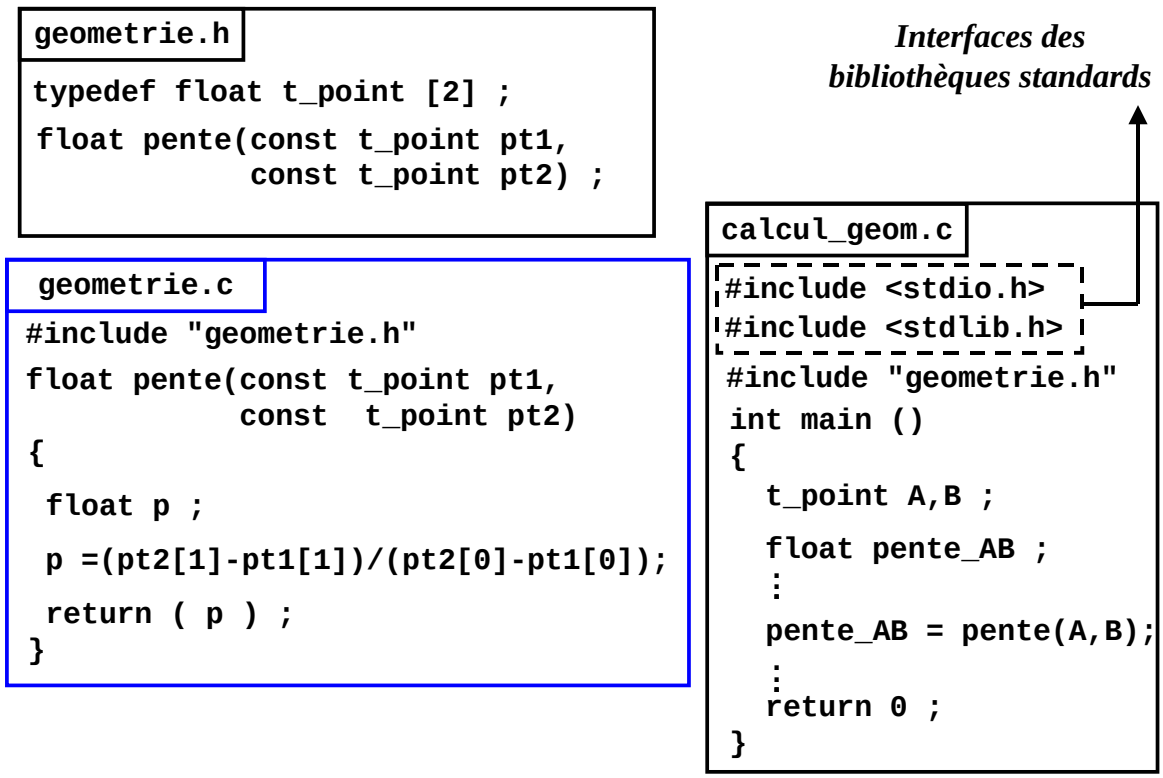



### 12.4.3. Compilation / édition des liens d'un programme divisé en modules



## 12.4.4. Programmation modulaire : un exemple

*Calcul de géométrie plane : on représente les coordonnées euclidiennes (x, y) d'un point par un tableau d'au plus 2 réels avec la convention suivante : si pt désigne un tel tableau, l'abscisse x est stocké dans pt [0] et l'ordonnée y dans p t[1].*




**Compilation + Edition des liens**

⇩

Fichier binaire exécutable : **`calcul_geom.exe`**

*Remarque :*

La directive **`#include`** permet d'inclure des fichiers dans le fichier C courant de deux manières différentes :

**`#include <stdio.h>`** : Le fichier **`stdio.h`** est recherché dans un répertoire de l'environnement de développement où sont les fichiers interfaces des bibliothèques

**`#include "geometrie.h"`** : Le fichier **`geometrie.h`** est recherché dans le répertoire courant.