

INSA

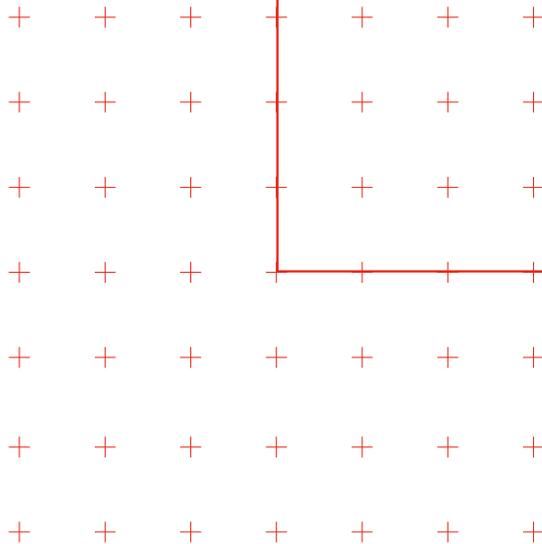
INSTITUT NATIONAL
DES SCIENCES
APPLIQUÉES
CENTRE VAL DE LOIRE

STPI 1A 2024-2025

ALGORITHMIQUE ET PROGRAMMATION 1

Cours et exercices

Camille Baudoin, Moncef
Hidane, Julien Olivier,
Hugo Raguet



Sommaire

1	Introduction à l'algorithmique et la programmation	2
1	Composition d'un ordinateur	2
2	Encodage	3
3	Les logiciels	4
4	Et l'algorithmique dans tout ça?	5
2	Notions de pseudo-langage	7
1	Opérations de base	7
2	Algorithme	8
3	Structure de contrôle	9
4	Tableaux	11
5	Appel de routine	11
3	La programmation en langage C	13
1	De l'algorithme à l'implémentation	13
2	Variables en langage C	13
3	Les entrées et sorties	15
4	Premiers programmes en C	16
5	Opérateurs et expressions	17
6	Structures de contrôle en C	19
7	Les tableaux statiques	22
4	Les fonctions en langage C	26
1	Généralités	26
2	Appel de fonction	27
3	Les tableaux comme paramètres de fonction	29
4	Méthodologie pour écrire une fonction	32
5	Chaînes de caractères en langage C	33
1	Le type <code>char</code>	33
2	Les chaînes de caractères	33
3	Utilisation de la bibliothèque standard	35
A	Exercices et problèmes	36
B	Navigation et compilation en ligne de commande	40
1	Navigation dans l'arborescence	40
2	Programmation et compilation	42

1+ Introduction à l'algorithmique et la programmation

1 Composition d'un ordinateur

Un ordinateur est une machine programmable composée d'éléments électroniques, et capable de réaliser des calculs binaires. Les ordinateurs actuels reposent sur le principe de l'électronique binaire : les composants exploitent des propriétés physiques (tension, courant, etc.) qui ne peuvent prendre que deux valeurs distinctes.

1.1 Composants de l'unité centrale

L'unité centrale d'un ordinateur est composée de :

- Un microprocesseur ou unité centrale de calcul (CPU : *Central Processing Unit* en anglais) : composant qui effectue les opérations binaires.
- Composants mémoires, zone de stockage d'informations. On distingue notamment deux types¹ :
 - la mémoire vive (RAM : *Random Access Memory*) : rapide d'accès mais de volume limité. Elle stocke les informations permettant le fonctionnement de l'ordinateur en temps réel et les données en cours d'utilisation. Cette mémoire est dite volatile : le contenu est perdu lorsque l'alimentation électrique est interrompue.
 - la mémoire morte (notamment le disque dur) : zone de mémoire plus volumineuse mais d'accès plus lent où l'on stocke les fichiers pour sauvegarde.
- La carte mère est le circuit imprimé qui supporte la plupart des composants, et qui assure leur liaison.
- Un microprocesseur graphique (GPU : *Graphics Processing Unit*) : processeur spécialisé pour certains calcul, notamment pour les applications graphiques.

Lorsque l'on exécute un programme, il y a un aller-retour constant d'information entre le microprocesseur qui effectue les instructions et la mémoire vive qui stocke les résultats intermédiaires.

Exemple. Échange d'informations lors de l'exécution un programme simple

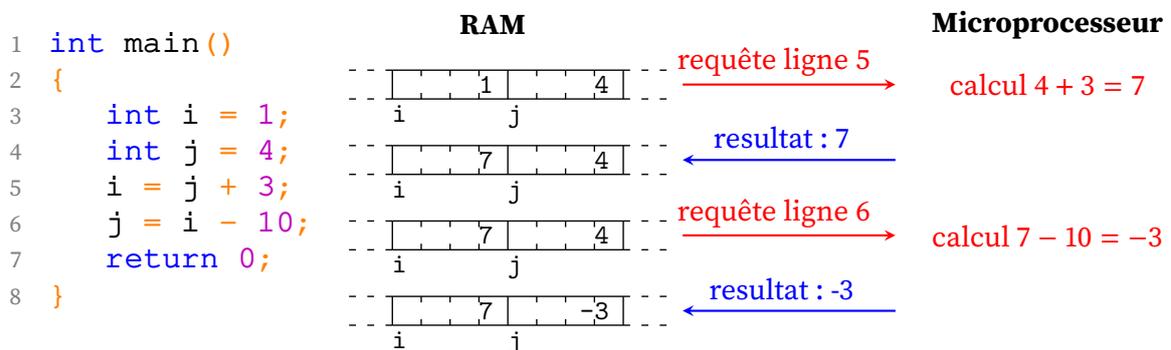


FIGURE 1.1 – Illustration de l'état de la mémoire vive, et des calculs effectués par le microprocesseur à chaque étape du programme dont le code source en C est reporté à gauche du schéma.

1. La composition d'un ordinateur est en réalité plus complexe. Il existe d'autres types de mémoire, notamment la mémoire cache mémoire à haute vitesse et de petite capacité, mémoire intermédiaire placée entre le CPU et la RAM.

Structure de la mémoire

La mémoire d'un ordinateur a une structure linéaire, toutes les informations sont stockées à la suite les unes des autres; on pourra représenter la mémoire comme une bande, illustrée sur la [figure 1.2](#).

La plus petite unité d'information est le *bit* qui ne peut prendre que deux valeurs (0 ou 1). Dans la pratique, les processeurs ou la mémoire n'opèrent pas sur chaque bit individuellement, mais sur des groupes de bits : les *octets*. Un octet est une réunion contiguë de 8 bits.² La capacité mémoire des composants électroniques est mesurée en octets (ko : 10^3 octets, Mo : 10^6 octets, Go : 10^9 octets).

Dans la mémoire, chaque octet possède une adresse. Elle est écrite en hexadécimal : en base 16 indiquée par le préfixe 0x, les chiffres utilisés étant 0–9 puis A–F.

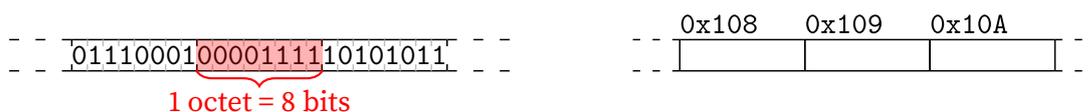


FIGURE 1.2 – Illustration de la mémoire en bits (gauche) et en octets (droite).

1.2 Périphériques

En dehors de l'unité centrale l'ordinateur possède aussi des périphériques, ce qui permet l'interaction de la machine avec l'extérieur (des humains ou d'autres machines) : écran, clavier, souris, disques durs, imprimante, scanner, etc.

2 Encodage

L'ordinateur réalise des opérations uniquement sur des données binaires. Ainsi toute l'information, les données (numérique, image, texte, son) et les instructions (programme, logiciel) doit être traduite de manière binaire pour être enregistrée et manipulée par un support numérique. L'ensemble de ces règles de traduction s'appelle l'*encodage*.

Encodage binaire des nombres entiers

On représente usuellement les entiers naturels avec la numération positionnelle en base 10; par exemple $208 = 2 \times 10^2 + 0 \times 10^1 + 8 \times 10^0$. On peut aussi utiliser la base 2; remarquons qu'elle ne nécessite que deux chiffres, permettant donc d'écrire n'importe quel entier naturel comme une suite de 0 et de 1, comme illustré sur la [figure 1.3](#).

$$13 = 8 + 4 + 1 = 1 \times 2^3 + 1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0$$

--- 00001101 ---

FIGURE 1.3 – Décomposition de l'entier naturel 13 en base 2, et écriture binaire sur 1 octet

La valeur maximale pouvant être encodée dépend du nombre d'octets affectés à la représentation.

Pour les entiers relatifs, un bit est consacré au signe. La représentation usuelle est un peu technique et dépasse le cadre de ce cours.³

2. En réalité, la plus petite unité adressable s'appelle un multiplète (*byte* en anglais) dont la taille peut varier selon les machines. De nos jours, c'est presque systématiquement 8 bits, c'est-à-dire un octet; à tel point que la confusion entre les deux termes est fréquente.

3. Il s'agit du complément à deux, mentionné dans le cours d'électronique numérique de deuxième année; voir aussi l'article Wikipédia https://fr.wikipedia.org/wiki/Complement_a_deux

Tout langage de programmation est associé à un logiciel en charge de traduire le code source en langage machine interprétable par le processeur. Il est important de bien distinguer le code source, un fichier texte compréhensible par l'humain, d'un exécutable, fichier binaire compréhensible par la machine, exemple sur la [figure 1.4](#).

<pre> 1 int main() 2 { 3 int i = 1, j = 4 i = j + 3; 5 return 0; 6 }</pre>	<pre> 00000000 7f 45 4c 46 02 01 01 00 00 00 00 00 00 00 00 00000200 03 00 3e 00 01 00 00 00 40 10 00 00 00 00 00 00000400 40 00 00 00 00 00 00 00 60 36 00 00 00 00 00 00000600 00 00 00 00 40 00 38 00 0d 00 40 00 1d 00 1c 00001000 06 00 00 00 04 00 00 00 40 00 00 00 00 00 00 00001200 40 00 00 00 00 00 00 00 40 00 00 00 00 00 00 00001400 d8 02 00 00 00 00 00 00 d8 02 00 00 00 00 00</pre>
--	---

FIGURE 1.4 – Exemple d'un code source écrit en langage C (à gauche) et extrait du fichier exécutable correspondant représenté en hexadécimal (à droite)

Langages interprétés et langages compilés

On distingue deux grandes familles : les langages interprétés, et les langages compilés. Pour les langages interprétés, la phase de traduction du fichier source vers le langage machine est effectuée à chaque exécution du programme, on parle d'exécution « à la volée » (exemple : Python, GNU Octave et R).

Dans les langages compilés, la phase de traduction du fichier source vers le langage machine est effectuée une seule fois lors de l'étape de *compilation*. Le résultat de cette traduction est stocké dans un fichier exécutable, c'est-à-dire en langage machine (exemple : C et C++).

Le développement dans les langages interprétés est souvent plus rapide que dans les langages compilés mais les performances numériques (rapidité, ressource en mémoire) sont moins grandes. Il n'y a pas nécessairement un langage « mieux qu'un autre »; tout dépend du contexte.

4 Et l'algorithmique dans tout ça ?

Un *algorithme* est une suite finie et non ambiguë d'instructions et d'opérations permettant de résoudre une classe de problèmes. L'algorithmique c'est la science des algorithmes.

Les algorithmes existent en dehors de tout langage de programmation. Lorsqu'un algorithme est traduit dans un langage spécifique, on parle alors de programmation. L'algorithmique se concentre sur la logique, et la programmation sur la mise en œuvre de cette logique.

Par exemple, la méthode de multiplication égyptienne permet de multiplier des nombres entiers en utilisant seulement la table de 2 et des additions. Elle repose sur la propriété suivante :

$$m \times n = \begin{cases} (m/2) \times (2n) & \text{si } m \text{ est pair,} \\ (m-1) \times n + n & \text{sinon.} \end{cases}$$

Cela suggère une marche à suivre : « si m est pair, on substitue respectivement m et n par leur moitié et leur double; sinon on retranche une unité à m et on ajoute n au résultat; puis on recommence jusqu'à ce que m soit nul. »

Ce qui précède décrit un algorithme; mais malgré sa simplicité, la formulation est lourde et manque de rigueur. Pour décrire un algorithme plus efficacement, nous avons besoin d'un langage plus adapté que la langue française. Un langage de programmation est plus adapté mais il est conçu pour communiquer avec un ordinateur. Les contraintes techniques de syntaxe et de gestion de la mémoire viennent interférer avec ce qui nous intéresse : la structure de l'algorithme.

Avant de traduire nos algorithmes dans le langage de programmation C, nous utiliserons alors un compromis : un pseudo-langage, qui, à l'instar d'un langage de programmation, pourra exprimer chaque opération de façon concise et avec le minimum d'ambiguïté, tout en étant débarrassé de la contrainte de faire fonctionner l'algorithme sur une machine.

Exemple. Algorithme de la multiplication égyptienne, écrit en pseudo-langage, puis traduit en C, avec un exemple de programme qui l'utilise.

Algorithme multiplication_égyptienne : $(m, n) \rightarrow p$ selon

$p \leftarrow 0$

Tant que $m \neq 0$ **répéter**

Si $m \setminus 2 = 0$ **alors** $m \leftarrow m / 2, n \leftarrow n \times 2$

sinon $m \leftarrow m - 1, p \leftarrow n + p$.

.

```
# include <insaio.h>
```

```
int multiplication_egyptienne(int m, int n)
```

```
{
```

```
    int p = 0;
```

```
    while (m != 0){
```

```
        if (m%2 == 0){
```

```
            m = m/2;
```

```
            n = n*2;
```

```
        }
```

```
        else{
```

```
            m = m - 1;
```

```
            p = p + n;
```

```
        }
```

```
    }
```

```
    return p;
```

```
}
```

```
int main()
```

```
{
```

```
    int m, n;
```

```
    AFFICHER("Saisir deux entier m et n \n");
```

```
    SAISIR(m, n);
```

```
    AFFICHER("m*n = ", multiplication_egyptienne(m, n), "\n");
```

```
    return 0;
```

```
}
```

*

* *

2+ Notions de pseudo-langage

1 Opérations de base

Dans notre pseudo-langage, une *expression* est formée d'un ou plusieurs *opérateurs* et de leurs *opérandes*. Les opérateurs représentent des opérations, par exemple l'addition (symbole +); les opérandes sont les objets sur lesquels on effectue les opérations.

1.1 Variables simples et affectation

Les *variables* sont des conteneurs qui permettent de manipuler des données. Les variables ont un nom, et une valeur. On définit une variable et on lui affecte une valeur avec l'opérateur d'affectation : $nom \leftarrow valeur$ où *nom* est le nom de la variable et *valeur* est une expression que l'on peut évaluer.

La variable se trouvant à gauche de l'opérateur \leftarrow prend la valeur se trouvant à droite de \leftarrow (valeur littérale, autre variable, etc.). Si *valeur* est une expression, celle-ci est d'abord évaluée, puis son résultat est affecté.

L'opérateur d'affectation permet aussi de modifier la valeur d'une variable existante.

```
i ← 42 # i vaut 42
j ← i # j prend la valeur de la variable i, ici 42
i ← i+1 # l'expression i+1 est d'abord évaluée, elle vaut 43,
# puis i prend la valeur 43
```

1.2 Opérateurs arithmétiques, relationnels, logiques

Opérateurs arithmétiques dans le pseudo-langage

- L'addition +, la soustraction – et la multiplication \times sur les nombres entiers et réels
- La division réelle est notée \div
- Sur les nombres entiers relatifs / et \ dénotent respectivement le quotient et le reste de la division euclidienne

```
x ← 42 ÷ 5 # x vaut 8.4
q ← 42 / 5 # q vaut 8 (division euclidienne de 42 par 5 : 42 = 8×5+2)
r ← 42 \ 5 # r vaut 2
```

Opérateurs relationnels

Les opérateurs relationnels évaluent des relations d'égalité ou des relations d'ordre sur les opérandes, la valeur du résultat étant vrai ou faux

- Les opérateurs $< \leq > \geq$ évaluent les relations d'ordre usuelles
- Les opérateurs $= \neq$ évaluent l'égalité de valeur

Opérateurs logiques

Les opérateurs logiques **et** (conjonction logique), **ou** (disjonction logique) **non** (négation logique) manipulent les *booléens*, c'est à dire des expressions dont la valeur est vrai ou faux. La signification de ces opérateurs est rappelée dans les tables de vérité de la [figure 2.1](#). Les opérateurs logiques permettent de construire des expressions logiques complexes à partir d'expressions élémentaires.

p	non(p)
vrai	faux
faux	vrai

p	q	p et q
vrai	vrai	vrai
vrai	faux	faux
faux	vrai	faux
faux	faux	faux

p	q	p ou q
vrai	vrai	vrai
vrai	faux	vrai
faux	vrai	vrai
faux	faux	faux

FIGURE 2.1 – Tables de vérité des opérateurs logiques usuels

```

x ← 0, y ← -4, z ← 8,
b1 ← x ≥ y, b2 ← non(x ≥ y) # b1 vaut vrai et b2 vaut faux
b ← (y ≥ z) et (x ≥ y), # b vaut faux
b ← (y ≥ z) ou (x ≥ y), # b vaut vrai

```

2 Algorithmme

Un *algorithme* est une suite d'instructions élémentaires, qui sont effectuées les unes après les autres. Dans notre pseudo-langage, elles sont séparées par des passages à la ligne, ou par des virgules si l'on veut condenser plusieurs instructions sur une même ligne.

Généralement, un algorithme produit un résultat (sortie) à partir d'une ou plusieurs entrées. Dans notre pseudo-langage, on utilisera la construction suivante :

Algorithme *nom* : *entrées* → *sorties* **selon**
corps

.

- Les *entrées* et *sorties*, lorsqu'elles sont plusieurs, elles sont entre parenthèses et séparées par des virgules. Lorsque l'algorithme n'a pas entrée (ou pas de sortie) on utilise les parenthèses vides.
- Le *corps* de l'algorithme sera l'ensemble des instructions à effectuer.
- Les mots **Algorithme** et **selon** sont des mots-clés du pseudo-langage indiquant respectivement le début d'un algorithme, et le début des instructions qu'il contient.
- Le symbole . représente la fin de l'algorithme.

Les entrées contiennent exclusivement des noms de variables, les sorties peuvent contenir des valeurs littérales, ou des variables ; ces dernières pouvant reprendre les variables d'entrées ou être créées dans le corps de l'algorithme.

Pour une question de lisibilité, on ajoute un alinéa aux instructions du corps par rapport au mot-clé **Algorithme** (le corps est décalé vers la droite, on parle aussi d'indentation).

Algorithme somme_et_différence : (x, y) → (s, d) **selon**
 s ← x + y, d ← x - y

.

Algorithme incrémenter : x → x **selon**
 x ← x + 1

.

3 Structure de contrôle

Ce qui précède permet de construire des instructions élémentaires, effectuées les unes après les autres de manière linéaire. Ce type de programme est extrêmement limité. En programmation informatique, une *structure de contrôle* est une instruction particulière qui modifie l'ordre dans lequel les instructions sont effectuées. On dira qu'elle dévie le *flot de contrôle*, ou contrôle du programme, c'est-à-dire l'enchaînement des instructions d'un programme.

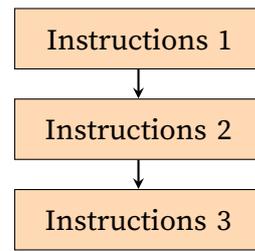


FIGURE 2.2 - Flot de contrôle linéaire

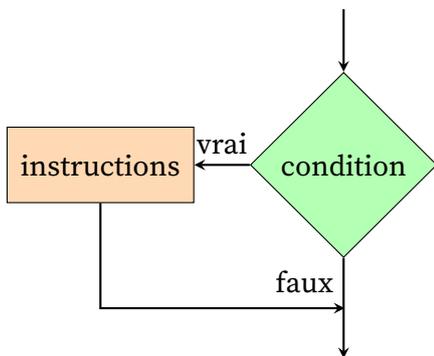
Pour bien mettre en évidence les structures de contrôle, leurs mots-clés commencent par une majuscule et leurs corps se termine par un point (.), comme pour un algorithme. Pour plus de lisibilité, on ira à la ligne dès que le corps comprendra plusieurs instructions; dans ce cas, on indentera (ajouter un alinéa) le corps.

Notons aussi que les structures de contrôle pourront s'imbriquer, le corps d'une structure peut lui-même contenir une autre structure.

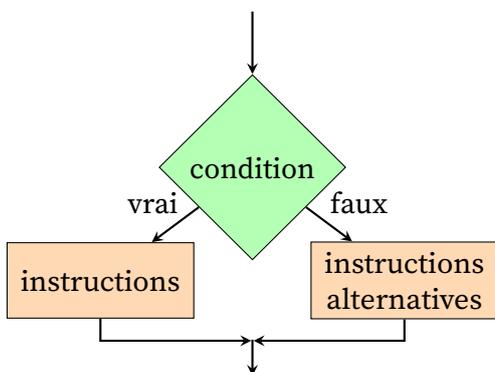
3.1 Branchements conditionnels

Les *branchements conditionnels* permettent de réaliser un ensemble d'instructions seulement si une condition est réalisée. Il suivent une des constructions suivantes, où chaque *condition* est une expression qui s'évalue en booléen, vrai ou faux. Les alternatives avec **si** et **si** sont facultatives.

Si condition alors instructions .



**Si condition alors instructions
sinon instructions alternatives .**



**Si condition 1 alors instructions 1
sinon si condition 2 alors instructions 2
sinon instructions alternatives .**

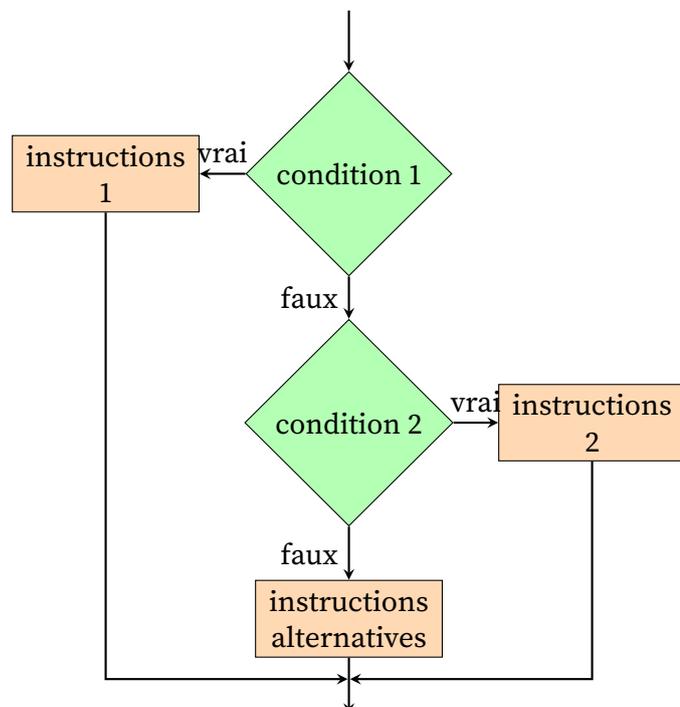


FIGURE 2.3 - Flot d'exécution des trois branchements conditionnels possibles

Remarque. Le branchement conditionnel **Si, sinon si, sinon .** peut être obtenu uniquement avec la construction **Si, sinon .** comme ci-dessous :

```

Si condition 1 alors instructions 1
sinon
  Si condition 2 alors instructions 2
  sinon instructions alternatives .

```

Exemple. Considérons un algorithme racines qui accepte en entrée deux valeurs réelles b et c et calcule en sortie les deux racines du polynôme du second degré $X^2 + bX + c$. Ici, on supposera l'existence dans le pseudo-langage de l'opérateur $\sqrt{\quad}$ qui calcule la racine carrée d'un nombre réel positif ou nul. On rendra la valeur \emptyset lorsqu'il n'y a qu'une ou aucune racine solution.

```

Algorithme racines : (b, c) → (r1, r2) selon
  d ← b × b - 4 × c
  Si d < 0 alors r1 ← ∅, r2 ← ∅
  sinon si d = 0 alors r1 ← -b ÷ 2, r2 ← ∅
  sinon
    r1 ← (-b - √d) ÷ 2
    r2 ← (-b + √d) ÷ 2

```

3.2 Boucles

Les *boucles* servent à répéter des instructions semblables un certain nombre de fois : on appelle cela itérer.

Boucle conditionnelle

Dans la *boucle conditionnelle*, le nombre d'itérations est déterminé par une condition selon la construction suivante :

Tant que condition **répéter** corps .

où *condition* est une expression dont la valeur s'évalue en booléen. La condition est d'abord évaluée, si *condition* vaut vrai, alors *corps* est exécuté, sinon on sort de la boucle; puis cette procédure est répétée.

Attention, le corps doit pouvoir modifier la condition, sinon la boucle sera infinie.

Exemple. L'algorithme multiplication_égyptienne présenté en § 4 du chapitre 1.

```

Algorithme multiplication_égyptienne : (m, n) → p selon
  p ← 0
  Tant que m ≠ 0 répéter
    Si m \ 2 = 0 alors m ← m / 2, n ← n × 2
    sinon m ← m - 1, p ← n + p .

```

Boucle énumérée

Une *boucle énumérée* est une boucle qui a un nombre d'itérations prédéfini. Dans ce type de boucle, un compteur garde en mémoire le numéro d'itération. On utilise la syntaxe suivante :

Pour *compteur* **parcourant** (*premier*, ..., *dernier*) **répéter** *corps* .

strictement équivalente à la boucle conditionnelle suivante :

compteur \leftarrow *premier*

Tant que *compteur* \leq *dernier* **répéter**

corps

compteur \leftarrow *compteur* + 1

.

Notons qu'il y a exactement *premier* – *dernier* + 1 itérations.

Remarque. Ce type de boucle est particulièrement utile pour le parcours des tableaux (cf. § 4)

4 Tableaux

Les *tableaux* sont des collections ordonnées de variables. On peut spécifier directement tous les éléments constitutifs d'un tableau en les regroupant dans des parenthèses comme des *n*-uplets en mathématiques.

On peut aussi créer un tableau avec un nombre d'éléments *n* arbitraire, avec la syntaxe [*n*]. Notons qu'on ne connaît pas la valeur initiale des éléments d'un tel tableau.

t1 \leftarrow (1, 2, 3, 4) # On crée un tableau de 4 entiers

t2 \leftarrow [nombre] # On crée un tableau constitué de nombre éléments

t3 \leftarrow *t1* # *t3* est une copie indépendante de *t1*

On appelle *taille* le nombre d'éléments qui le constituent. En pseudo-langage, on pourra y accéder avec la syntaxe **nélém**(*nom_tab*).

On peut faire référence aux éléments d'un tableau en les indexant avec des parenthèses, suivant la syntaxe *nom_tab*(*indice*), où *indice* est un entier naturel indiquant le numéro de l'élément désiré, donc vérifiant $1 \leq \text{indice} \leq \text{nélém}(\text{nom_tab})$.

Exemple. L'algorithme suivant *créer_initialiser_tableau* crée un tableau de *n* éléments qu'on initialise à la valeur *x*. On renvoie en sortie ce tableau.

Algorithme *créer_initialiser_tableau* : (*n*, *x*) \rightarrow *t* **selon**

t \leftarrow [*n*]

Pour *i* **parcourant** (1, ..., **nélém**(*t*)) **répéter** *t*(*i*) \leftarrow *x* .

.

Exemple. L'algorithme suivant *trouver_élément* accepte en entrée un tableau et un élément, et renvoie en sortie l'indice dans le tableau où apparaît l'élément. Si l'élément n'est pas dans le tableau l'algorithme rendra la valeur -1.

Algorithme *trouver_élément* : (*tab*, *x*) \rightarrow *ind* **selon**

ind \leftarrow -1

Pour *i* **parcourant** (1, ..., **nélém**(*tab*)) **répéter**

Si *tab*(*i*) = *x* **alors** *ind* \leftarrow *i*.

.

5 Appel de routine

Il est fréquent qu'un algorithme soit appelé au sein d'un algorithme plus complexe. Cela permet de mettre en évidence sa structure, de simplifier sa conception et son analyse. Ce mécanisme évite aussi de réécrire les mêmes instructions si on s'en sert plusieurs fois. Nous appellerons ce mécanisme *appel de routine*.

Une fois qu'un algorithme est défini, il peut être appelé comme routine dans un autre algorithme. La construction est alors la suivante :

$$\text{résultat} \leftarrow \text{nom}(\text{arguments})$$

où *résultat* et *arguments* doivent être respectivement compatibles avec les *sorties* et *entrées* de la routine. Lorsqu'il y a plusieurs variables de sortie, on les affecte à des variables regroupées entre parenthèses.

Exemple. Appel de la routine *somme_et_différence*

$(x, y) \leftarrow \text{somme_et_différence}(3, 4)$ # *x* vaut 7, *y* vaut -1

On peut voir la routine comme une fonction mathématique, tout objet qui n'est pas passé en sortie n'est plus accessible par l'algorithme qui appelle la routine, et en particulier les valeurs en entrée ne sont pas modifiées.

Exemple. Appel de la routine *incrémenter*

$y \leftarrow 2, x \leftarrow \text{incrémenter}(y)$ # *x* vaut 3, *y* n'a pas changé

*
* *

3+ La programmation en langage C

1 De l'algorithme à l'implémentation

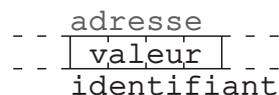
Au chapitre précédent, on a vu comment manipuler des objets de manière abstraite avec le pseudo-langage : opérations, structures de contrôles, etc. À partir de maintenant, l'objectif est de faire réaliser ces tâches par un ordinateur.

Dans nos cours de programmation nous utilisons le langage C, car il est fondamental et permet de mieux comprendre le fonctionnement d'un ordinateur. Développé dans les années 1970¹, il est encore aujourd'hui l'un des langages les plus répandus.² C'est aussi un langage qui permet un contrôle fin des ressources de l'ordinateur, et donc de meilleures performances.

Cela rajoute des contraintes. Il faut s'efforcer de comprendre comment l'ordinateur va interpréter ce que l'on écrit à chaque étape, et comment évolue l'état de la mémoire.

2 Variables en langage C

En programmation, une variable est un moyen de stocker en mémoire une donnée et d'y associer un identifiant. Pour qu'une variable existe, il faut qu'elle soit créée en mémoire pour que l'ordinateur sache où chercher lorsqu'on modifie ou accède à la valeur d'une variable spécifique. En C, chaque variable est associée à une adresse mémoire. Une variable peut être vue comme une boîte de mémoire qu'on aurait associée à un identifiant spécifique.



2.1 Types de bases

Comme vu dans le chapitre 2, l'ordinateur ne manipule que des 0 et des 1, et les différents types de variables ne sont pas représentés de la même manière en mémoire. En C, une variable est associée à un type particulier, qui a des propriétés particulières, notamment l'encodage et la taille en mémoire (nombre d'octets correspondant). En résumé une variable en C aura trois propriétés : un *identifiant* (on dit aussi *identificateur*), une *valeur*, et un *type*. Les types de bases en C sont :

- (a) `char` pour les caractères (1 octet)
- (b) `int` pour les nombres entiers (généralement 4 octets)
- (c) `float` et `double` pour les nombres décimaux à virgule flottante, en précision respectivement simple et double (généralement resp. 4 et 8 octets)

2.2 Déclaration

En C, toute variable que l'on utilise doit être au préalable déclarée. Lorsque l'on fait une *déclaration*, un espace mémoire est réservé. La syntaxe générale d'une déclaration est :

```
type identifiant ;
```

1. Développé par Dennis RITCHIE

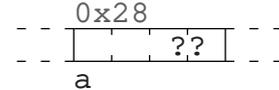
2. À tel point qu'on peut dire qu'il « constitue l'ADN de tous les logiciels modernes ». Jason Perlow, « Without Dennis Ritchie, there would be no Steve Jobs ». *ZDNet.com*, 9 octobre 2015.
<http://www.zdnet.com/article/without-dennis-ritchie-there-would-be-no-jobs/>

Attention, une fois défini, le type d'une variable ne peut plus changer.

Remarque. Dans la syntaxe du langage C, toutes les instructions se terminent par un symbole « ; » ; en particulier, une déclaration est une instruction.

Exemple. Illustration de la déclaration d'un entier a.

```
int a ; /* au moment de la déclaration une
boite mémoire de 4 octets est réservée */
```



Remarque. Dans la syntaxe du langage C, ce qui est placé entre « /* ... */ » n'est pas pris en compte. On s'en sert pour donner des explications sur le programme, c'est ce qu'on appelle des *commentaires*³

2.3 Opérateur d'affectation

Une variable uniquement déclarée est encore non *initialisée* : sa valeur est indéterminée. Il ne faut jamais utiliser la valeur d'une variable non initialisée.

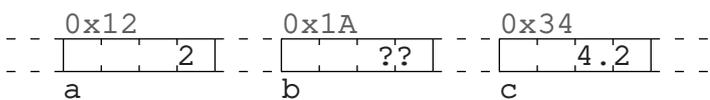
Pour initialiser la variable, on doit utiliser une affectation, de façon analogue au pseudo-langage (cf §1.1). En C, le symbole pour l'affectation est le symbole « = ». Attention à ce symbole, une affectation n'est pas une égalité au sens mathématique : l'expression `a = a + 1;` est correcte en C!

Dans la syntaxe du langage C, on peut:

- affecter une valeur à une variable lors de la déclaration (ligne 1 de l'exemple)
- déclarer plusieurs variables du même type sur la même ligne (ligne 2 de l'exemple)

Exemple. Déclaration et initialisation

```
1 int a = 2;
2 float b, c = 4.2 ;
```



Attention, on ne peut pas déclarer deux variables de types différents sur la même ligne.

```
float pi = 3.14, int a; /* instruction incorrecte */
```

Règles de nommage des variables

- Les identifiants ne peuvent comprendre que des lettres majuscules ou minuscules sans accents, des chiffres, et des tirets bas `_` donc pas d'espaces, pas de caractères spéciaux.
- Le premier caractère doit être une lettre.
- Un identifiant ne peut pas être un mot-clé du langage.

Le langage C est un langage sensible à la casse : majuscules et minuscules sont considérées comme des lettres différentes ; a et A sont les identifiants de deux variables différentes.

2.4 Constantes littérales

Dans le programme on doit pouvoir donner des valeurs spécifiques à nos variables, pour cela on utilise des *constantes littérales*.

- Les entiers sont de type `int` ; par exemple `42` et `-1`.
- Les nombres décimaux et en notation scientifique sont de type `double` ; par exemple `42.1`, `-1.0`, `42.` et `1e-3`.
- Les caractères avec guillemets simples sont de type `char` ; par exemple `'a'`, `'?'` et `'7'`.

3. nous avons déjà écrit des commentaires en pseudo-langage sans l'avoir précisé, en utilisant le symbole « # ».

- Les chaînes de caractères avec des guillemets doubles; par exemple "ceci est une "chaîne de caractères"; nous verrons les détails au chapitre 5.

3 Les entrées et sorties

Pour qu'un programme ait un intérêt, nous avons besoin de lui définir des entrées et des sorties. Il y a différentes manières de le faire : saisie clavier, affichage à l'écran, lecture/écriture sur un fichier, interaction avec d'autres systèmes, etc. Ici on verra l'affichage sur le terminal pour effectuer une sortie, et la saisie au clavier pour effectuer une entrée.

Pour le premier semestre, on utilisera les instructions AFFICHER et SAISIR, mise à disposition dans une bibliothèque interne à l'établissement `insaio.h`. On se libère ainsi des considérations pratiques afin de se concentrer sur les bases de la programmation en C. Pour inclure cette bibliothèque, on ajoutera la ligne suivante au début de notre code source.

```
#include <insaio.h>
```

3.1 AFFICHER

L'instruction AFFICHER effectue l'affichage sur la console. Cette commande peut afficher des variables ou des constantes littérales (uniquement les types de base et les chaînes de caractères) passées en argument entre parenthèses et séparées par des virgules. On peut afficher jusqu'à neuf éléments dans l'ordre spécifié.

```
int a = 2;
float b = 5.3;
AFFICHER(a); /* 2 */
AFFICHER("Hello, World!"); /* Hello, World! */
AFFICHER("a = ", a, ", b = ", b); /* a = 2, b = 5.300000 */
```

Afin de pouvoir mettre en forme les chaînes de caractères, on utilise des caractères spéciaux appelés *séquences d'échappement*. Pour l'instant, seuls deux nous intéressent :

- '`\n`' pour aller à la ligne
- '`\t`' pour une tabulation horizontale

Par exemple, l'instruction suivante affiche sur la console les lignes ci-après.

```
AFFICHER("Ligne 1 : 1.2\t", "2.56", "\n", "Ligne 2\nLigne 3\n");
```

```
Ligne 1 : 1.2      2.56
Ligne 2
Ligne 3
```

3.2 SAISIR

Pour permettre à l'utilisateur du programme de saisir des valeurs dans le terminal, on utilise l'instruction SAISIR en passant une ou plusieurs variables en argument (uniquement les types de base et chaînes de caractères). Lors de l'exécution de cette instruction, le programme se met en pause et attend que l'utilisateur saisisse les valeurs. Une fois celles-ci saisies, les valeurs seront affectées aux variables passées en argument dans l'ordre attendu.

```
int a;
float b;
AFFICHER("Saisir un entier et un nombre décimal :\n");
SAISIR(a, b); /* a et b prendront les valeurs saisies au clavier ;
a première valeur saisie, b deuxième valeur */
```

4 Premiers programmes en C

4.1 La fonction `main()`

Un programme écrit en langage C peut être très complexe et comprendre de nombreux fichiers/fonctions. Il est nécessaire de définir l'ordre de tout ce qui va se passer dans le programme.

C'est le rôle de la fonction `main()`, c'est le point d'entrée de tout programme écrit en C. Seules les instructions qui s'y trouvent seront exécutées, et elles le seront rigoureusement dans l'ordre du `main`.

Tout programme écrit en C doit donc contenir une et une seule fonction `main()`. Voici la syntaxe générique de cette fonction :

```
int main()
{
    instructions
    return 0;
}
```

L'instruction `return` dans `main()` termine l'exécution du programme. La valeur retournée par `main()` est particulière : il s'agit d'un code que certains systèmes d'exploitation peuvent consulter lorsque le programme a cessé de fonctionner. La valeur `0` indique que l'exécution s'est bien déroulée, toute autre valeur indique une exécution anormale.

Remarque. Pour la lisibilité : indentation (décalage) vers la droite lorsqu'on entre dans des accolades.

Exemple. Code source d'un programme qui effectue la saisie et l'affichage de deux entiers.

```
#include <insaio.h>
int main()
{
    int a, b;
    AFFICHER("Saisir deux valeurs entières");
    SAISIR(a, b);
    AFFICHER("La variable a vaut :", a, "La variable b vaut", b);
    return 0;
}
```

4.2 La compilation

Le microprocesseur ne peut manipuler que des fichiers binaires. Notre code source (fichier texte) doit être transformé en langage machine (cf le [chapitre 1](#)). C'est l'étape de *compilation* qui se fait dans le terminal, selon la syntaxe suivante (sur Windows) :

```
C:\Users\cb\monDossier> gcc fichier_source.c -o nom_executable.exe
```

où `fichier_source.c` est le fichier contenant votre code source en langage C, et le nom du fichier exécutable est `nom_executable.exe`. Pour exécuter le programme, c'est à dire, mettre effectivement en œuvre les instructions du programme par l'ordinateur, on devra lancer la commande d'exécution suivante (sur Windows) :

```
C:\Users\cb\monDossier> nom_executable
```

La navigation et la compilation en ligne de commande sont détaillées dans l'[annexe B](#).

5 Opérateurs et expressions

5.1 Opérateurs arithmétiques

Dans le langage C, on retrouve les opérateurs arithmétiques binaires classiques : addition $+$, soustraction $-$, multiplication $*$ et division $/$.

Attention, dans le langage C, la division n'est pas la même pour les types entiers (`char`, `int`) et pour les types décimaux à virgule flottante (`float`, `double`). La division entre deux entiers sera la division euclidienne⁴ et celle entre deux décimaux sera une division réelle approchée.

L'opérateur `%` calcule le reste de la division entière entre deux entiers.

Il est important de noter que si les opérandes sont de types différents, une *conversion de type implicite* aura lieu. Les règles de conversions peuvent être compliquées et il n'est pas nécessaire de rentrer dans les détails : retenons simplement que la conversion choisie est généralement celle qui *perd le moins d'information*. Le cas le plus important à souligner est quand un type entier est combiné avec un type flottant ; l'entier sera temporairement converti vers le flottant le plus proche avant de faire l'opération.

Notons aussi que le même genre de conversion peut s'appliquer quand on affecte une valeur d'un certain type à une variable de type différent.

5.2 Règles de priorité et d'associativité

Lorsque qu'une expression est constituée de plusieurs opérateurs binaires, l'expression peut être interprétée de plusieurs manières. Les règles de priorité et d'associativité déterminent la sémantique d'une expression composée.

```
int a = 2;
float pi = 3.12;
/* Interprétations possibles (a*2) + 1 ou a*(2 + 1)
règle : * prioritaire devant + */
a * 2 + 1; /* équivalent à (a*2) + 1 */
/* expression ambiguë : * et / ont la même priorité.
règle : * et / sont associatifs de gauche à droite */
a / 2.0 * pi; /* équivalent à (a/2.0)*pi */
```

L'ensemble des règles de priorité et d'associativité des opérateurs du langage C peuvent être consulté sur Internet⁵. Il n'est pas nécessaire d'apprendre toutes les règles, il vaut mieux utiliser des parenthèses pour retirer toute ambiguïté d'une expression.

Les parenthèses permettent aussi de modifier la sémantique d'une expression, par exemple $(x + 2) * (y - 1)$; au lieu de $x + 2 * y - 1$;.

Attention, les règles de priorité et d'associativité ne fournissent aucune indication quant à l'ordre d'évaluation des opérandes. Par exemple, dans l'expression $(x + 2) * (y - 1)$ on ne sait pas laquelle des sous-expressions $x + 2$ ou $y - 1$ est évaluée en premier.

5.3 Expressions logiques

Le langage C ne possède pas de type booléen, les valeurs de vérité sont représentées par des entiers. La valeur `0` est utilisée pour représenter faux et toute valeur différente de `0` représente vrai.

Les *expressions logiques*⁵ permettent de vérifier des propriétés. Elle peuvent être construites à l'aide des *opérateurs de comparaisons* (relationnels et d'égalité) qui produisent la valeur entière

4. pour les entiers positifs; pour les entiers relatifs, la définition est légèrement différente

5. https://en.cppreference.com/w/c/language/operator_precedence

0 si l'expression est fausse et 1 si l'expression est vraie. Ils peuvent être utilisés pour comparer des entiers (`char` ou `int`) et des décimaux (`float` ou `double`) :

- strictement inférieur (resp. supérieur) à `<` (resp. `>`)
- inférieur (resp. supérieur) ou égal à `<=` (resp. `>=`)
- égal à `==`
- différent de `!=`

Remarque. Notons la différence entre l'opérateur `==` (test d'égalité) et l'opérateur `=` (affectation). Attention, dans notre pseudo-langage le test d'égalité utilise le symbole `=`.

Règle de priorité et d'associativité

Les opérateurs relationnels et d'égalités ont aussi leurs règles de priorités et d'associativités. Elles ne sont pas à connaître par cœur, il faut savoir évaluer et écrire une expression logique, soit à partir de parenthèses, soit lorsque les règles sont rappelées.

- (a) Les opérateurs logiques ont une priorité inférieure à celle des opérateurs arithmétiques.
- (b) Les opérateurs logiques sont associatifs à gauche.
- (c) Les opérateurs d'égalité ont une priorité plus faible que celle des opérateurs relationnels.

```
1 i + j < k - 1; /* équiv. à (i + j) < (k - 1) selon (a) */
2 i < j < k; /* équiv. à (i < j) < k selon la règle d'associativité
3 à gauche ; attention, ne signifie pas j compris entre i et k */
4 i < j == j < k; /* équiv. à (i < j) == (j < k) selon (c) */
```

Remarque. Notons que la [ligne 2](#) revient à comparer une valeur entière à une valeur de vérité ce qui n'a pas de sens. À éviter même si le langage le permet.

5.4 Opérateurs logiques

Les opérateurs logiques permettent de construire des expressions logiques complexes à partir d'expressions élémentaires. Les tables de vérité sont les mêmes que celles présentées pour le pseudo-langage, § 1.2 du [chapitre 2](#). Les opérateurs logiques produisent, eux aussi, 0 (faux) ou 1 (vrai).

- négation logique (non) `!`
- disjonction logique (ou) `||`
- conjonction logique (et) `&&`

Remarque. Souvent, les opérandes des opérateurs logiques sont des expressions logiques. Cependant ceci n'est pas une obligation. Dans tous les cas, un opérande différent de 0 est considérée comme vrai et un opérande égal à 0 est considéré comme faux.

```
int a = 2, b = 0;
!a /* faux */
a || b /* vrai */
a && (b + 3) /* vrai */
```

Remarque. L'évaluation d'une expression faisant intervenir les opérateurs `&&` ou `||` est court-circuitée : l'opérande de gauche est évalué en premier.

```
int i, j;
(i != 0) && (j/i > 0); /* pas d'erreur car l'expression j/i > 0
n'est évaluée que lorsque i != 0 */
```

6 Structures de contrôle en C

En C, comme en pseudo-langage, une structure de contrôle est un élément qui va modifier le flot des instructions. Les structures de contrôle sont associées à un corps, c'est-à-dire un ensemble d'instructions. Ce corps est défini par un bloc entre accolades. Un corps sera associé à la structure de contrôle qui le précède directement.

```
structure_de_contrôle{
    instructions
}
```

Notons que les structures de contrôles peuvent être imbriquées. Pour une question de lisibilité, les instructions d'un sous-bloc (dès qu'on rencontre des accolades) doivent être indentées (ajout d'alinéa) par rapport à celles du bloc supérieur, et l'accolade fermante du bloc doit être alignée avec le début de la structure de contrôle.

6.1 Branchements conditionnels

Les branchements conditionnels fonctionnent en C exactement comme dans notre pseudo-langage, présentés en § 3.1 du chapitre 2. Les mots clés correspondant à **Si**, **sinon si**, **sinon** sont leurs équivalents anglais respectivement **if**, **else if**, **else**.

```
if (condition 1) {
    corps 1
}
else if (condition 2) {
    corps 2
}
else {
    corps 3
}
```

Ici, si la valeur de *condition 1* est différente de 0, le *corps 1* est exécuté; sinon, on passe à la suite. Notons bien que l'expression à l'intérieur des parenthèses n'est pas nécessairement une expression logique; *condition* peut être n'importe quel type d'expression, la condition sera vérifiée dès lors que la valeur de l'expression n'est pas nulle.

```
if (a) { /* équivalent à if (a != 0) */
    AFFICHER("a n'est pas nul");
}
```

Remarque. Si on confond l'opérateur égalité `==` avec l'affectation `=`, le programme n'aura pas le comportement attendu. Pour comprendre ce qu'il se passe, il faut signaler qu'en C, toute expression possède une valeur. En particulier, une affectation possède à la fois un effet (modifier la valeur d'une variable) mais aussi une valeur; dans le cas de l'affectation, la valeur de l'opérande de droite. Ainsi dans l'exemple suivant :

```
if (i = 50) { /* à la place de i == 50 */
    AFFICHER(i, " = 50");
}
```

Quelque soit la valeur de *i*, la valeur de l'expression d'affectation `i = 50` sera 50. Ainsi la condition sera considéré comme vraie pour tout *i*, contrairement à ce que l'on cherchait.

6.2 Boucle

Rappel : une boucle est une structure de contrôle qui exécute de manière répétée un ensemble d'instructions, appelé corps de la boucle. Chaque exécution du corps de la boucle

s'appelle une *itération*.

En C, toutes les boucles ont des expressions de contrôle : à chaque itération (au début ou à la fin), cette expression est évaluée; si le résultat est différent de zéro, le corps de la boucle est exécuté; sinon on sort de la boucle.

Boucle `while` : boucle conditionnelle

C'est la boucle la plus simple et la plus essentielle en C, elle s'écrit selon la syntaxe ci-dessous. Au début de chaque itération de la boucle `while`, *condition* est évaluée; si le résultat est différent de 0, le corps de la boucle est exécuté; sinon on sort de la boucle.

```
while (condition) {
    instructions
}
```

Exemple. Les instructions suivantes calculent la plus petite puissance de 2 supérieure ou égale à un nombre *n*. Voici une trace d'exécution selon l'itération.

	ligne	i	itération	commentaire
	2	1		
1	<code>int n = 7;</code>	3	1	<code>i < n</code> vaut 1, on rentre dans la boucle
2	<code>int i = 1;</code>	4	1	<code>i</code> change de valeur
3	<code>while (i < n) {</code>	3	2	<code>i < n</code> vaut 1, on rentre dans la boucle
4	<code> i = i*2;</code>	4	2	<code>i</code> change de valeur
5	<code>}</code>	3	3	<code>i < n</code> vaut 1, on rentre dans la boucle
		4	3	<code>i</code> change de valeur
		3	4	<code>i < n</code> vaut 0, on sort de la boucle

Boucle `do while`

La boucle `do while` est très similaire à la boucle `while`. Dans une boucle `do while`, l'expression de contrôle est testée à la fin de chaque exécution du corps de la boucle. En particulier, les instructions du corps d'une boucle `do while` sont exécutées au moins une fois.

```
do {
    instructions
} while (condition); /* attention au point-virgule à la fin */
```

Exemple. Les instructions suivantes effectuent un compte à rebours en utilisant une boucle `do while`.

	ligne	i	itération	commentaire
	1	3		
1	<code>int i = 3;</code>	3	2	<code>i</code> change de valeur
2	<code>do {</code>	4	2	<code>i</code> vaut 2, on retourne dans la boucle
3	<code> i = i - 1;</code>	3	1	<code>i</code> change de valeur
4	<code>} while (i);</code>	4	1	<code>i</code> vaut 1, on retourne dans la boucle
		3	0	<code>i</code> change de valeur
		4	0	<code>i</code> vaut 0, on sort de la boucle

Boucle `for` : boucle énumérée

La boucle `for` permet de réaliser toutes sortes de boucles notamment les boucles énumérées. Sa syntaxe est la suivante :

```
for (initialisation; test; incrément) {
    instructions
}
```

- *initialisation* : expression d'initialisation, elle est exécutée une seule fois au début.
- *test* : expression de contrôle, elle est exécutée avant chaque itération. On procède à l'itération suivante si et seulement si l'expression *test* est non nulle.
- *incrément* : expression exécutée à la fin de chaque itération, généralement pour modifier l'état d'une variable figurant dans l'expression de contrôle.

La boucle `for` est notamment utilisée pour réaliser des boucles énumérées, dont on connaît le nombre d'itérations au préalable. Elle prend généralement l'une des deux formes suivantes :

```
for (i = min; i < max; i++) {    for (i = max - 1; i >= min; i--) {
    instructions                    instructions
}                                }
```

Dans ces exemples, le nombre d'itérations est $\max - \min$, et le compteur *i* parcourt les valeurs entre *min* inclus et *max* exclu en croissant (version de gauche) ou en décroissant (version de droite).

Remarque. L'expression `i++` (respectivement `i--`) incrémente (respectivement décrémente) la variable *i* de 1, elle remplace `i = i + 1` (respectivement `i = i - 1`).⁶

Exemple. Trace d'exécution d'une boucle `for`

```
1 int i;
2 for (i = 3; i > 1; i--) {
3     AFFICHER("i = ", i, "\n");
4 }
```

ligne	i	itération	commentaire
1	?		
2	3	1	<code>i > 1</code> vaut 1, on rentre dans la boucle
3	3	1	affichage « i = 3 »
4	2	1	fin de l'itération, <i>i</i> change de valeur
2	2	2	<code>i > 1</code> vaut 1, on rentre dans la boucle
3	2	2	affichage « i = 2 »
4	1	2	fin de l'itération, <i>i</i> change de valeur
2	1	3	<code>i > 1</code> vaut 0, on sort de la boucle

À de rares exceptions près, une boucle `for` peut être transformée en boucle `while` :

```
for (expr1; expr2; expr3) {    expr1;
    instructions                while(expr2) {
}                                instructions
                                expr3;
                                }
```

6. l'expression `i++` a le même effet que `i = i + 1` mais n'a pas la même valeur. L'expression `i++` vaut *i* alors que l'expression `i = i + 1` vaut *i + 1*

Remarque. Attention dans les structures de contrôles, ne pas terminer la condition entre parenthèses par le symbole « ; ». Une bonne pratique est de mettre une accolade ouvrante « { » immédiatement après la parenthèse fermante «) ». En effet, considérons une modification de l'exemple précédent comme suit :

```

1 int i;
2 for (i = 3; i > 1; i--);
3 {
4     AFFICHER("i = ", i, "\n");
5 }

```

La ligne 2 sera interprétée comme `for (i = 3; i > 1; i--) {};`. L'instruction vide « ; » sera considérée comme le corps de la structure de contrôle `for`; et le bloc d'affichage des lignes 3–5, n'étant pas associé à la structure de contrôle, ne sera exécuté qu'une seule fois après la boucle.

7 Les tableaux statiques

Un tableau est une structure de données constituée d'un ensemble d'éléments de même type, chacun étant identifié par un indice. Le tableau peut être vu comme un espace mémoire contigu découpé en compartiments. On peut accéder directement aux compartiments à partir de leur indice.

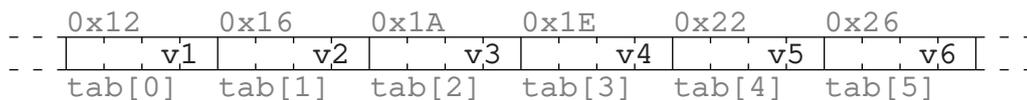


FIGURE 3.1 – Illustration d'une case mémoire d'un tableau `tab`, de taille 6, de valeurs v_i où i varie de 1 à 6

Un tableau est une variable : il possède un identifiant, un type et une valeur. Le type sera celui des éléments contenus dans les cases. Comme vu au chapitre 2, § 4, on appelle *taille* le nombre de compartiments d'un tableau.

Les tableaux présentés ici sont des tableaux dits *statiques* : leur taille doit être connue lors de la compilation et ne peut être changer lors de l'exécution du programme. Voici deux syntaxes pour déclarer un tableau statique :

```

/* Déclaration d'un tableau de 10 cases.
   Les valeurs des cases ne sont pas initialisées. */
type tab[10];
/* Déclaration et initialisation d'un tableau de 10 cases.
   Le compilateur déduit la taille du tableau. */
type tab[] = {v1, v2, v3, v4, v5, v6, v7, v8, v9, v10};

```

Pour accéder à l'élément d'indice i d'un tableau `tab`, on utilise l'opérateur d'indexation « [] », selon la syntaxe suivante :

```
tab[i]
```

À noter, en C, l'indexation des tableaux débute à 0; c'est-à-dire que l'indice de la première case est 0, l'indice de la seconde case est 1, et plus généralement l'indice de la case numéro i est $i - 1$.

Exemple. Déclaration de deux tableaux `tab1` et `tab2` de taille trois. Trois cases sont créées pour chaque tableau (d'indices 0, 1, 2); attention il n'y a pas de case d'indice 3.

```

1 /* Déclaration d'un tableau de 3 entiers */
2 int tab1[3];
3 /* Déclaration et initialisation d'un tableau de 3 flottants */
4 float tab2[] = {1.2, 2.0, 3.4};

```

0x12	0x16	0x1A
??	??	??
tab1[0]	tab1[1]	tab1[2]
0x40	0x44	0x48
1.2	2.0	3.4
tab2[0]	tab2[1]	tab2[2]

Remarque. Contrairement à ce qu'on se permet de faire en pseudo-langage, on ne peut pas utiliser l'opérateur d'affectation «=» pour copier le contenu d'un tableau dans un autre. Il faudra parcourir le tableau et copier terme à terme les éléments.

Les constantes symboliques et macro-définitions : #define

Les instructions suivantes ne sont pas valables en C.

```
int n = 5;
int tab[n];
```

En effet, à la compilation la variable `n` n'existe pas; le compilateur ne peut donc pas connaître la taille du tableau. Pour définir une taille de tableau dont la valeur peut varier d'une compilation à l'autre, de manière cohérente dans tout le code source et en ne modifiant qu'une seule ligne, on utilise les *constantes symboliques*. La syntaxe est la suivante :

```
#define NOM valeur
```

Cette syntaxe permet de substituer toutes les occurrences de `NOM` par `valeur` dans le code source. L'opération de substitution a lieu avant l'étape de compilation. Attention, une constante symbolique n'est pas une variable. Pour distinguer les deux dans un code source, on écrit par convention les identifiants des variables en minuscules et les constantes symboliques, plus généralement les macro-définitions, en majuscules.

Exemple.

```

1 #define TAILLE 4
2 #define PI 3.14159265359
3
4 int main()
5 {
6     double rayons[TAILLE]; /* Déclaration d'un tableau de 4 double */
7     double aires[TAILLE]; /* Déclaration d'un tableau de 4 double */
8     int i;
9     for(i = 0; i < TAILLE; i++){ /* i varie de 0 à 3 */
10        rayons[i] = 4*i + 12;
11        aires[i] = PI*rayons[i]*rayons[i];
12    }
13    return 0;
14 }

```

Notons qu'on peut utiliser les constantes symboliques pour d'autres choses que des tailles de tableau, par exemple des paramètres physiques d'un problème. De plus, ce mécanisme n'est qu'un cas particulier d'un mécanisme plus général de *macro-définition*.

7.1 Parcours d'un tableau

Les boucles énumérées sont particulièrement adaptées à l'utilisation des tableaux. En particulier, lorsqu'on voudra parcourir toutes les cases ou une partie contiguë d'un tableau, on privilégiera la boucle `for`.

Exemple. Programme de saisie et d'affichage d'un tableau statique.

```

#include <insaio.h>
#define TAILLE 5

int main ()
{
    int i;
    int tab[TAILLE];
    /* Saisie du tableau */
    for(i = 0; i < TAILLE; i++){
        AFFICHER("Saisir la valeur d'indice ", i);
        SAISIR(tab[i]);
    }
    /* Affichage du tableau */
    AFFICHER("[");
    for(i = 0; i < TAILLE - 1; i++){
        AFFICHER(tab[i], ", ");
    }
    AFFICHER(tab[i], "]\n");
    return 0;
}

```

7.2 Tableaux multidimensionnels

```
int t[4][3];
```

t[0][0]	t[0][1]	t[0][2]
t[1][0]	t[1][1]	t[1][2]
t[2][0]	t[2][1]	t[2][2]
t[3][0]	t[3][1]	t[3][2]

FIGURE 3.2 – Déclaration et représentation d'un tableau bidimensionnel d'identifiant t, de taille 4 lignes et 3 colonnes

On peut considérer le tableau t comme ayant 4 lignes et 3 colonnes. Selon cette convention, on accèdera à la case de la ligne d'indice i et de la colonne d'indice j avec l'appel suivant : t[i][j].

En réalité, un tableau bidimensionnel n'est pas stocké en mémoire comme une matrice, mais comme un tableau unidimensionnel résultant de la concaténation des lignes de la matrice correspondante, comme illustré sur la figure 3.3.

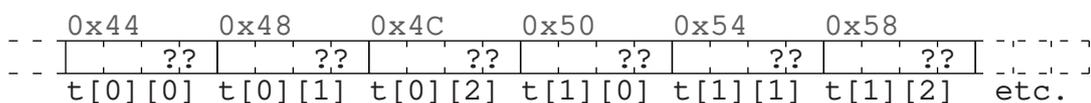


FIGURE 3.3 – Représentation en mémoire du tableau bidimensionnel t

Exemple. Saisie d'un tableau bidimensionnel de 5 lignes et 4 colonnes. Notez que pour parcourir toutes les cases, on doit utiliser des boucles imbriquées pour parcourir chaque colonne au sein de chaque ligne du tableau

```
#include <insaio.h>
#define N_LIGNES 5
#define N_COL 4

int main ()
{
    int i;
    int tab[N_LIGNES][N_COL];
    for(i = 0; i < N_LIGNES; i++){ /* Parcours des lignes */
        for(j = 0; j < N_COL; j++){ /* Parcours des colonnes */
            AFFICHER("Saisir la valeur tab[", i, "][", j, " ");
            SAISIR(tab[i][j]);
        }
    }
    return 0;
}
```

*
* *

4+ Les fonctions en langage C

1 Généralités

En langage C, le mécanisme d'appel de routine, présenté en § 5 du chapitre 2, est mis en œuvre par les *fonctions*.

- Une fonction est un ensemble d'instructions possédant un identifiant.
- Les données initiales de la fonctions sont appelées ses *paramètres*.
- Le résultat calculé par la fonction est appelé *valeur de retour*.

1.1 Définition d'une fonction

Syntaxe

```
1 type_retour id_fonction(liste_parametres)
2 {
3     instructions
4     return valeur_retour;
5 }
```

- La *ligne 1* s'appelle l'*en-tête* ou le *prototype* de la fonction.
- Le *corps* est l'ensemble des instructions à l'intérieur des accolades (les *lignes 2-5*).

La liste des paramètres

Les paramètres d'entrée sont les variables initiales manipulées par la fonction. Pour les manipuler, la fonction a besoin de savoir quel est le type de chacune de ces variables. Ainsi, dans la liste des paramètres chaque paramètre est précédé de son type :

```
type_retour id_fonction(type_1 param_1, type_2 param_2, ...)
```

Les identifiants de fonction et de paramètres doivent mettre en évidence leur rôle pour une question de clarté. Les règles de nommage des fonctions sont identiques à celles des variables, voir § 2.3 du chapitre 3.

Une fonction peut aussi avoir une liste de paramètres vide.

Corps d'une fonction

Le corps d'une fonction est la mise en œuvre d'un algorithme permettant de réaliser les opérations voulues. On peut y déclarer de nouvelles variables, utiliser tout type d'instructions du langage C, y compris l'appel à d'autres fonctions.

Valeur de retour : mot-clé `return`

Le mot-clé `return` permet de produire une valeur en sortie de la fonction. Le type de cette valeur doit être compatible avec le `type_retour` précisé dans l'en-tête de la fonction.

Lorsque le programme rencontre le mot-clé `return`, l'exécution de la fonction est immédiatement stoppée ; et le flot d'exécution reprend au niveau de l'instruction qui a appelé la fonction. S'il s'agit du `return` de la fonction `main()`, l'exécution du programme est terminée.

Lorsqu'une fonction ne produit pas de valeur résultat, on indiquera `void` pour le type de retour. Une fonction `void` peut utiliser le mot-clé `return`. Dans ce cas, aucune expression

ne devra être mentionnée après `return`. Une telle fonction peut néanmoins utiliser le mot-clé `return` seul, sans expression de retour spécifiée, pour sortir de la fonction.

Au contraire, `return` est indispensable pour spécifier une sortie non `void`. En particulier, il est important de vérifier que `return` est présent dans tous les branchements d'une telle fonction.

Exemple.

```

1  /* Calcule la moyenne de deux nombres décimaux */
2
3  float moyenne(float x, float y)
4  {
5      return (x + y)/2.0; /* ici le retour est une expression */
6  }
7
8  /* Vérifie si un nombre est compris entre deux autres */
9
10 int est_dans_intervalle(int x, int min, int max)
11 {
12     if(min < x && x < max){
13         return 1; /* si min < x < max on sort de la fonction ici */
14     }
15     return 0; /* équivalent à else { return 0; } */
16 }
```

2 Appel de fonction

Syntaxe

```
id_fonction(argument_1, argument_2, ...);
```

Contrôle du programme

Lorsqu'une fonction est appelée, selon la syntaxe ci-dessus, le *contrôle du programme* est transféré vers cette fonction. Les éventuels paramètres et variables locales sont créés et les diverses instructions exécutées.

La fonction s'arrête si toutes les instructions de la fonction ont été exécutées (quand on rencontre la dernière accolade), ou si un `return` est rencontré. À ce moment le contrôle est rendu à la *fonction appelante*.

Arguments

Lorsqu'on appelle une fonction, on doit donner une valeur initiale à chacun de ses paramètres. Ces valeurs peuvent être une valeur littérale, la valeur d'une variable, ou une expression. Ces valeurs sont appelées les *arguments*.

Lorsque la fonction est exécutée, les valeurs initiales des paramètres prennent les valeurs des arguments, comme si on avait écrit les instructions `type_1 param_1 = argument_1;`, `type_2 param_2 = argument_2;`, etc.

La liste des arguments donnée lors de l'appel d'une fonction doit être compatible, en nombre et en type,¹ avec la liste des paramètres de l'en-tête de la fonction.

1. une conversion peut avoir lieu si un argument n'est pas du même type que le paramètre correspondant

Exploitation du résultat

Comme toute instruction C, l'appel à une fonction est une expression qui possède une valeur : la valeur de retour. On peut exploiter ce résultat de différentes manières : en l'affectant à une variable, en l'affichant, en l'utilisant dans un test, etc.

Les fonctions qui ne produisent pas de résultats, comme les fonctions d'affichage, sont utilisées comme des instructions et non des expressions. C'est l'effet produit par l'appel qui nous intéresse et non pas la valeur de retour.

Remarque. Une fonction ne peut appeler une autre que si celle-ci a été définie avant dans le code source.²

Exemple. Programme utilisant les fonctions `moyenne()` et `est_dans_intervalle()` définies précédemment

```
#include <insaio.h>

...
/* Définitions des fonctions moyenne et est_dans_intervalle */

int main()
{
    int a = 4, b = 5, c = 8;
    float x = 4.4, y = 2.0, z;
    AFFICHER(moyenne(x, y), "\n"); /* affiche 3.200000 */
    z = moyenne(0.0, y + 2.0) /* z vaut 2.0 */
    if (est_dans_intervalle(c, a, b)){
        AFFICHER(c, " est compris entre ", a, " et ", b, "\n");
    }
    return 0;
}
```

2.1 Passage par valeur

En C, le passage d'arguments se fait toujours par valeur : lors de l'appel d'une fonction, les paramètres sont définis et initialisés avec la valeur des arguments associés. Dit autrement, les paramètres sont des copies indépendantes des arguments.

Les arguments ne sont donc jamais modifiés lors d'un appel de fonction même si les paramètres sont modifiés au sein de la fonction.

Porté d'une variable

En C, une variable n'existe que dans le *bloc* de code où elle a été déclarée, c'est-à-dire le bloc défini par la dernière accolade ouvrante « { » avant cette déclaration et son accolade fermante « } » correspondante. La zone de visibilité d'une variable est appelée la *portée*.

Notons qu'un bloc peut contenir des sous-blocs et qu'ainsi une variable déclarée à l'intérieur d'un sous-bloc n'est visible qu'au sein de ce sous-bloc. Cependant une variable déclarée dans un bloc est accessible dans tout sous-bloc imbriqué.

2. En fait, il suffit de déclarer son prototype, l'éditeur de lien saura retrouver la définitions, mais ces considérations dépassent le cadre de ce cours.

Exemple.

```

{
    int a = 1;

    if (a == 1) {
        double x = a * a ;
        /* Dans ce sous-bloc, a est visible */
    }

    x = x + 1;
    /* Erreur ! Dans ce bloc x n'existe pas */
}

```

Un appel de fonction introduit une nouvelle portée, au début de laquelle les seules variables existantes sont les paramètres de la fonction.

En particulier, les variables visibles pour la fonction appelante ne sont pas visibles pour la fonction appelée. Il faut donc que l'algorithme implémenté par la fonction appelée soit capable de réaliser les opérations voulues uniquement avec ses paramètres d'entrée.

De même, à la fin d'un appel de fonction, les paramètres ainsi que les variables locales de la fonction sont détruits, ils ne seront pas visibles au sein de la fonction appelante.

Exemple. Dans le programme suivant, l'appel de la fonction `echanger ()` n'a aucun effet sur les variables `a` et `b` de la fonction `main ()` ; la [figure 4.1](#) schématise les états successifs de la mémoire.

```

1 #include <insaio.h>
2
3 void echanger(int x, int y)
4 {
5     int tmp = x;
6     x = y;
7     y = tmp;
8 }
9
10 int main()
11 {
12     int a = 1, b = 2;
13     echanger(a, b);
14     AFFICHER("a = ", a, ", b = ", b);
15     /* affiche a = 1, b = 2 */
16     return 0;
17 }

```

3 Les tableaux comme paramètres de fonction

Syntaxe de l'en-tête

Il est aussi possible de passer un tableau en paramètre d'une fonction. La syntaxe de l'en-tête consiste à ajouter des crochets vides dans les paramètres de la fonction pour lui spécifier qu'il s'agit d'un tableau.

```

type_retour id_fonction(type id_tableau[], ...)

```

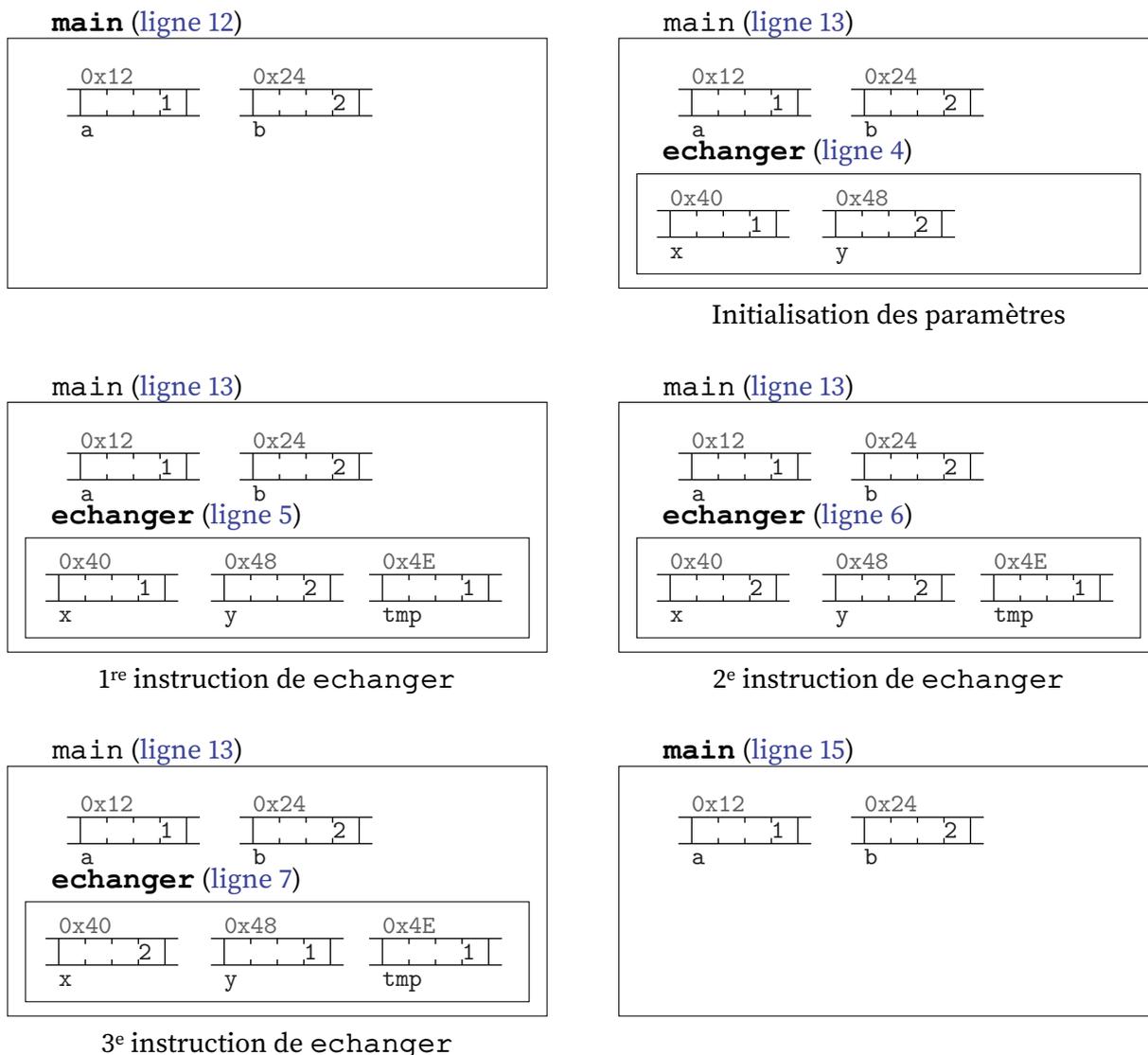


FIGURE 4.1 – Schéma de l'état de la mémoire du programme précédent, les portés fonctions sont représentées par des rectangles. À chaque étape, la fonction ayant le contrôle du programme est indiquée en gras, et la ligne du code source correspondante est indiquée entre parenthèses.

Pour un tableau bidimensionnel, il est obligatoire d'indiquer la taille de la deuxième dimension.

```
type_retour id_fonction(type id_tableau2D[][n_colonnes], ...)
```

Syntaxe de l'appel

Au moment de l'appel de fonction, lorsque l'on passe un tableau en argument, on précise simplement son identifiant (sans les crochets).

```
id_fonction(id_tableau, ...);
```

Remarque. Passer un tableau en paramètre ne donne aucune information à la fonction sur la taille du tableau. Si celle-ci n'est pas connue par ailleurs (par exemple avec une macro-définition), il est nécessaire de la préciser à l'aide d'un deuxième paramètre.

Exemple. Programme définissant et appelant des fonctions pour l'affichage de tableaux.

```
#include <insaio.h>
#define NB_LIGNES 2
#define NB_COLONNES 4
void afficher_tableau(float tab[], int taille)
{
    int i;
    AFFICHER("[");
    for(i = 0; i < taille - 1; i++){
        AFFICHER(tab[i], ", ");
    }
    AFFICHER(tab[i], "]\n");
}
/* La taille de tab2D est donnée par les constantes symboliques */
void afficher_tableau2D(int tab2D[][NB_COLONNES])
{
    int i, j;
    for(i = 0; i < NB_LIGNES; i++){
        AFFICHER("[");
        for(j = 0; j < NB_COLONNES - 1; j++){
            AFFICHER(tab2D[i][j], "\t ");
        }
        AFFICHER(tab2D[i][j], "]\n");
    }
}
int main()
{
    float tab[] = {3.2, 2.3, 1.0, 2.0, 0.0};
    int tab2D[][NB_COLONNES] = { {1, 2, 3, 4}, {5, 6, 7, 8} };
    afficher_tableau(tab, 5);
    afficher_tableau2D(tab2D);
    return 0;
}
```

Particularités des tableaux

Nous avons écrit plus haut § 2.1 qu'une fonction ne modifier ses arguments. Cependant, les tableaux sont un cas particulier important : une fonction peut modifier les éléments d'un tableau passé en argument. Nous reviendrons en détails sur ce mécanisme au second semestre.

Exemple. Programme qui définit et appelle une fonction qui remplace la valeur dans la dernière case d'un tableau d'entier par 10.

```
void modifier_tableau(int tab[], int n)
{
    tab[n - 1] = 10;
}

int main()
{
    int t[] = {0, 1, 2, 3, 4};
    modifier_tableau(t, 5); /* t vaut {0, 1, 2, 3, 10}*/
    afficher_tableau(t, 5);
    return 0;
}
```

4 Méthodologie pour écrire une fonction

- (a) Réfléchir à l'algorithme permettant d'obtenir le résultat voulu.
- (b) Identifier les données initiales nécessaire pour effectuer cet algorithme.
- (c) Déterminer si elle va produire un résultat : type de la valeur retournée.
- (d) Rédiger l'en-tête de la fonction : choix de l'identifiant de la fonction, de quels paramètres on a besoin, quel est leur type.
- (e) Implémenter cet algorithme : le corps de la fonction.

*
* *

5+ Chaînes de caractères en langage C

1 Le type `char`

Le type `char` est un type entier encodé sur un seul multipllet¹. On utilise ce type pour représenter les caractères. La conversion entre les entiers et les caractères est établie par une table de correspondance.

L'une des tables les plus répandues est la table ASCII. En pratique, la table utilisée varie d'un ordinateur à l'autre, ou d'un compilateur à l'autre; on n'évite donc de faire des hypothèses sur la table de correspondance en écrivant un programme.

Pour affecter des caractères spécifiques, on utilise les caractères littéraux entre guillemets simples. Certains caractères spéciaux utilisent des séquences d'échappement.

```
char c = 'a'; /* le caractère a */
c = '7'; /* le caractère 7 et non pas l'entier 7 */
char c = ' '; /* le caractère espace */
c = '\n'; /* le caractère nouvelle ligne */
```

Le fait que les `char` soient représentés par des entiers a plusieurs implications :

- On peut affecter une valeur entière à une variable de type `char`.
`char ch = 65; /* dans la table ASCII, correspond à ch = 'A' */`
- On peut utiliser les opérations arithmétiques ou de comparaison sur les caractères comme sur les entiers. En particulier si les majuscules et les minuscules sont consécutives et dans l'ordre alphabétique dans la table de correspondance², on peut faire ce qui suit.
`ch = ch + 1; /* ch est égal a 'B' */`
`ch = ch - 'A' + 'a'; /* ch est égal a 'b' */`
- On peut mélanger caractères et entiers dans les opérations.
`ch = 'A' + 1; /* ch est égal à 'B' */`

2 Les chaînes de caractères

En C, une *chaîne de caractères* est un tableau de `char` comprenant au moins une occurrence d'un caractère de terminaison : le caractère dit « nul », noté `'\0'`. Le caractère de terminaison indique la fin de la zone d'intérêt, ce qui suit ce caractère ne sera pas lu.

Exemple. Considérons le tableau `str` de caractères suivant :

```

-- 0x28
-- | a | b | c | d | \0 | 5 | t | A | 2 | --
-- str
```

- le tableau `str` est un tableau de `char` de 9 éléments
- la chaîne de caractères représentée par `str` est `"abcd"`, de longueur 4.

On peut remarquer ici qu'il y a une différence entre la taille du tableau et la longueur de la chaîne de caractères qu'il représente. La taille minimale d'un tableau de `char` pour contenir une chaîne de caractères de longueur n est $n + 1$ (pour le caractère de terminaison).

Remarque. Une chaîne de caractères doit impérativement contenir un caractère nul à la fin de la chaîne, l'oublier est un source d'erreurs importantes.

1. Comme mentionné en § 1.1 du chapitre 1, le plus souvent ce multipllet correspond à un octet.
2. c'est toujours le cas, mais ça ne marchera pas avec les accents...

Chaîne de caractères littérale

Une chaîne littérale est une suite de caractères entre des guillemets doubles :

"Ceci est chaîne de caractères !"

2.1 Initialisation d'une chaîne de caractères

Lorsqu'un compilateur C rencontre une chaîne littérale de taille n , il alloue une zone mémoire permettant de contenir $n + 1$ caractères (la chaîne de caractères plus le caractère de terminaison). Ainsi, les chaînes littérales sont pratiques pour l'allocation et l'initialisation des chaînes de caractères.

- Avec la syntaxe ci-dessous, on alloue un tableau de `char` de taille minimale pour contenir « Hello », c'est à dire de taille 6.

```
char str1[] = "Hello";
```

0xA0	--		H		e		l		l		o		\0		--	
		str1														

- La syntaxe suivante est identique à la précédente mais moins pratique

```
char str1[] = {'H', 'e', 'l', 'l', 'o', '\0'};
```

- Avec la syntaxe ci-dessous, on alloue un tableau de `char` de taille spécifique 8 contenant la chaîne "Hey".

```
char str2[8] = "Hey !";
```

0xAC	--		H		e		y		\0		?		?		?		?	
		str2																

```
1 /* Attention ici bug : oubli du caractère '\0' */
2 char str3[] = {'H', 'e', 'l', 'l', 'o'};
3 AFFICHER(str3); /* comportement indéfini ! */
4 /* Présence d'un '\0' supplémentaire */
5 char str4[] = "Hel\0lo";
6 AFFICHER(str4, "\n"); /* affiche Hel */
```

Remarque. Comme dans le cas des tableaux de types quelconques, on ne peut pas utiliser l'opérateur d'affectation « = » pour copier le contenu d'une chaîne de caractères dans une autre.

2.2 Parcours d'une chaîne de caractères

Une chaîne de caractères étant un tableau de caractères, on peut utiliser l'opérateur d'indexation « [] » pour accéder à ses éléments.

On parcourt un chaîne de caractère en examinant successivement ses caractères jusqu'à rencontrer le caractère nul.

Exemple. La fonction `compter_espace` parcourt une chaîne de caractères et incrémente de 1 le compteur `cpt` lors qu'un espace est rencontré.

```
1 int compter_espace(char s[])
2 {
3     int cpt = 0, i = 0;
4     while (s[i] != '\0') {
5         if (s[i] == ' ') {
6             cpt++;
7         }
8         i++;
9     }
10    return cpt;
11 }
```

3 Utilisation de la bibliothèque standard

La bibliothèque standard de C, accessible par le fichier d'en-tête `string.h`, fournit un ensemble de fonctions permettant de manipuler des chaînes de caractères. Pour l'inclure, on ajoutera la ligne suivante au début de notre code source :

```
#include <string.h>
```

3.1 strcpy : copie de chaîne de caractères

La fonction `strcpy` accepte en entrée deux chaînes de caractères, et copie la seconde jusqu'au caractère nul (inclus) dans la première.

```
char str1[50];  
char str2[] = "Une nouvelle chaîne";  
strcpy(str1, str2); /* str1 vaut "Une nouvelle chaîne" */
```

Attention, `strcpy` n'a pas de moyen de vérifier que `str1` est suffisamment grand pour contenir tous les caractères de `str2`. Dans le cas où la taille de `str1` est inférieure à celle de `str2`, le comportement est indéfini.

3.2 strlen : longueur d'une chaîne de caractères

La fonction `strlen` retourne la longueur d'une chaîne de caractères passée en argument de la fonction. Attention, cette longueur n'est pas la taille du tableau sous-jacent, il s'agit du nombre de caractères se trouvant avant le caractère nul `'\0'`.

```
int len = strlen("abc"); /* len vaut 3 */  
len = strlen(""); /* len vaut 0 */  
len = strlen("ab\0c"); /* len vaut 2 */
```

3.3 strcat : concaténation de chaîne de caractères

La fonction `strcat` accepte en entrée deux chaînes de caractères, et copie la seconde à la suite de la première.

```
1 /* str1 contient 'a', 'b', 'c', 'd', '\0', ?, ?, ? */  
2 char str1[8] = "abcd";  
3 /* str2 contient 'z', 'y', 'x', '\0', ?, ?, ? */  
4 char str2[] = "zyx";  
5 /* str1 contient 'a', 'b', 'c', 'd', 'z', 'y', 'x', '\0' */  
6 strcat(str1, str2);
```

Encore une fois, attention à l'espace mémoire disponible dans le tableau contenant la première chaîne de caractères.

3.4 strcmp : comparaison de chaînes de caractères

La fonction `strcmp` accepte en entrée deux chaînes de caractères et calcule une distance entre les deux, qui respecte l'ordre *lexicographique*.

Pour deux chaînes de caractères `str1` et `str2`, l'appel `strcmp(str1, str2)` ; retourne :

- une valeur strictement négative si `str1 < str2`,
- 0 si `str1` et `str2` sont identiques,
- une valeur strictement positive si `str1 > str2`.

A+ Exercices et problèmes

Algorithme de base et branchement conditionnel

Exercice 01.

- Écrire un algorithme `échanger` qui accepte en entrée deux variables quelconques, qui échange leurs valeurs et rend en sortie les deux variables d'entrées ainsi modifiées.
- Écrire un algorithme `échanger_circulaire` qui accepte en entrée trois variables quelconques, et qui permute leurs valeurs (la deuxième entrée reçoit la valeur de la première, la troisième reçoit celle de la deuxième et la première celle de la troisième), et qui rend en sortie les variables d'entrées ainsi modifiées.

Exercice 02. Soit l'algorithme `test` suivant

Algorithme `test` : $(a, b) \rightarrow (\text{test1}, \text{test2}, \text{test3}, \text{test4})$ **selon**

`test1` \leftarrow faux, `test2` \leftarrow faux, `test3` \leftarrow faux, `test4` \leftarrow faux

Si `a > b` **alors** `test1` \leftarrow vrai
sinon si `a > 10` **alors** `test2` \leftarrow vrai .

Si `b < 10` **alors** `test3` \leftarrow vrai
sinon `test4` \leftarrow vrai .

.

- Déterminer la sortie de l'algorithme pour les entrées suivantes :
 - `a = 10` et `b = 5` — `a = 5` et `b = 10` — `a = 20` et `b = 10`
 - `a = 5` et `b = 5` — `a = 10` et `b = 10` — `a = 20` et `b = 20`
- Écrire un algorithme équivalent sans utiliser de branchement conditionnel.

Exercice 03. Écrire un algorithme `tri_triplet` qui accepte en entrée un triplet de valeurs et qui les ordonne par ordre croissant. L'algorithme renverra un nouveau triplet ordonné par ordre croissant.

Boucles

Exercice 04. Écrire un algorithme `factorielle` qui accepte en entrée un entier naturel et qui calcule sa factorielle.

On proposera deux versions, une qui utilise une boucle énumérée, l'autre qui utilise une boucle conditionnelle.

Exercice 05 (Algorithme d'Euclide). Écrire un algorithme `pgcd` qui accepte en entrée deux entiers naturels, et qui calcule leur *plus grand diviseur commun*. *Indication: le plus grand diviseur commun de a et b est le même que le plus grand diviseur commun de b et du reste de la division euclidienne de a par b .*

Exercice 06 (Méthode de Héron). Nous avons supposé dans un exemple du cours au [chapitre 2 § 3.1](#) l'existence dans le pseudo-langage d'un opérateur calculant la racine carrée d'un nombre réel. Dans le cas général nous ne dotons pas le pseudo-langage d'un tel opérateur, car il cache des considérations autrement plus complexes que les opérateurs arithmétiques de base.

Étant donné un réel positif x , on cherche un autre réel positif y tel que $y^2 = x$. Si on part d'une approximation $y_0 > 0$, on sait que $y_0 \times \frac{x}{y_0} = x$. Nécessairement, l'un des deux termes y_0

ou $\frac{x}{y_0}$ est supérieur à \sqrt{x} , et l'autre est inférieur : la solution du problème est entre les deux.

On peut alors raffiner l'approximation avec la moyenne des deux, $y_1 = \frac{y_0 + x/y_0}{2}$.

On peut répéter ce processus jusqu'à obtenir une approximation satisfaisante ; par exemple en s'assurant que l'erreur commise est plus faible qu'une petite valeur de tolérance donnée $\epsilon > 0$: $|y_n^2 - x| < \epsilon$.

À partir de ces indications, écrire un algorithme `racine_carrée` acceptant en entrée deux nombres réels positifs, et qui calcule une approximation de la racine carrée du premier, respectant la précision indiquée par le deuxième.

Nota : cet algorithme, connu depuis l'antiquité, est un cas particulier de la Méthode de Newton.

Tableaux

Exercice 07. Écrire un algorithme `calculer_binaire` qui accepte en entrée un tableau composé de 0 et de 1 uniquement (représentation binaire d'un entier naturel où le bit de poids fort se trouve dans la dernière case du tableau), et qui calcule la valeur de l'entier naturel représenté par le tableau. Par exemple :

```
tab ← (1, 0, 1, 0, 1)
```

```
calculer_binaire(tab) # vaut 21 (1×20+0×21+1×22+0×23+1×24)
```

Exercice 08. Écrire un algorithme `somme_et_moyenne` qui accepte en entrée un tableau qui contient des nombres et qui retourne en sortie la somme et la moyenne de tous ses éléments.

Dans les [exercices 09, 0A et 10](#), on considère des tableaux qui contiennent des valeurs que l'on peut comparer avec l'opérateur de relation d'ordre \leq .

Exercice 09. Écrire un algorithme `trouver_minimum` qui accepte en entrée un tableau et qui calcule l'élément minimum, ainsi que son indice dans le tableau.

Exercice 0A (Tri par sélection).

- (a) On considère un tableau et un entier pouvant indexer le tableau. Écrire un algorithme `sélectionner_minimum`, qui cherche l'élément minimum entre l'indice *inclus* et la fin du tableau, et qui échange cet élément minimum avec l'élément de l'indice. Par exemple :

```
tab1 ← (23, 0, 2, -9, 4, 0)
```

```
tab2 ← (2, 7, 6, 9, 3, 0)
```

```
tab ← sélectionner_minimum(tab1, 3) # tab vaut (23, 0, -9, 2, 4, 0)
```

```
tab ← sélectionner_minimum(tab2, 1) # tab vaut (0, 7, 6, 9, 3, 2)
```

- (b) Écrire un algorithme `tri_sélection` qui accepte en entrée un tableau et qui le trie par ordre croissant en faisant appel à l'algorithme précédent.

Indication : Que se passe-t-il pour un tableau quelconque `tab` lorsqu'on effectue les instructions suivantes ?

```
tab ← sélectionner_minimum(tab, 1)
```

```
tab ← sélectionner_minimum(tab, 2)
```

```
tab ← sélectionner_minimum(tab, 3)
```

Exercice 0B (Tri par insertion).

- (a) On considère un tableau et un entier pouvant indexer le tableau. On suppose que le tableau est ordonné par ordre croissant jusqu'à l'indice *exclu*. Écrire un algorithme `insérer_indice`, qui ordonne le tableau par ordre croissant jusqu'à l'indice *inclus*.

Note : pour un tableau `tab` trié par ordre croissant jusqu'à un indice `i exclu`, l'appel `insérer_indice(tab, i)` modifie `tab` en insérant l'élément `tab(i)` "à sa place" dans la partie triée du tableau. Par exemple :

```

tab ← (2, 6, 7, 9, 3, 0)
tab1 ← insérer_indice(tab, 5) # tab1 vaut (2, 3, 6, 7, 9, 0)
# on a inséré l'élément tab(5) (3 est inséré entre 2 et 6).

```

- (b) Écrire un algorithme `tri_insertion` qui accepte en entrée un tableau et qui le trie par ordre croissant.

Exercice 0C (Crible d'Ératosthène). Le *crible d'ERATOSTHÈNE* est un algorithme calculant tous les nombres premiers inférieurs à un entier donné, qui procède comme suit : on place les nombres entiers dans un tableau, et pour chacun d'eux, on parcourt tout le tableau en remplaçant les multiples du nombre de l'itération courante par 0. À la fin de la procédure, les nombres non nuls restant sont les nombres premiers recherchés.

- (a) Écrire l'algorithme `crible_Eratosthène`.
- (b) Proposer des améliorations pour réduire le nombre d'opérations nécessaires.
Indication : si d divise n , alors $d' = n/d$ divise aussi n ; observer que $d \leq \sqrt{n} \iff d' \geq \sqrt{n}$.

Exercice 0D (Recherche de séquence). Écrire un algorithme `rechercher_séquence` qui prend en entrée deux tableaux et qui vérifie si la séquence du deuxième tableau est contenue dans le premier tableau.

Par exemple :

```

tab1 ← (1, 4, 23, 16, 11, 12)
tab2 ← (4, 23, 16)
tab3 ← (4, 11)
rechercher_séquence(tab1, tab2) # vaut vrai
rechercher_séquence(tab1, tab3) # vaut faux

```

Du pseudo-langage au langage C

Exercice 0E (Saisir et afficher un tableau). Pour la mise en œuvre de programme en langage C manipulant des tableaux, nous avons besoin d'un programme de saisie et d'affichage de tableaux.

- (a) Écrire un programme qui :
- Déclare un tableau de flottants de taille 20.
 - Demande à l'utilisateur de fournir un nombre de flottants à saisir en s'assurant que ce nombre est positif, et inférieur ou égal à la taille du tableau
 - Réalise la saisie des valeurs
 - Affiche le tableau
- (b) Que doit-on modifier pour manipuler un tableau d'entiers de maximum 10 éléments ?

Exercice 0F (Calcul de moyenne et somme).

- (a) Déclarer et initialiser un tableau de flottants de maximum 30 éléments, dont la taille effective et les valeurs sont saisies par l'utilisateur ([exercice 0E](#)).
- (b) Écrire un programme qui met en œuvre l'algorithme `somme_et_moyenne` de l'[exercice 08](#) et affiche le résultat à l'écran.
- (c) Réécrire le programme pour un tableau d'entiers. À quoi doit-on faire attention ?

Exercice 10 (Tri par insertion). Écrire un programme qui met en œuvre le tri par insertion d'un tableau d'entiers saisi par l'utilisateur. Le programme affichera le tableau initial, ainsi que le tableau trié.

Exercice 11 (Recherche de séquence). Écrire un programme qui implémente l'algorithme `rechercher_séquence` pour des tableaux d'entiers.

- Le tableau de recherche `tab` sera saisi par l'utilisateur;
- la séquence à chercher contient 3 éléments et est définie et initialisée directement dans le code source;
- le programme affiche les deux tableaux et indique si la séquence du tableau `seq` est incluse dans le tableau `tab`.

Les fonctions en langage C

Exercice 12 (Tri par sélection en C). Dans cet exercice, on reprend l'exercice l'exercice 0A du tri par sélection, et on l'implémente dans le langage C en utilisant des fonctions.

Avant d'écrire la définition d'une fonction, on commencera toujours par écrire la liste des paramètres d'entrée, le type de retour, et l'en-tête de la fonction.

- Écrire une fonction `saisir_tableau` qui réalise la saisie des valeurs d'un tableau de flottants en demandant à l'utilisateur la taille souhaitée.
- Écrire une fonction `afficher_tableau` qui réalise l'affichage d'un tableau de flottants.
- Écrire le programme principal qui déclare un tableau de flottants, le saisit et l'affiche.
- Écrire la fonction `selectionner_minimum` en langage C.
- Écrire la fonction `tri_selection` en langage C.
- Modifier le programme principal pour inclure le tri du tableau, et afficher le tableau trié.

Exercice 13 (Morpion). On souhaite réaliser un jeu de morpion. Rappel des règles : Les joueurs jouent chacun leur tour. À chaque tour, un joueur place un jeton correspondant à son symbole dans une grille 3×3 . Le premier joueur à avoir aligner 3 jetons (en ligne, en colonne ou en diagonale) gagne.

La grille de jeu est modélisée par un tableau statique d'entiers 2D de taille 3×3 .

- La valeur `0` correspondra à une case vide
- La valeur `1` correspondra à un jeton placé par le joueur 1
- La valeur `2` correspondra à un jeton placé par le joueur 2

Un joueur sera représenté par un entier correspondant à son numéro (`1` ou `2`).

Pour réaliser ce projet, nous découperons le problème en plusieurs fonctions :

- Une fonction `initialiser_grille` qui initialise la grille de jeu à `0`.
- Une fonction `afficher_grille` qui permet d'afficher la grille de jeu.
- Une fonction `ajouter_jeton`, qui demande au joueur dont c'est le tour de saisir les coordonnées de la case sur laquelle il souhaite jouer, ajoute le numéro du joueur dans la grille aux coordonnées saisies si elles sont valides. Si les coordonnées saisies ne sont pas correctes, la fonction retourne `0`. Si les coordonnées sont valides, la fonction retourne `1`.
- Une fonction `vainqueur` qui analyse la grille de jeu et détermine s'il y a un vainqueur. Si un vainqueur est identifié, son numéro de joueur est retourné. En cas de match nul la fonction retourne `-1`. Sinon la fonction retourne `0`.

- Écrire les en-tête de chacune de ces fonctions.
- En supposant définies chacune de ces fonctions, écrire le programme principal.

B+ Navigation et compilation en ligne de commande

1 Navigation dans l'arborescence

Le cours de première année sur les outils numériques introduit les notions de fichier numérique, d'encodage, de fichier exécutable, de système d'exploitation, et d'arborescence des fichiers et dossiers.

1.1 Interface en ligne de commande

La plupart des systèmes d'exploitation proposent une interface graphique permettant d'explorer l'arborescence des fichiers et dossiers : on double-clique sur un dossier pour voir les sous-dossiers et fichiers qu'il contient, ou sur d'autres boutons pour remonter au dossier le contenant (qu'on appelle *parent*).

Il est parfois préférable d'utiliser des outils plus fondamentaux. La façon la plus fondamentale d'interagir avec un système d'exploitation est à travers une *interface en ligne de commande*, appelée aussi abusivement *console* ou *terminal*, affichant seulement du texte.

Sous Windows, une telle interface est fournie par le programme CMD, qu'on peut trouver en tapant cmd dans la recherche du menu « Démarrer » . GNU/Linux utilise généralement l'interpréteur de commandes Bash.

TABLE B.1 – Principales commandes pour naviguer avec un terminal.

Windows (CMD)	Linux (Bash)	Fonction et observations
cd	cd	Changer le répertoire courant. Initiales de <i>current directory</i> ou <i>change directory</i> . Les répertoires spéciaux « . », « .. » et « ~ » désignent respectivement le répertoire courant, le répertoire parent, et (seulement sous Linux) le répertoire personnel /home/<utilisateur>. La commande cd sans argument affiche le répertoire courant sous Windows et renvoie au répertoire personnel sous Linux.
cd	pwd	Afficher le chemin du répertoire courant. Initiales de <i>current directory</i> et <i>present working directory</i> .
dir	ls	Lister les éléments du répertoire courant. Abréviations de <i>directory</i> et <i>list</i> .
copy	cp	Copier un fichier; utilisation : <i>copy <source> <destination></i> . Si <destination> n'existe pas, une copie de <source> est créé avec ce nom. S'il existe et que c'est un fichier, il sera remplacé par la copie. S'il existe et que c'est un dossier, une copie de <source> est créée avec le même nom dans ce dossier.
xcopy	cp -r	Copier un dossier et tout ce qu'il contient. L'option -r signifie <i>récuratif</i> .
move	mv	Déplacer un fichier ou un dossier. Abréviations de <i>move</i> .
del	rm	Supprimer un fichier. Abréviations de <i>delete</i> et <i>remove</i> .
rmdir	/S rm -r	Supprimer un dossier et tout ce qu'il contient.
mkdir	mkdir	Créer un dossier vide. Abréviations de <i>make directory</i> .

L'*invite de commande* indique généralement le répertoire de travail, appelé le *répertoire courant*, et parfois les noms de l'utilisateur et de l'ordinateur. Vient ensuite le *curseur*, à partir duquel on peut écrire des commandes pour naviguer dans l'arborescence et exécuter des programmes.

Les commandes de base sont résumées dans la [table B.1](#) et illustrées sur la [console B.1](#). On exécute une commande en appuyant sur la touche *entrée* . De plus, l'utilisation du terminal est grandement facilitée par les touches *tabulation*  pour compléter automatiquement les commandes, et *flèche haut*  pour réutiliser les commandes précédentes.

Windows (CMD)

```
C:\Users\aturing\> cd
C:\Users\aturing
C:\Users\aturing\> cd ../../Programs
C:\Programs\> cd
C:\Programs
C:\Programs\> dir
. . . cmd.exe
C:\Programs\> cd ../../Users/aturing
C:\Users\aturing\> rmdir /S Documents
C:\Users\aturing\> mkdir Multimedia
C:\Users\aturing\> move Music Multimedia
C:\Users\aturing\> move Videos Multimedia
C:\Users\aturing\> dir
. . . Images Multimedia
C:\Users\aturing\> dir Multimedia
. . . Music Videos
```

GNU/Linux (Bash)

```
aturing@machine ~$ pwd
/home/aturing
aturing@machine ~$ cd ../../bin
aturing@machine bin$ pwd
/bin
aturing@machine bin$ ls
bash
aturing@machine bin$ cd
aturing@machine ~$ rm -r Documents
aturing@machine ~$ mkdir Multimedia
aturing@machine ~$ mv Music Multimedia
aturing@machine ~$ mv Videos Multimedia
aturing@machine ~$ ls
. . . Images Multimedia
aturing@machine ~$ ls Multimedia
. . . Music Videos
```

CONSOLE B.1 – Exemple d'utilisation de l'interpréteur de commandes. Les arborescences respectives sont supposées suivre initialement la [figure B.1](#). L'invite de commande est en vert.

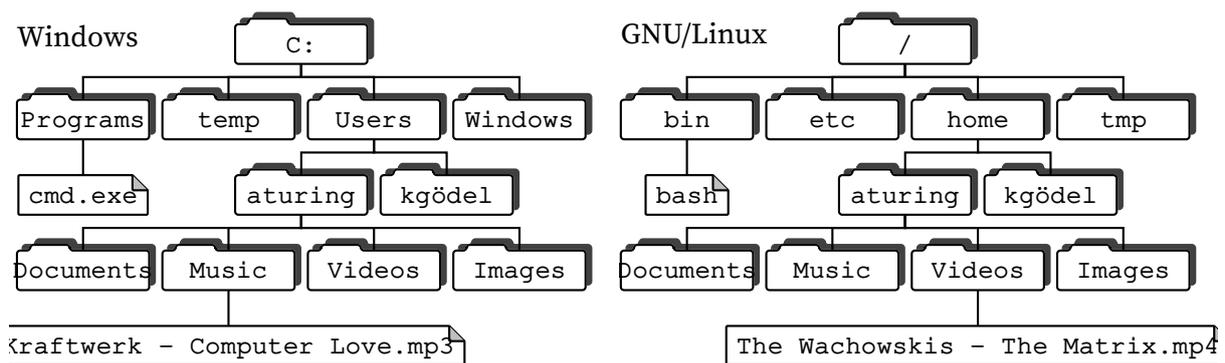


FIGURE B.1 – Illustration d'arborescences avec des noms de dossiers typiques sous Windows et GNU/Linux.

1.2 Disques locaux et disques distants

Les arborescences représentées en [figure B.1](#) sont non seulement partielles, mais elles n'illustrent pas le fait que plusieurs supports physiques d'enregistrement peuvent être accessibles à un même système d'exploitation.

Sous Windows, chaque support physique est généralement la racine d'une arborescence,

identifiée par une lettre de l'alphabet. Le premier disque dur est presque toujours C : ,¹ puis les lettres suivantes sont attribuées successivement aux autres disques durs, puis aux supports amovibles (lecteur optique, clé USB, carte SD, etc.). Enfin, des supports peuvent être accessibles à distance par une connexion réseau; on leur attribue alors une lettre plus loin dans l'alphabet, typiquement U : ou Z : .

C'est le cas sur les ordinateurs de l'INSA, où les disques réseaux permettent aux utilisateurs de retrouver leur travail à partir de n'importe quel ordinateur connecté. Cependant, il ne faut pas travailler directement sur un emplacement réseau, car cela sollicite excessivement la connexion. La solution est de travailler sur le disque local, et de ne pas oublier de rapatrier son travail sur le réseau à la fin de chaque séance.²

2 Programmation et compilation

Les concepts de programmation informatique, langage de programmation, langage machine, code source et compilation ont été introduit au [chapitre 3](#). Nous détaillons ici la création d'un code source et la compilation en ligne de commande.

2.1 Création d'un code source en langage C

Pour écrire du code source, on utilise un programme qui permet de lire et de modifier du texte brut : un *éditeur de texte*. La plupart sont capables d'identifier le langage de programmation à partir de l'extension du fichier source, et d'afficher les mots-clés avec des couleurs adaptées. Cette fonctionnalité s'appelle la *coloration syntaxique*, et facilite grandement la manipulation de code source. Nous conseillons aux débutants Notepad++ sous Windows, et Gedit sous GNU/Linux.³ La [console B.2](#) illustre comment créer notre premier code source C avec l'interpréteur de commandes.

Windows (CMD)

```
C:\Users\aturing\> cd
C:\Users\aturing
C:\Users\aturing\> mkdir TP0
C:\Users\aturing\> cd TP0
C:\Users\aturing\TP0\> notepad++ hello.c
```

GNU/Linux (Bash)

```
aturing@machine ~$ pwd
/home/aturing
aturing@machine ~$ mkdir TP0
aturing@machine ~$ cd TP0
aturing@machine TP0$ gedit hello.c&
```

CONSOLE B.2 – Création d'un répertoire et d'un fichier source. Notons qu'on utilise l'extension `.c` pour les fichiers sources en langage C. Sous GNU/Linux, l'esperluette « & » est nécessaire pour interpréter d'autres commandes après avoir démarré Gedit.

Écrivons alors notre premier programme, suivant le [fichier B.1](#). Il faut bien penser à l'enregistrer à partir de l'éditeur de texte, sinon le fichier ne sera pas créé.

2.2 Compilation et exécution basiques d'un programme en langage C

Pour compiler du code source en langage C, nous utiliserons le compilateur le plus répandu : GCC pour *GNU C compiler*.⁴

1. pour des raisons historiques : A : et B : sont réservées pour des disquettes
2. Il est regrettable que certaines salle de travaux pratiques à l'INSA ne permettent que de travailler sur le réseau : les logiciels nécessaires ne sont pas installés localement.
3. Sur les systèmes Windows de l'INSA, l'éditeur Notepad++ se trouve à l'emplacement `C:\Program files (x86)\Notepad++\notepad++.exe`, mais nous avons configuré le système pour le trouver simplement avec la commande `notepad++`. Sous GNU/Linux, Gedit se trouve généralement à `/usr/bin/gedit`, mais il suffit aussi d'utiliser la commande `gedit`.
4. Sur les systèmes Windows de l'INSA, il se trouve à l'emplacement `C:\TDM-GCC-64\bin\gcc.exe`, mais une fois de plus nous avons configuré le système pour l'appeler simplement avec la commande `gcc`.

```

1 #include <stdio.h>
2 int main ()
3 {
4     printf("Bonjour tout le monde !\n");
5     return 0;
6 }

```

FICHER B.1 – Premier programme, hello.c.

La compilation est un processus compliqué et GCC accepte beaucoup d'options; cependant, nous l'utiliserons essentiellement sous la forme `gcc <sources> -o <exécutable>`, où `<sources>` sera un ou plusieurs fichiers sources (extension `.c`) et l'option `-o` permet de spécifier un nom pour l'exécutable créé⁵; comme illustré sur [console B.3](#).

Windows (CMD)

```

C:\...\TP0\> gcc hello.c -o hello.exe
C:\...\TP0\> dir
. . . hello.c hello.exe
C:\...\TP0\> hello
Bonjour tout le monde !

```

GNU/Linux (Bash)

```

... TP0$ gcc hello.c -o hello
... TP0$ ls
. . . hello.c hello
... TP0$ ./hello
Bonjour tout le monde !

```

CONSOLE B.3 – Compilation et exécution. Sous GNU/Linux, pour lancer un exécutable dans le dossier courant, il est nécessaire de précéder le nom par « `./` ».

Si la compilation réussit, alors le fichier exécutable est créé, il peut être appelé depuis la ligne de commande, comme illustré en [console B.3](#). Si l'exécution ne se passe pas comme prévu, il est possible de l'arrêter avec la combinaison de touches *contrôle* et C  + .

2.3 Débuguer

Il est très fréquent de commettre des erreurs en programmant. Quand on commet une *erreur de syntaxe*, c'est-à-dire que le code source ne respecte pas les règles du langage C, le compilateur ne peut pas créer l'exécutable, et essaie de renseigner le programmeur sur son erreur. Par exemple, la [console B.4](#) reproduit le message d'erreur de GCC si on oublie la dernière accolade à la [ligne 6](#) du [fichier B.1](#).

```

C:\Users\aturing\TP0\> gcc hello.c -o hello.exe
hello.c: In function 'main':
hello.c:5:2: error: expected declaration or statement at end of input
  return 0;
  ^

```

CONSOLE B.4 – Message d'erreur obtenu si on oublie l'accolade fermante « `}` » à la [ligne 6](#) du [fichier B.1](#). Notons qu'un message d'erreur de GCC mentionne le nom du fichier source, la fonction le cas échéant, et les numéros de la ligne et du caractère où l'erreur semble se trouver.

Parfois, le message d'erreur est clair et l'erreur est facile à trouver. Il arrive cependant que, même sur des programmes simples, le message d'erreur soit difficile à comprendre. Quand le compilateur rapporte plusieurs erreurs à la suite, il est conseillé de toujours commencer par traiter la première erreur rapportée. En effet, il arrive qu'une erreur dans une instruction

⁵ Sous GNU/Linux, le chemin de l'exécutable est habituellement `/usr/bin/gcc`, mais il suffit aussi d'utiliser la commande `gcc`.

5. sans cette option, le nom par défaut est `a.exe` ou `a.out`, ce qui n'est pas très informatif.

engendre des erreurs dans les instructions qui suivent, même si ces dernières sont bien écrites.

De plus, il faut se familiariser avec les termes techniques, en français et en anglais ; sur la [console B.4](#), *function* signifie « fonction », *declaration* signifie « déclaration », *statement* signifie « instruction » et *input* signifie « fichier source ».

Aussi, il est utile de supprimer des instructions et de recompiler pour identifier celles qui posent problème. Pour ne pas avoir à les réécrire plus tard, on peut simplement les mettre en commentaires en les entourant par les délimiteurs « `/* ... */` ».

Rappelons ici qu'il ne faut pas oublier d'enregistrer les modifications dans le fichier source avant de recompiler.

Il faut garder à l'esprit que le compilateur ne peut pas deviner l'intention du programmeur. En particulier, certaines erreurs peuvent ne pas être identifiées par le compilateur, parce que les instructions restent conformes aux règles du langage C, bien que ne correspondant pas à ce que le programmeur souhaitait. La compilation s'effectuera alors sans erreur, mais le programme sera incorrect.

Certaines de ces erreurs sont cependant si fréquentes que la plupart des compilateurs ont un mécanisme pour les éviter : les *avertissements* (*warnings* en anglais). Il est vivement conseillé aux débutants de *traiter les avertissements comme des erreurs*, c'est-à-dire de s'efforcer d'écrire du code source qui compile sans avertissement.

2.4 Entrées et sorties simplifiées avec INSAIO

Notre premier programme, [fichier B.1](#), utilise la *bibliothèque standard* de C, avec le *fichier d'en-tête* `stdio.h`, afin d'afficher un message à l'aide de la fonction `printf(...)`. Cette bibliothèque doit être connue de tout programmeur en C, mais est délicate à utiliser pour les débutants.

Pour cette raison, nous utilisons au premier semestre le fichier d'en-tête `insaio.h`, qui fournit les *macro-définitions* `AFFICHER(...)` et `SAISIR(...)`, acceptant jusqu'à neuf arguments. La première affiche les arguments successivement sur le terminal ; la seconde affecte aux variables correspondantes les valeurs que l'utilisateur écrit sur le terminal ; comme illustré sur le [fichier B.2](#) et la [console B.5](#).

```
#include <insaio.h>

int main()
{
    int i;
    float f;
    char str[64];

    AFFICHER("Veuillez saisir un nombre entier, un nombre réel, "
            "puis une chaîne de caractères : ");

    SAISIR(i, f, str);

    AFFICHER("Le nombre entier est ", i, ", le nombre réel est ", f,
            ", et ma chaîne de caractères est \"", str, "\".\n");

    return 0;
}
```

FICHER B.2 – Programme `insaio.c` illustrant l'utilisation de `insaio.h`.

Il est important de noter que le fichier d'en-tête `insaio.h` utilise des fonctionnalités spé-

```
C:\Users\aturing\TP0\> gccinsa insaio.c -o insaio.exe
C:\Users\aturing\TP0\> insaio
Veillez saisir un nombre entier, un nombre réel, puis une chaîne de
caractères : 42 3.14 Bonjour tout le monde !
Le nombre entier est 42, le nombre réel est : 3.140000, et ma chaîne de
caractères est : "Bonjour tout le monde !".
```

CONSOLE B.5 – Compilation et exécution du programme `insaio.c`, [fichier B.2](#). On utilise ici le script `gccinsa`, qui appelle `gcc` avec des options adaptées à `insaio.h`. L'entrée «42 3.14 Bonjour tout le monde !» est écrite par l'utilisateur pendant l'exécution.

cifiques de GCC, et peut ne pas être compatible avec d'autres compilateurs. Pour les étudiants qui souhaiteraient l'utiliser sur leur ordinateur personnel, le fichier est mis à disposition sur la plate-forme numérique Célène. Sous GNU/Linux, il suffit généralement de le copier dans le répertoire `/usr/include/` pour qu'il soit disponible sur tout le système.

Enfin, les macro-définitions définies par `insaio.h` peuvent émettre des messages d'erreurs adaptés pendant la compilation qui facilite encore plus leur utilisation. Pour rendre les messages d'erreurs lisibles, il faut utiliser l'option de GCC `-ftrack-macro-expansion=0`. La commande de compilation complète devient alors `gcc -ftrack-macro-expansion=0 <sources> -o <exécutable>`. C'est un peu long, mais rappelons qu'une fois écrite et exécutée, cette commande peut être récupérée en utilisant la touche flèche haut . Pour simplifier encore plus la tâche, nous mettons aussi à disposition sur les ordinateurs de l'INSA le *script* `gccinsa` qui se charge d'inclure cette option. La commande de compilation devient finalement `gccinsa <sources> -o <exécutable>`, comme utilisée en [console B.5](#).

*

* *