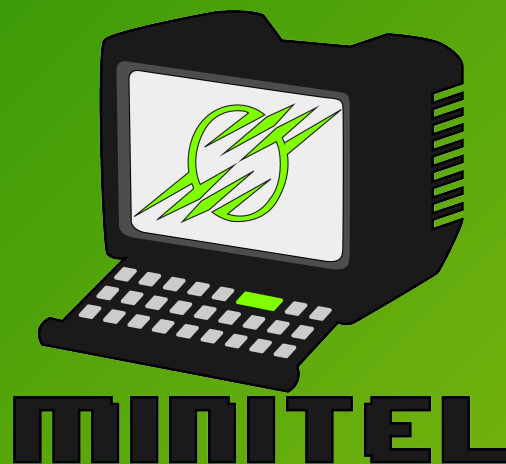


A. Nonyme

# Informatique

Une introduction à la pratique

Version je-sais-pas-combien-mais-pas-encore-la-1.0





# PRÉFACE

Ce document a été écrit dans le but premier d’offrir aux étudiants de l’École Nationale Supérieure des Mines de St-Étienne en cursus ISMIN un support de cours solide pour la programmation en C. De fil en aiguille, il a finit par couvrir une part plus large de l’informatique qui se trouvait nécessaire à la bonne compréhension de la programmation et ne pouvait être considérée comme acquise par toutes les filières de CPGE.

Une bibliographie est disponible en **Annexe** qui propose de manière thématique certains ouvrages spécialisés<sup>1</sup>. Cette bibliographie couvre les thèmes suivants :

- Représentation binaire des nombres [7][11]
- Programmation générale en C [4][9]
- Programmation sous Linux [5]
- Compilation [10][6][8][1]
- Programmation système [3]
- Informatique théorique [2]

Toutes coquilles relevées<sup>2</sup>, suggestions d’amélioration ou autres remarques quelconques peuvent être envoyées à la section “Formation” du serveur Discord de l’association Minitel de l’ISMIN.

---

1. Je n’ai pas tout lu... Pas que l’envie m’en manque, mais classes préparatoires et ISMIN obligent, il n’y a pas toujours le temps de se marrer autant qu’on le voudrait;)

2. MINITEL n’est pas assez riche pour récompenser en dollars hexadécimales ces braves efforts, mais ce serait gentil malgré tout de bien vouloir nous faire part de ces erreurs;)

# Table des matières

Préface . . . . .	1
<b>I Prolégomènes</b>	<b>8</b>
Introduction de l'introduction . . . . .	9
<b>1 Présentation du sujet</b>	<b>10</b>
1.1 L'ordinateur . . . . .	10
1.1.1 Vue de haut . . . . .	10
1.1.2 À l'intérieur . . . . .	11
1.1.3 Données manipulées . . . . .	12
1.2 Opérations logiques . . . . .	14
1.2.1 Arité d'un opérateur, d'une fonction . . . . .	14
1.2.2 Opérations logiques et tables de vérité . . . . .	14
1.2.3 Opérations logiques élémentaires . . . . .	15
1.2.4 Opérations de décalage . . . . .	17
1.2.5 Exercices . . . . .	17
1.3 Opérations arithmétiques . . . . .	18
1.3.1 Représentation d'un nombre entier en binaire . . . . .	18
1.3.2 Nombres signés . . . . .	19
1.3.3 Addition . . . . .	20
1.3.4 Soustraction . . . . .	21
1.3.5 Décalage arithmétique à droite . . . . .	21
1.4 Opérations flottantes . . . . .	22
1.4.1 Représentation des nombres à virgules . . . . .	22
1.4.2 Écriture en base 2 de nombres décimaux . . . . .	23
1.4.3 Norme <i>IEEE 754</i> . . . . .	24
1.4.4 Notation pour la suite du cours . . . . .	26
1.5 Représentation hexadécimale . . . . .	26
1.6 Algorithmes et programmation . . . . .	28
1.6.1 Algorithmes et programmes . . . . .	30
<b>2 Programmer un ordinateur</b>	<b>31</b>
2.1 Un peu de vocabulaire . . . . .	31
2.2 Une classification . . . . .	33
2.2.1 Objectifs des langages de programmation . . . . .	33
2.2.2 Langages compilés et interprétés . . . . .	34
2.3 Le langage C . . . . .	35
2.3.1 Installer un éditeur et un compilateur . . . . .	35
2.3.2 Compiler le premier programme . . . . .	37
2.3.3 Analyse du premier programme . . . . .	38

<b>II Du langage C</b>	<b>41</b>
Introduction . . . . .	42
<b>3 Fondamentaux du langage</b>	<b>44</b>
3.1 Identifiants . . . . .	44
3.2 Commentaires . . . . .	46
3.2.1 Commentaires sur une ligne . . . . .	46
3.2.2 Commentaires par bloc . . . . .	47
3.3 Le point-virgule . . . . .	47
3.4 Point d'entrée du programme . . . . .	48
3.5 Directives préprocesseurs (1) . . . . .	49
<b>4 Bases du langage</b>	<b>50</b>
4.1 Variables . . . . .	50
4.1.1 Taille et type d'une variable . . . . .	52
4.1.2 Entiers signés et non signés . . . . .	53
4.1.3 Introduction à la portée des variables : les blocs . . . . .	54
4.1.4 Exercices . . . . .	55
4.2 Formatage de texte . . . . .	55
4.2.1 Formatage des caractères spéciaux . . . . .	56
4.2.2 Formatage de variables . . . . .	56
4.2.3 Exercices . . . . .	57
4.3 Opérateurs sur les variables . . . . .	57
4.3.1 Opérateurs arithmétiques . . . . .	59
4.3.2 Opérateurs bit-à-bit . . . . .	60
4.3.3 Opérateurs logiques, ou relationnels . . . . .	60
4.3.4 Opérateurs d'assignation . . . . .	61
4.3.5 Exercices . . . . .	63
4.4 Projection de type . . . . .	65
4.4.1 Exercices . . . . .	66
4.5 Structures de contrôle . . . . .	66
4.5.1 Structures élémentaires . . . . .	67
4.5.2 Structures complexes . . . . .	70
4.5.3 Exercices . . . . .	76
4.6 Routines . . . . .	76
4.6.1 Signature et prototype . . . . .	78
4.6.2 Note relative à la copie des arguments . . . . .	79
4.6.3 La pile d'exécution . . . . .	80
4.6.4 Exercices . . . . .	82
4.7 Pointeurs . . . . .	82
4.7.1 Formatage en chaîne de caractères . . . . .	84
4.7.2 Les pointeurs en paramètres de routines . . . . .	85
4.7.3 Projection de type . . . . .	86
4.7.4 Pointeurs itérés . . . . .	88
4.7.5 Exercices . . . . .	89
4.8 Interagir avec les flux standards . . . . .	90
4.8.1 Flux d'entrée standard . . . . .	91
4.8.2 Flux de sortie standard et flux d'erreur standard . . . . .	91
4.8.3 Exemple pratique d'utilisation de <i>stderr</i> . . . . .	93
4.8.4 Exercices . . . . .	93
4.9 Tableaux statiques . . . . .	93
4.9.1 Définition . . . . .	95

4.9.2	Les tableaux statiques, kékoï pour de vrai ?	96
4.9.3	Exercices	99
4.10	Tableaux dynamiques	100
4.10.1	Introduction au tas	100
4.10.2	Les routines <code>malloc</code> et <code>free</code>	100
4.10.3	Exercices	102
4.11	Tableaux multidimensionnels	102
4.11.1	Tableaux multidimensionnels statiques	102
4.11.2	Tableaux multidimensionnels dynamiques	103
4.11.3	Exercices	105
4.12	Structures	105
4.12.1	Initialisation à la déclaration	109
4.12.2	Exercices	110
4.13	Modulation et entêtes	111
4.13.1	Fichier d'entête	112
4.13.2	Fichier de code source	113
4.13.3	Un exemple simple	113
4.13.4	Compilation modulaire	115
4.13.5	Problème des chaînes d'inclusions	117
4.13.6	Garde-fous	119
4.13.7	Exercices	120
4.14	Chaînes de caractères	120
4.14.1	Caractères ASCII	120
4.14.2	Chaîne de caractères	121
4.14.3	En langage C	122
4.14.4	Se faciliter la vie avec les chaînes littérales	123
4.14.5	Exercices	123
4.15	Les flux de fichiers	124
4.15.1	Droits des fichiers	124
4.15.2	Les flux de fichiers en C	126
4.15.3	Rediriger un flux standard vers un fichier	128
4.15.4	Exercices	129
<b>5</b>	<b>Concepts avancés (en cours de rédaction)</b>	<b>130</b>
5.1	Paramètres d'un programme	130
5.1.1	Le véritable prototype de <i>main</i>	131
5.1.2	Exercices	131
5.2	Unions	132
5.2.1	Motivation par un exemple	132
5.2.2	Principe et syntaxe	133
5.2.3	Exercices	134
5.3	Champs de bits	134
5.3.1	Motivation et principe	134
5.3.2	Syntaxe	135
5.3.3	Le revers de la médaille	136
5.4	Classes de stockage	138
5.4.1	Quelques détails de la structure d'un programme en mémoire	138
5.5	Espaces locaux et espace global	141
5.6	Construction de littéraux	141
5.7	Pointeurs de routines	141
5.8	Tableaux de routines	141
5.9	Fonctions variadiques	141

5.10	Alignement . . . . .	141
5.11	Directives de préprocesseurs (2) . . . . .	141
5.12	Assembleur en C . . . . .	141
5.13	La virgule . . . . .	141
5.13.1	La virgule comme opérateur . . . . .	142
5.13.2	La virgule comme séparateur . . . . .	143
5.13.3	Conclusion . . . . .	143
5.14	Tricks récréatifs (ft. Basile) . . . . .	143
5.14.1	Indexation inversée et boutisme . . . . .	144
5.14.2	Définition <i>K&amp;R</i> de fonctions . . . . .	144
5.14.3	Digraphes et trigraphes . . . . .	144
5.14.4	Un peu d'imagination . . . . .	145
<b>III</b>	<b>Le vrai monde de la réalité réelle</b>	<b>148</b>
	Introduction . . . . .	149
<b>6</b>	<b>Outils utiles à la programmation</b>	<b>150</b>
6.1	La documentation . . . . .	150
6.1.1	Liste des sites d'information . . . . .	150
6.1.2	Documentation hors-ligne . . . . .	150
6.1.3	Lire un prototype . . . . .	150
6.1.4	Liste des sites de forum . . . . .	151
6.1.5	À propos des GPTs . . . . .	151
6.2	Makefiles . . . . .	152
6.2.1	Généralités . . . . .	152
6.2.2	Principe et premiers exemples . . . . .	152
6.2.3	Personnaliser la production grâce aux variables . . . . .	153
6.2.4	Variables automatiques . . . . .	154
6.2.5	Réduire le nombre de règles avec la <i>stem</i> . . . . .	155
6.2.6	Les caractères génériques . . . . .	155
6.2.7	Substitution de chaînes . . . . .	156
6.2.8	Les règles virtuelles . . . . .	157
6.2.9	Idées pour aller plus loin . . . . .	158
6.3	Créer sa bibliothèque externe . . . . .	158
6.4	Utiliser un débogueur . . . . .	158
<b>7</b>	<b>Méthodes de programmation propre</b>	<b>159</b>
7.1	La gestion des erreurs . . . . .	159
7.1.1	Un petit exemple pour prendre la température . . . . .	160
7.1.2	Codes d'erreur . . . . .	161
7.2	Tests unitaires . . . . .	161
7.3	À propos du style . . . . .	161
<b>IV</b>	<b>Annexes</b>	<b>162</b>
<b>8</b>	<b>Correction des 42 exercices</b>	<b>163</b>
8.1	Première partie . . . . .	163
8.1.1	Opérations logiques sur les mots binaires . . . . .	163
8.1.2	Opérations arithmétiques . . . . .	166
8.1.3	Opérations sur les nombres à virgules . . . . .	166

8.1.4	Représentation hexadécimale . . . . .	166
8.2	Deuxième partie . . . . .	166
8.2.1	Bases du langage . . . . .	166
8.2.2	Concepts avancés . . . . .	181

<b>Bibliographie</b>		<b>184</b>
----------------------	--	------------

## Liste des tableaux

1.1	Table de vérité de l'opérateur $\vee$ . . . . .	15
1.2	Table de vérité de l'opérateur $\oplus$ . . . . .	15
1.3	Table de vérité de l'opérateur $\wedge$ . . . . .	16
1.4	Table de vérité de l'opérateur $\neg$ . . . . .	16
4.1	Priorités des opérateurs en C . . . . .	59
4.2	Opérateurs arithmétiques . . . . .	59
4.3	Opérateurs bit-à-bit . . . . .	60
4.4	Opérateurs logiques . . . . .	61

## Liste des définitions

1.6.1	Algorithme . . . . .	28
1.6.2	Agent algorithmique . . . . .	29
1.6.3	Programme . . . . .	30
2.1.4	BIOS . . . . .	31
2.1.5	Système d'exploitation . . . . .	31
2.1.6	Répertoire . . . . .	32
2.1.7	Fichier . . . . .	32
2.1.8	Extension de nom fichier . . . . .	32
2.1.9	Système de fichiers . . . . .	32
2.1.10	Répertoire de travail . . . . .	32
2.1.11	Chemin d'accès . . . . .	32
2.1.12	Instruction . . . . .	33
2.1.13	Exécutable . . . . .	33
2.1.14	Niveau d'abstraction . . . . .	33
3.1.15	Identifiant . . . . .	44
4.3.16	Opérateurs et opérandes . . . . .	57
4.3.17	Vérité d'une expression . . . . .	60
4.5.18	Flot d'exécution/de contrôle . . . . .	66

4.6.19	Prototype . . . . .	78
4.6.20	Signature . . . . .	78
4.6.21	Argument . . . . .	79
4.8.22	Flux . . . . .	90
4.9.23	Structure de donnée . . . . .	93
4.13.24	Module et programmation modulaire . . . . .	112
5.4.25	Portée d'une variable et durée de vie . . . . .	138



Partie I

# PROLÉGOMÈNES



## INTRODUCTION DE L'INTRODUCTION

Ce document vise à donner une base solide en informatique « pratique », c'est-à-dire en programmation. Cette pratique est toutefois fondée sur la théorie. Il est donc nécessaire à l'ingénieur de maîtriser également quelques bases théoriques relatives à l'informatique, et plus précisément à la programmation.

C'est dans cette optique que cette première partie s'attache à décrire certains fondamentaux qui peuvent, dépendamment de la filière, ne pas avoir été étudiés. En particulier :

- les composants fondamentaux d'un ordinateur classique
- les opérations logiques élémentaires sur les mots binaires
- les représentations binaires standards des nombres entiers et flottants<sup>3</sup>
- une première approche simple<sup>4</sup> de l'idée d'algorithme
- quelques premières définitions de vocabulaire technique relatif à l'informatique pratique

Pour cela, il est supposé que le/la lecteur/rice a effectué deux ou trois ans de classes préparatoires et se trouve à la lecture de ce document en entrée d'école d'ingénieur. Un certain bagage mathématique est donc considéré comme acquis.

Pour finir, un encadrement sera proposé pour installer les outils de base nécessaires à la programmation en langage C.

---

3. Faut pas haïr les maths, ou ça risque d'être *complicado*

4. Qui se veut simple en tout cas...

## CHAPITRE

# 1

## PRÉSENTATION DU SUJET



### L'ORDINATEUR



#### 1.1.1 Vue de haut

On peut considérer un ordinateur comme un système évoluant selon des entrées pour produire des sorties :



Ces entrées et sorties sont captés et produites via des appareils appelés périphériques d'entrée/sortie. On garde généralement l'appellation en anglais *Input/Output Drivers* ou simplement *I/O Drivers*.

**Petit aparté : De la rigueur de la définition d'un "ordinateur"**

Un ordinateur est appareil qui "calcule". La signification mathématique du terme "calculer" n'a rien d'évidente. Calculer est au départ une notion que l'on peut qualifier d'assez intuitive et il est donc difficile d'apporter une définition formelle claire qui satisfasse ce qu'un humain peut subjectivement appeler le "calcul".

La notion de calculabilité, qui définit ce qui est calculable et ce qui ne l'est pas, apporte une définition du calcul. Cette notion est fondé sur des modèles fondamentaux du calcul (tous équivalents, et heureusement!) comme les machines de Turing et le  $\lambda$ -calcul. Les ordinateurs contemporains sont fondés sur le modèle de la machine de Turing et sont dits équivalents au modèle de la machine de Turing.

Sans plus de détail <sup>a</sup> car cela serait tout à fait hors-sujet, la machine de Turing est basiquement une bande infinie de caractères munit d'une tête de lecture/écriture qui se déplace latéralement sur cette bande et agit selon la valeur des caractères lus grâce à une fonction de transition.

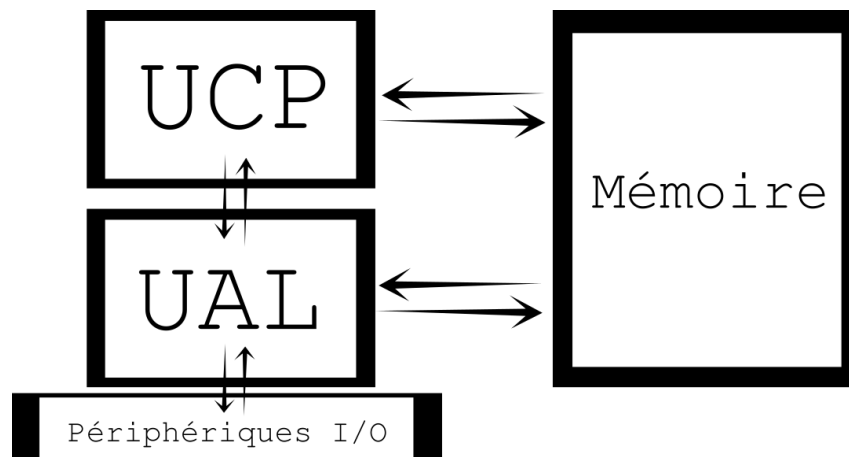
<sup>a</sup>. Pour plus de détail, voir le livre [2]

Le symbole *I/O* est extrêmement récurrent puisqu'il apparaît dans tous les cas où existe un système d'entrée et de sortie d'informations.

### 1.1.2 À l'intérieur

Les périphériques permettent l'interaction avec le système de l'ordinateur. Pourtant, celui-ci peut très bien fonctionner sans. Par exemple, on lisait les résultats des premiers ordinateurs directement à l'intérieur, et on y entrait les programmes "à la main", en modifiant les branchements des câbles pour "écrire le programme". En soi, cette notion d'écriture est assez récente, puisqu'elle date des ordinateurs disposant d'un clavier et d'un écran, périphériques d'entrée/sortie fondamentales pour une interaction "facile" avec un ordinateur.

L'ordinateur lui-même est uniquement un calculateur possédant une mémoire. Il est représenté de manière simplifiée par le schéma suivant qui représente la modélisation d'un ordinateur appelée *architecture de Von Neumann* :



**UAL** signifie Unité Arithmétique et Logique. Ce système constitué de circuits informatiques permet d'effectuer toutes les opérations fondamentales de l'ordinateur, comme l'addition, la multiplication, la soustraction, la division, la lecture de la mémoire et l'écriture dans la mémoire.

**UCP** signifie Unité de Contrôle de Programme. Ce système supervise l'UAL, et contrôle son exécution. Il s'occupe en particulier de "comprendre" les instructions lues dans la mémoire pour faire exécuter par l'UAL les opérations correctes. Il effectue aussi le passage d'une instruction à l'autre.

Les registres forment un système de stockage de capacité très (très (très)) faible, destinés à stocker des valeurs temporaires. Ils servent en particulier de tampons pour les calculs de l'UAL et de l'UCP.

**Remarque 1 :** L'UCP et l'UAL, avec les registres, forment l'**UCC** (Unité Centrale de Calcul, **CPU** en anglais).

**Remarque 2 :** Le schéma de l'ordinateur présenté ici permet de comprendre *dans les grandes lignes* le fonctionnement d'un ordinateur, mais n'est ni complet ni particulièrement détaillé. Ainsi, ni l'alimentation, ni la carte mère, ni une potentielle carte graphique ne sont explicités ici. Ce schéma propose une vision très abstraite de ce qu'est un ordinateur et ne détaille rien des techniques mises en oeuvre.

La mémoire elle-même n'est dans les faits pas constituée d'un unique bloc mais de plusieurs supports de mémoire dont la vitesse et la capacité varient. On distingue par ordre de vitesse croissante et de capacité décroissante :

- les bandes magnétiques (très lents, capacité native de 18 *To* avec la technologie LTO-9, voir [https://fr.wikipedia.org/wiki/Linear\\_Tape-Open](https://fr.wikipedia.org/wiki/Linear_Tape-Open))
- les disques durs (lents, capacité allant de 250 *Go* à 2 *To*)
- la mémoire RAM (pour Random Access Memory) (rapide, capacité allant de 1 *Go* à 8 *Go*, et par agrégation jusqu'à 128 *Go* voire plus).
- la mémoire SRAM (pour Static RAM) (5 à 10 fois plus rapide que la RAM, même capacité mais beaucoup plus chère)
- les registres (extrêmement rapide, capacité allant de 1 octet à 8 octets, 16 octets pour les ordinateurs spécialisés en calcul scientifique).

Ces types de support mémoire ne seront pas toutes détaillées dans la suite, en tout cas pas dans leur principe de fonctionnement. L'utilité de certains supports sera tout de même expliqué (notamment le disque dur et la mémoire RAM, ainsi que certains registres ultérieurement).

**Remarque :** En considérant seulement l'ensemble constitué de l'UAL et de l'UCP, la mémoire peut aussi être vue comme un support d'entrée/sortie. Ainsi, la lecture d'une information dans la mémoire constitue une entrée du système (UAL + UCP), et l'écriture d'une information dans la mémoire constitue une sortie du système (UAL + UCP).

### 1.1.3 Données manipulées

#### Binaire

Un ordinateur est une machine constituée de circuits électroniques. Les informations qui circulent à l'intérieur de celui-ci sont donc des signaux électriques, caractérisés par leur tension. Cette tension traduit deux états, pour des raisons de stabilité et de facilité d'implantation. Le premier état représente l'absence de tension ( $U \approx 0$  V), et le second état représente la présence d'une tension ( $U \approx V_{ref} > 0$ ).

Le premier état est représenté par le chiffre<sup>1</sup> 0 et le second par le chiffre<sup>2</sup> 1. Chaque chiffre (0 ou 1) est appelé un *bit* (*binary digit* en anglais). Cette modélisation sera justifiée par la suite de par les liens efficaces que l'on peut établir entre ces séquences d'états et  $\mathbb{N}$ .

1. Il ne s'agit que d'un symbole sans signification.

2. *idem*

Toutes les données manipulées par un ordinateur sont donc constituées de 0 et de 1. L'ensemble  $\{0, 1\}$  est en théorie du langage un ensemble de *lettres*. Une séquence de lettres est appelée un *mot* et le langage binaire est l'ensemble des mots écrits avec l'alphabet  $\{0, 1\}$ . On ne peut pas encore appeler le mot 1000010 un nombre puisqu'il ne lui est pour l'instant associé aucune interprétation numérique<sup>3</sup>.

### Octets

Pour structurer l'information, on regroupe ces bits par paquets de 8 appelés *octets*. Un octet est donc un nombre écrit avec 8 bits.

On a ensuite les mêmes préfixes du système international que pour n'importe quelle unité physique :

- 1 *Ko* = 1000 *o*
- 1 *Mo* = 1000 *Ko*
- 1 *Go* = 1000 *Mo*
- 1 *To* = 1000 *Go*
- 1 *Po* = 1000 *To*
- etc...

Ainsi que d'autres préfixes plus spécifiques à l'informatique qui sont basés sur l'interprétation entière des mots binaires :

- 1 *Kio* (kibiocet) =  $2^{10}$  *o* = 1024 *o*
- 1 *Mio* (mebiocet) =  $2^{10}$  *Kio* = 1024 *Kio*
- 1 *Gio* (gibiocet) =  $2^{10}$  *Mio* = 1024 *Mio*
- 1 *Tio* (tebiocet) =  $2^{10}$  *Gio* = 1024 *Gio*
- 1 *Pio* (pebiocet) =  $2^{10}$  *Tio* = 1024 *Tio*
- etc...

#### Petit aparté : octet comme unité d'information

L'octet est l'unité de l'information. Il permet de mesurer la quantité d'information enregistrée (voir théorie de l'information de Shannon). En informatique pratique, il est utilisé comme mesure du stockage de l'information (ce qui est légèrement différent de la mesure de l'information qui est son sens premier). La pratique a donc légèrement déformé le sens originel du bit.

On parle d'ailleurs souvent d'un *bit d'information*, qui fait référence à la théorie de l'information de Shannon.

Les données sur un ordinateur sont mesurés en octets. La mémoire vive, ou d'un disque dur, consiste en un immense mot binaire, qui puisqu'il est divisé en groupes de 8 est en fait un immense tableau de cases de un octet chacune.

Le numéro de chaque case est appelé son adresse. La première case est d'adresse 0 et la dernière dans une mémoire de  $T$  octets est  $T - 1$ .

3. Il existe des interprétations non numériques de mots binaires. Par exemple, on peut associer à chaque lettres  $a, b, \dots, z$  un mot binaire spécifique. Cela revient à interpréter un mot binaire comme une lettre.



# OPÉRATIONS LOGIQUES



Les opérateurs sur les mots binaires sont des fonctions qui permettent d'agir sur les mots binaires, de les modifier.

## 1.2.1 Arité d'un opérateur, d'une fonction

Est appelé *arité* d'une fonction son nombre de paramètres.

Un opérateur 1-aire, dit *unaire* (ou encore parfois *monadique*), n'agit que sur une seule variable. Il ne possède qu'un seul paramètre.

Un opérateur 2-aire, dit binaire, possède deux paramètres. Il agit sur deux variables. Par exemple, l'addition de deux nombres est une opération binaire.

Il est naturellement possible de considérer les opérations unaires sur un mot binaire de  $N$  bits comme des fonctions  $N$ -aires, c'est-à-dire à  $n$  paramètres. Par exemple, on pourrait considérer que l'inversion de bits d'un mot binaire de  $N$  bits est en vérité une opération  $N$ -aire puisqu'elle agit sur  $N$  bits qui constituent  $N$  paramètres. De même, une opération binaire de deux mots de  $N$  bits peut être considérée comme  $2N$ -aire.

Toutefois, après avoir posé  $N$  comme le nombre de bits d'un mot binaire, un mot binaire de  $N$  bits est *par convention* considéré comme un unique paramètre pour l'ensemble des opérateurs. Ainsi, l'ensemble des opérateurs décrits ci-dessous sont des opérateurs unaires ou binaires uniquement.

## 1.2.2 Opérations logiques et tables de vérité

Les opérations dites *logiques* sont des opérations qui s'effectuent sur les lettres de mots binaires sans nécessaire considération pour une quelconque interprétation numérique de ces mots.

Les *tables de vérité* sont un moyen commode de visualiser les opérations logiques. Il s'agit de représenter toutes les sorties d'une fonction opératrice selon toutes les entrées possibles. Pour une fonction d'opérateur  $N$ -aire, on a  $2^N$  possibilités d'entrées<sup>4</sup>. Il faut décrire chacune des  $2^N$  sorties associées pour décrire la fonction et ainsi l'opérateur.

**Remarque :** Il s'agit d'une définition de fonction dite *par extension* dans laquelle on liste toutes les correspondances. La manière classique de décrire une fonction grâce à une expression<sup>5</sup> est appelée définition *par compréhension*. Ce vocabulaire est emprunté à la théorie des ensembles puisqu'une fonction est un produit cartésien.

**Exemple :** Voici une table de vérité d'une fonction logique  $f$  3-aire, dont on note les trois entrées  $A$ ,  $B$  et  $C$  :

---

4. En effet, il n'y a que deux lettres dans l'alphabet  $\{0, 1\}$ .

5. Comme  $f : x \mapsto x^2$  par exemple

$A$	$B$	$C$	$f(A, B, C)$
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	1

On observe que le caractère *exponentiel* du nombre d'entrées et de sorties rend cette représentation inutilisable pour des fonctions d'arité grande.

En interprétant la lettre/symbole 0 par le nombre 0 et la lettre/symbole 1 par le nombre 1, on peut aussi décrire  $f$  par compréhension :

$$\begin{aligned} f &: \{0, 1\}^3 \rightarrow \{0, 1\} \\ (A, B, C) &\mapsto AB + BC + CA - 2ABC \end{aligned}$$

### 1.2.3 Opérations logiques élémentaires

Il existe quelques opérateurs logiques classiques très largement utilisés en informatique :

- le OU logique exclusif
- le OU logique inclusif
- le ET logique
- le NON logique

On peut expliquer les noms donnés à ces opérateurs en considérant l'interprétation suivante des symboles 0 et 1 :

- 0 correspond à la valeur logique « Faux »
- 1 correspond à la valeur logique « Vrai »

#### OU logique inclusif

L'opérateur binaire OU logique inclusif, noté  $\vee$ , est décrit par la table de vérité suivante :

$A$	$B$	$A \vee B$
0	0	0
0	1	1
1	0	1
1	1	1

TABLE 1.1 – Table de vérité de l'opérateur  $\vee$

$A \vee B$  est vraie si et seulement si l'une OU l'autre des deux entrées est vraie.

#### OU logique exclusif

L'opérateur binaire OU logique exclusif, noté  $\oplus$ , est décrit par la table de vérité suivante :

$A$	$B$	$A \oplus B$
0	0	0
0	1	1
1	0	1
1	1	0



TABLE 1.2 – Table de vérité de l'opérateur  $\oplus$ 

$A \oplus B$  est vraie si et seulement si l'une *OU* l'autre des deux entrées est vraie, mais pas les deux en même temps!

### ET logique

L'opérateur binaire ET logique exclusif, noté  $\wedge$ , est décrit par la table de vérité suivante :

$A$	$B$	$A \wedge B$
0	0	0
0	1	0
1	0	0
1	1	1

TABLE 1.3 – Table de vérité de l'opérateur  $\wedge$ 

$A \wedge B$  est vraie si et seulement si les deux entrées sont vraies en même temps.

### NON logique

Le NON logique  $\neg$  est une fonction unaire/monadique :

$A$	$\neg A$
0	1
1	0

TABLE 1.4 – Table de vérité de l'opérateur  $\neg$ 

Ainsi,  $\neg A$  est vraie si et seulement si  $A$  est faux et inversement.

### Extension aux mots

On pose  $\mathcal{B} = \{0, 1\}$  et  $N \in \mathbb{N}^*$

Toutes les opérations décrites ci-dessus peuvent être étendues sur l'ensemble des  $N$ -uplets de  $\mathcal{B}^N$ .

Soit  $*$   $\in \{\vee, \wedge, \oplus\}$ . Soient  $a, b \in \mathcal{B}^N$ ,  $a = \begin{pmatrix} a_0 \\ \vdots \\ a_{N-1} \end{pmatrix}$  et  $b = \begin{pmatrix} b_0 \\ \vdots \\ b_{N-1} \end{pmatrix}$ . Alors :

$$a * b = \begin{pmatrix} a_0 * b_0 \\ \vdots \\ a_{N-1} * b_{N-1} \end{pmatrix}$$

De plus (valable pour toute fonction logique monadique autre que  $\neg$ ) :

$$\neg v = \begin{pmatrix} \neg v_0 \\ \vdots \\ \neg v_{N-1} \end{pmatrix}$$

### 1.2.4 Opérations de décalage

#### Décalage à gauche

L'opérateur  $\ll$  est défini comme suit :

$$\begin{aligned} \ll : \mathcal{B}^N \times \llbracket 0, N \rrbracket &\rightarrow \mathcal{B}^N \\ ((a_{N-1} \dots a_0), i) &\mapsto a_{N-1-i} \dots a_0 \underbrace{0 \dots 0}_{i \text{ fois}} \quad \text{si } i < N \\ &\mapsto 0 \dots 0 \quad \text{si } i = N \end{aligned}$$

#### Décalage logique à droite

L'opérateur  $\gg_l$  est défini comme suit :

$$\begin{aligned} \gg_l : \mathcal{B}^N \times \llbracket 0, N \rrbracket &\rightarrow \mathcal{B}^N \\ ((a_{N-1} \dots a_0), i) &\mapsto \underbrace{0 \dots 0}_{i \text{ fois}} a_{N-1} \dots a_i \quad \text{si } i < N \\ &\mapsto 0 \dots 0 \quad \text{si } i = N \end{aligned}$$

### 1.2.5 Exercices

Dans tous les exercices,  $\mathcal{B} = \{0, 1\}$  et  $N \in \mathbb{N}^*$

**Exercice 1 (La compréhension pour mieux comprendre).** Décrire les fonctions  $\vee$ ,  $\oplus$ ,  $\wedge$  et  $\neg$  par compréhension.

**Exercice 2 (Universalité des fonctions logiques élémentaires).** Montrer que toute fonction logique  $N$ -aire peut s'exprimer comme une combinaison des fonctions  $\vee$ ,  $\wedge$  et  $\neg$ .

**Exercice 3 (Porte NAND).** On pose la fonction  $\uparrow$  dite « NAND » définie par :

$$\begin{aligned} \uparrow : \mathcal{B}^2 &\rightarrow \mathcal{B} \\ (A, B) &\mapsto \neg(A \wedge B) \end{aligned}$$

1. Déterminer la table de vérité de l'opérateur NAND
2. Exprimer chacune des fonctions  $\vee$ ,  $\wedge$  et  $\neg$  par compréhension en utilisant uniquement la fonction  $\uparrow$

*Note :* Un opérateur capable d'exprimer les opérateurs logiques élémentaires à lui seul est dit « universel »

**Exercice 4 (Porte NOR).** On pose la fonction  $\downarrow$  dite « NOR » définie par :

$$\begin{aligned} \downarrow : \mathcal{B}^2 &\rightarrow \mathcal{B} \\ (A, B) &\mapsto \neg(A \vee B) \end{aligned}$$

1. Déterminer la table de vérité de l'opérateur NOR
2. Montrer que  $\downarrow$  est un opérateur universel.

**Exercice 5 (Petit retour à l'algèbre fondamentale).** Démontrer que  $G = (\mathcal{B}^N, \oplus)$  est un groupe commutatif. Pour tout  $b \in \mathcal{B}^N$ , déterminer  $b^{-1}$ .

Rappel : Un groupe est un ensemble  $E$  muni d'une loi de composition interne  $*$  associative respectant les propriétés suivantes :

1. il existe un unique élément neutre  $e$  tel que pour tout  $x \in E$ ,  $x * e = e * x = x$

2. chaque élément  $x \in E$  admet un inverse  $x^{-1}$ , c'est-à-dire tel que  $x * x^{-1} = x^{-1} * x = e$ .

Un groupe est commutatif si  $*$  est commutative.

**Exercice 6 (Inversibilité des décalages).** Est-ce que les décalages logiques droit et gauche sont inversibles ? Justifier.



## OPÉRATIONS ARITHMÉTIQUES



Sont ici décrites les opérations arithmétiques sur  $\mathcal{B}^N$ . L'arithmétique définie est une arithmétique modulaire, c'est-à-dire que l'ensemble  $\llbracket 0; 2^N \rrbracket$  est assimilé à l'ensemble des représentants des classes d'équivalence de  $\frac{\mathbb{Z}}{2^N \mathbb{Z}}$ .

### 1.3.1 Représentation d'un nombre entier en binaire

On considère la fonction suivante :

$$\begin{aligned} btoi_u : \mathcal{B}^N &\rightarrow \llbracket 0; 2^N \rrbracket \\ b &\mapsto \sum_{i=0}^{N-1} (b_i 2^i) \end{aligned}$$

Cette fonction calcule la valeur numérique d'un mot binaire. Il s'agit de l'interprétation d'un  $N$ -uplet de bits comme écriture en base 2 d'un nombre entier naturel (sans signe : *unsigned* en anglais). Comme l'écriture d'un nombre en base 2 existe et est unique pour tout nombre naturel, il s'agit d'une bijection de  $\mathcal{B}^N$  dans  $\llbracket 0; 2^N \rrbracket$ .

Sous la forme d'un polynôme de Horner évalué en 2 on a :

$$btoi_u(b) = b_0 + 2(b_1 + 2(b_2 + 2(\dots))) \quad (1.1)$$

On écrit plus simplement :

$$n = (b_{N-1}b_{N-2} \dots b_1b_0)_2 = btoi_u(b)$$

Cette écriture est appelée représentation en base 2 de  $n$ , ou encore représentation binaire de  $n$ .

**Remarque :** On peut trouver la représentation en base 2 de  $n$  par des divisions entières successives par 2 grâce à l'équation (1.1). Pour tout  $i \in \llbracket 0; N-1 \rrbracket$ , on a  $b_i \equiv \left\lfloor \frac{n}{2^i} \right\rfloor \% 2$  [2]

Cette remarque vient également corroborer l'expression de  $btoi^{-1}$  :

$$\begin{aligned} btoi^{-1} : \mathbb{Z} &\rightarrow \mathcal{B}^N \\ n &\mapsto \begin{pmatrix} n \% 2 \\ \vdots \\ \left\lfloor \frac{n}{2^i} \right\rfloor \% 2 \\ \vdots \\ \left\lfloor \frac{n}{2^{N-1}} \right\rfloor \% 2 \end{pmatrix} \end{aligned}$$

où l'opération *modulo* noté  $\%$  est défini pour tout  $a, b \in \mathbb{Z}$  par  $a \% b = a - b \left\lfloor \frac{a}{b} \right\rfloor$ , c'est-à-dire qu'en posant la division euclidienne de  $a$  par  $b$ , il existe un unique  $q$  tel que  $a = bq + (a \% b)$  et  $0 \leq a \% b < b$

Par exemple, trouvons la représentation binaire de 53 :

$$\blacksquare \left\lfloor \frac{53}{2^0} \right\rfloor = 53 \equiv 1[2]$$

$$\blacksquare \left\lfloor \frac{53}{2^1} \right\rfloor = 26 \equiv 0[2]$$

$$\blacksquare \left\lfloor \frac{26}{2^2} \right\rfloor = 13 \equiv 1[2]$$

$$\blacksquare \left\lfloor \frac{13}{2^3} \right\rfloor = 6 \equiv 0[2]$$

$$\blacksquare \left\lfloor \frac{6}{2^4} \right\rfloor = 3 \equiv 1[2]$$

$$\blacksquare \left\lfloor \frac{3}{2^5} \right\rfloor = 1 \equiv 1[2]$$

■ On s'arrête car  $2^6 > 53$ . Du fait de la partie entière, toutes les prochaines divisions donneront 0.

D'où  $53 = 1 \times 2^5 + 1 \times 2^4 + 0 \times 2^3 + 1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0 = (110101)_2$

### Exercice 7 (Conversion en binaire).

Trouver la représentation binaire des dix entiers naturels ci-dessous :

$$\blacksquare 76$$

$$\blacksquare 188$$

$$\blacksquare 33$$

$$\blacksquare 109$$

$$\blacksquare 92$$

$$\blacksquare 211$$

$$\blacksquare 4$$

$$\blacksquare 238$$

$$\blacksquare 161$$

$$\blacksquare 126$$

Ici sont détaillées l'ensemble des opérations logiques et arithmétiques les plus communes qui peuvent être appliquées aux  $N$ -uplets de  $B^N$ . La bijection *btoi* sera considéré implicitement pour la suite, c'est-à-dire que toute fonction ou relation applicable sur  $B^N$  le sera aussi sur  $\llbracket 0; 2^N \rrbracket$ , en considérant un nombre par sa représentation binaire.

Plus formellement, pour toute fonction  $*$  :  $B^N \rightarrow E$ , on peut considérer de manière équivalente la fonction :

$$\begin{array}{ccc} *_{\mathbb{N}} & : & \llbracket 0; 2^N \rrbracket \rightarrow E \\ n & & \mapsto *(btoi^{-1}(n)) \end{array}$$

### 1.3.2 Nombres signés

Pour représenter des nombres *signés*, c'est-à-dire dont la valeur peut être négative comme positive (nombres qui possèdent un signe), on considère comme représentants de  $\frac{\mathbb{Z}}{2^N \mathbb{Z}}$  les nombres :  $(-2^{N-1}), \dots, -1, 0, 1, \dots, (2^{N-1}-1)$ .

Soit  $v \in B^N$  quelconque.

$$\begin{array}{ccc} btoi_s & : & B^N \rightarrow \llbracket 2^{N-1}; 2^N \rrbracket \\ b & & \mapsto \sum_{i=0}^{N-2} (b_i 2^i) - b_{N-1} 2^{N-1} \end{array}$$

**Remarque :** Si  $b_{N-1} = 0$ , alors  $btoi_u(v) = btoi_s(v)$ .

**Proposition 1.**  $(btoi_s(v)) = (btoi_u(w)) \Leftrightarrow v = w$

**Démonstration :**  $(btoi_s(v)) \equiv \sum_{i=0}^{N-2} (b_i 2^i) - b_{N-1} 2^{N-1} + b_{N-1} 2^N [2^N]$

Donc  $(btoi_s(v)) = (btoi_u(v))$ , or la représentation en base 2 d'un entier naturel est unique, donc  $v = w$ .

**Interprétation :** Quelque soit l'interprétation du mot binaire (signé ou non signé), la représentation binaire est unique. Ainsi, la proposition précédente est équivalente à  $\forall x, y \in \mathbb{Z}, \dot{x} = \dot{y} \Leftrightarrow btoi^{-1}(x) = btoi^{-1}(y)$

**Proposition 2.**  $b_{N-1} = 1 \Leftrightarrow btoi_s(v) < 0$ .

**Démonstration :**  $\sum_{i=0}^{N-2} (b_i 2^i)$  est majoré par  $\sum_{i=0}^{N-2} (2^i) = 2^{N-1} - 1 < b_{N-1} 2^{N-1}$

**Interprétation :** Il suffit de regarder le bit de poids fort d'un nombre pour connaître son signe.

**Proposition 3.**  $btoi_s(v) = -(btoi_s(\neg v) + 1)$

**Démonstration :**

$$\begin{aligned}
 -(btoi_s(\neg v) + 1) &= -\left(\sum_{i=0}^{N-2} ((1 - b_i) 2^i) - (1 - b_{N-1}) 2^{N-1} + 1\right) \\
 &= \sum_{i=0}^{N-2} ((b_i 2^i) - 2^i) + (1 - b_{N-1}) 2^{N-1} - 1 \\
 &= \sum_{i=0}^{N-1} ((b_i 2^i) - 2^{N-1} - b_{N-1} 2^{N-1}) \\
 &= btoi_s(v)
 \end{aligned}$$

### 1.3.3 Addition

On définit l'opération d'addition de deux  $N$ -uplets de  $\mathcal{B}^N$  comme l'opération :

$$\begin{aligned}
 + : \mathcal{B}^N \times \mathcal{B}^N &\rightarrow \mathcal{B}^N \\
 (x, y) &\mapsto btoi^{-1}(btoi_u(x) + btoi_u(y))
 \end{aligned}$$

**Interprétation :** Il s'agit en fait de l'addition des deux nombres en base 2 (comme on additionne des nombres en base 10, ou  $b$ ) dont on ne garde que les  $N$  premiers bits. Ainsi, si on effectue l'addition  $2^{N-1} + 2^{N-1} \notin \llbracket 0; 2^N \rrbracket$ , il faudrait un bit supplémentaire pour stocker l'information. Or ce bit n'est pas présent, il est donc simplement ignoré.

**Proposition 4 (Pseudo-linéarité).**  $\forall x, y \in \llbracket 0; 2^N \rrbracket, x + y \equiv btoi_u(btoi^{-1}(x) + btoi^{-1}(y)) [2^N]$

**Démonstration :**

$$\begin{aligned}
 btoi_u(btoi^{-1}(x) + btoi^{-1}(y)) &= btoi_u(btoi^{-1}(btoi_u(btoi^{-1}(x))) + btoi_u(btoi^{-1}(y))) \\
 &= (btoi_u(btoi^{-1}(x)) + btoi_u(btoi^{-1}(y))) \\
 &= (btoi_u(btoi^{-1}(x))) + (btoi_u(btoi^{-1}(y))) \\
 &= \dot{x} + \dot{y} \\
 &= (x + y)
 \end{aligned}$$

**Interprétation :** Additionner deux entiers naturels directement ou d'abord représenter en binaire ces entiers et ensuite additionner produit le même résultat modulo  $2^N$  par la représentation sur  $N$  bits.

**Exemples : (pour  $N = 8$ )**

- $(0011\ 0101)_2 + (0001\ 1000)_2 = (0100\ 1101)_2$   
c'est-à-dire  $53 + 24 = 77$
- $(1000\ 1000)_2 + (0000\ 1111)_2 = (1001\ 0111)_2$   
c'est-à-dire  $136 + 15 = 151$  en non signé et  $-120 + 15 = -105$  en signé
- $-(0101\ 1100)_2 = (1010\ 0100)_2$   
c'est-à-dire  $-92 = 164$  en non signé et  $-92 = -92$  en signé
- $(0111\ 1100)_2 + (0100\ 1001)_2 = (1100\ 0101)_2$   
c'est-à-dire  $124 + 73 = 197$  en non signé et  $124 + 73 = -59$  en signé
- $(1110\ 1110)_2 + (0010\ 1111)_2 = (1001\ 1101)_2$   
c'est-à-dire  $238 + 47 = 29$  en non signé et  $-18 + 47 = 29$  en signé

### 1.3.4 Soustraction

En utilisant la **Propriété 3.**, on définit simplement l'opération de soustraction par :

$$\begin{aligned} - & : \mathcal{B}^N \times \mathcal{B}^N \rightarrow \mathcal{B}^N \\ (x, y) & \mapsto x + (-y) = x + (\neg y + 1) \end{aligned}$$

### 1.3.5 Décalage arithmétique à droite

Au contraire du décalage logique à droite qui ne prend en considération aucune interprétation d'un mot binaire, le décalage arithmétique prend en considération le signe du nombre représenté par le mot binaire.

L'opération  $\gg_a$  est défini comme suit :

$$\begin{aligned} \gg_a & : \mathcal{B}^N \times \llbracket 0, N \rrbracket \rightarrow \mathcal{B}^N \\ ((a_{N-1} \dots a_0), i) & \mapsto \underbrace{a_{N-1} \dots a_{N-1}}_{i \text{ fois}} a_{N-1} \dots a_i \quad \text{si } i < N \\ & \mapsto a_{N-1} \dots a_{N-1} \quad \text{si } i = N \end{aligned}$$

$\gg_a$  conserve le signe, puisque le bit de poids fort reste constant (voir **Propriété 2.**).

**Proposition 5.**  $\forall v \in \mathcal{B}^N, btoi_u(v \gg_l 1) = \left\lfloor \frac{btoi_u(v)}{2} \right\rfloor$

**Démonstration :** Soit  $v = v_{N-1} \dots v_0 \in \mathcal{B}^N$ .

$$\begin{aligned} btoi_u(v \gg_l 1) &= btoi_u(0v_{N-1} \dots v_1) \\ &= \sum_{i=1}^{N-1} (v_i 2^{i-1}) + \underbrace{\left\lfloor \frac{v_0}{2} \right\rfloor}_{=0} \\ &= \left\lfloor \frac{btoi_u(v)}{2} \right\rfloor \quad \text{car } \forall (n, x) \in \mathbb{N} \times \mathbb{R}, n + \lfloor x \rfloor = \lfloor n + x \rfloor \end{aligned}$$

**Proposition 6.**  $\forall v \in \mathcal{B}^N, btoi_s(v \gg_a 1) = \left\lfloor \frac{btoi_s(v)}{2} \right\rfloor$

**Démonstration :** Soit  $v = v_{N-1} \dots v_0 \in \mathcal{B}^N$ .

$$\begin{aligned}
 btoi_s(v \gg_a 1) &= btoi_s(v_{N-1}v_{N-1} \dots v_1) \\
 &= \sum_{i=1}^{N-1} (v_i 2^{i-1}) - v_{N-1} 2^{N-1} + \underbrace{\left\lfloor \frac{v_0}{2} \right\rfloor}_{=0} \\
 &= \left\lfloor \sum_{i=0}^{N-1} (v_i 2^{i-1}) - v_{N-1} 2^{N-1} \right\rfloor \\
 &= \left\lfloor \sum_{i=0}^{N-2} (v_i 2^{i-1}) - v_{N-1} 2^{N-2} \right\rfloor \\
 &= \left\lfloor \frac{btoi_s(v)}{2} \right\rfloor
 \end{aligned}$$

**Remarque :**  $btoi_s(v \gg_a 1) = \left\lfloor \frac{btoi_u(v)}{2} \right\rfloor$  et  $btoi_u(v \gg_a 1) = \left\lfloor \frac{btoi_s(v)}{2} \right\rfloor$  ne sont vraies *a priori* que pour des entiers naturels. En effet, le bit de signe ne doit pas être conservé en représentation non signée alors qu'il doit l'être en représentation signée.



## OPÉRATIONS FLOTTANTES



Les résultats qui vont suivre sont beaucoup plus spécifiques à une partie mineure de l'informatique. Par manque de temps pour la rédaction, ils n'ont pas pu être particulièrement développés. Cela sera fait dans une version future. Veuillez excuser d'avance pour le manque de rigueur et de détails...<sup>6</sup>

### 1.4.1 Représentation des nombres à virgules

La manipulation de valeurs réelles est extrêmement importante, et même nécessaire, pour la manipulation d'espaces continues. On peut en particulier penser à des simulations physiques.

Pourtant, on peut observer simplement qu'un ordinateur manipule des données finies. En ce sens, il n'est envisageable de ne manipuler que des sous-ensembles de  $\mathbb{Q}$ . Par exemple :

- $\pi$  possède un nombre infini de chiffres dont l'information ne peut être stockée par une formule calculable<sup>7</sup>
- $\frac{1}{3}$  possède un nombre infini de chiffres après la virgule. Il suffit de stocker la partie entière 0 ainsi que la partie répétée des décimales, c'est-à-dire 3 (avec l'information de la répétition)
- 1.414 possède un nombre fini de chiffres après la virgule. Il suffit alors de stocker deux entiers : celui avant la virgule, et celui après.

Dépendamment du type de nombre manipulé (avec un nombre infini de chiffre après la virgule ou non), le stockage des informations contenues par ce nombre diffère (et est parfois impossible comme dans le cas de  $\pi$ ).

6. On rate en particulier le lien extraordinaire entre interprétation entière et interprétation flottante par la norme *IEEE 754* des mots binaires qui fournit une approximation du logarithme, justifie ainsi le maintien de la relation d'ordre par changement d'interprétation et donc également la raison d'être de cette norme.

7. En informatique, une fonction est dite *calculable* si il existe un algorithme qui pour tout  $x$ , détermine  $f(x)$  en temps fini.

**Remarque 1 :** pour qu'un programme informatique manipulant des nombres à virgules puisse être exécuté sur plusieurs machines différentes, il faut que toutes ces machines aient la même représentation des nombres à virgules. Une norme est donc nécessaire.

**Remarque 2 :** Les processeurs d'ordinateur classiques<sup>8</sup> manipulent des mots binaires de taille fixe  $N \in \{8, 16, 32, 64, 80, 128\}$

**Remarque 3 :** Un mot de taille  $N = 16$  ne peut représenter que quelques milliers de valeurs différentes ( $2^{16} = 65536$ ) ce qui est ridicule pour représenter dans la pratique des nombres à virgules un tant soit peu différents.

**Remarque 4 :** Imaginons qu'on choisisse de représenter *naïvement* des nombres à virgules sur  $N = 32$  bits en stockant sur 16 bits la partie entière et la partie flottante. Cela limite particulièrement la précision des nombres représentés puisqu'il ne peut y avoir que 65536 parties fractionnaires différentes.

Une norme est donc née de l'*Institute of Electrical and Electronics Engineers (IEEE)*, nommée la norme *IEEE 754*. Il s'agit de la norme la plus largement utilisée en informatique, qui se base sur la notation scientifique.

### 1.4.2 Écriture en base 2 de nombres décimaux

Avant de décrire la norme *IEEE 754*, on commence par détailler l'écriture en base 2 d'un nombre réel. On rappelle qu'un nombre  $x \in \mathbb{R}$  s'écrit de manière générique en base 10 sous la forme suivante :

$$x = \lfloor x \rfloor + \text{fract}(x)$$

La partie entière de  $x$  s'écrit en base 10 :

$$\lfloor x \rfloor = c_{n-1}10^{n-1} + c_{n-2}10^{n-2} + \dots + c_010^0, \text{ où } n = \lceil \log_{10}(\lfloor x \rfloor) \rceil$$

Tandis que la partie fractionnaire de  $x$  s'écrit en base 10 :

$$\text{fract}(x) = c_{-1}10^{-1} + c_{-2}10^{-2} + \dots$$

avec pour tout  $i \in \mathbb{Z}$ ,  $c_i \in \llbracket 0; 9 \rrbracket$

En changeant de base, et en choisissant la base 2, on a de manière équivalente :

$$\begin{cases} \lfloor x \rfloor &= c_{n-1}2^{n-1} + c_{n-2}2^{n-2} + \dots + c_02^0, \text{ où } n = \lceil \log_2(\lfloor x \rfloor) \rceil \\ \text{fract}(x) &= c_{-1}2^{-1} + c_{-2}2^{-2} + \dots \end{cases}$$

avec pour tout  $i \in \mathbb{Z}$ ,  $c_i \in \llbracket 0; 1 \rrbracket$

On relève au passage la propriété qui permet de calculer la représentation en base  $b$  d'un nombre réel  $x$  :

**Proposition 7.**  $\forall n \in \mathbb{N}^*, \lfloor b^n \text{fract}(x) \rfloor \equiv c_{(-n)}[b]$ , où  $c_{(-i)}$  est le  $i^{\text{e}}$  après la virgule dans l'écriture de  $x$  en base  $b$ .

**Exemple :** On peut appliquer récursivement cette propriété sur  $x = 21.125 = (10101)_2 + 0.125$  :

- $c_{-1} = \lfloor 2^1 \times 0.125 \rfloor = \lfloor 0.25 \rfloor = 0$
- $c_{-2} = \lfloor 2^2 \times 0.125 \rfloor = \lfloor 0.5 \rfloor = 0$
- $c_{-3} = \lfloor 2^3 \times 0.125 \rfloor = \lfloor 1 \rfloor = 1$

---

8. Comme les processeurs *Intel x86*



- $\forall i < -3, c_{(-i)} = 0$

d'où :  $x = (10101.001)_2$

On peut ainsi calculer la notation scientifique en base  $b = 2$  de tout réel  $x$  (voir section suivante)

**Exercice 8 (Écriture en base 2 de nombres non entiers).** Écrire en base 2 les nombres suivants :

- 14.5625
- 1.40625
- 7.4375
- 0.1 : Que remarque-t-on ? Exhiber un exemple d'un tel phénomène en écriture décimale.

### 1.4.3 Norme *IEEE 754*

Ne seront décrits ici que les nombres à virgules suivant la norme *IEEE 754* d'une taille  $N = 32$ . En effet, le principe est le même pour les autres tailles si ce n'est que certaines constantes diffèrent.

#### Rappel de vocabulaire en notation scientifique

Un nombre  $x \in \mathbb{R}$  en notation scientifique est écrit selon le format suivant :

$$x = \pm \text{mantisse} \times \text{base}^{\text{exposant}}$$

avec  $\text{mantisse} \in [1, \text{base}[$  et  $\text{exposant} \in \mathbb{Z}$

Par exemple :

$$+5.56 \times 7^{-12}$$

Ici :

- + est le *signe* du nombre
- 5.56 est la *mantisse* du nombre
- -12 est l'*exposant* du nombre
- 7 est la *base de représentation* du nombre

Dans toute la suite, la base de représentation sera fixe et vaudra 2. Il n'y aura donc pas besoin de la stocker explicitement dans le mot binaire.

Il reste donc trois éléments, qui seront chacun stockés sur un certain nombre de bits.

Selon la norme *IEEE 754*, sur 32 bits :

- le signe  $s$  : 1 bit
- la mantisse  $m$  : 23 bits
- l'exposant  $e$  : 8 bits

Visuellement :

$$\underbrace{0}_s \underbrace{00000000}_e \underbrace{000000000000000000000000}_m$$

#### Calcul du signe

Si  $x < 0$ ,  $s = 1$ , et  $s = 0$  sinon.

### Calcul de la mantisse

Soit  $x \in \mathbb{R}$ .

En base 2, il existe  $(c_i)_{i \in \mathbb{Z}} \in \{0, 1\}^{\mathbb{Z}}$  une suite unique telle que  $x = \pm \sum_{i=-\infty}^N (c_i 2^i)$  où  $N$  est minimal.

On a :

$$x = \pm 2^N \sum_{i=-\infty}^N (c_i 2^{i-N}) = \pm m \times 2^N \text{ où } 1 \leq m < 2$$

Il s'agit très exactement de l'écriture de  $x$  en notation scientifique en base 2 et on peut calculer les  $c_i$  grâce à l'algorithme déductible de la propriété de la section précédente.

Par ailleurs, on sait que  $m \geq 1$ , il n'y a donc pas besoin de stocker cette information. On ne conserve pour la mantisse dans la représentation en norme *IEEE 754* que les bits après la virgule.

### Calcul de l'exposant

L'exposant peut être positif ou négatif. Cependant, on utilise pas ici la technique du complément à deux. En effet, cela rendrait plus complexe la comparaison entre deux nombres à virgules représentés selon cette norme.<sup>9</sup> À la place, on utilise un *biais*.

Si l'exposant est écrit sur  $n$  bits, la valeur du biais est  $2^{n-1} - 1$ . Cette valeur est donc constante une fois la norme fixée (puisqu'on fixe le nombre de bits sur lesquels sera écrit l'exposant). On ajoute ce biais à la valeur  $N$  trouvée lors du calcul de la mantisse. Ainsi, pour  $N \in \llbracket -2^{n-1} - 1; 2^{n-1} - 1 \rrbracket$ , on a  $N + \text{biais} \geq 0$  et donc représentable par un mot binaire interprété comme non signé.

**Remarque :** On interdit  $N = 2^{n-1}$  (pour pouvoir représenter les valeurs spéciales  $\pm\infty$  et  $NaN$ )

### Représentation finale

On distingue alors deux catégories selon la valeur de  $N + \text{biais}$  pour le choix de la mantisse :

- $N + \text{biais} = 0$  : le nombre est dit “*dénormalisé*”
- $N + \text{biais} \in \llbracket 1; 2^n - 2 \rrbracket$  : le nombre est dit “*normalisé*”

La normalisation du nombre affecte en partie la valeur de la mantisse et de l'exposant encodé :

**Nombre normalisé :** on garde pour la mantisse les chiffres  $c_{N-1} \dots c_{N-23}$ .

**Nombre dénormalisé :** on effectue un décalage supplémentaire dans l'écriture de  $x$  pour l'écrire sous la forme  $x = (2^{-1}m) \times 2^{N+1}$

Une fois qu'on a déterminé la représentation binaire de  $N + \text{biais}$  et les chiffres à conserver de la mantisse, il suffit de “coller” le bit de signe, la représentation binaire de  $N + \text{biais}$  et les chiffres conservés de la mantisse pour obtenir la représentation binaire finale du nombre.

**Exemple (normalisé) :** Représentons le nombre  $x = 21.125 = (10101.001)_2$  selon la norme *IEEE 754* sur 32 bits.

$x = (1.0101001)_2 \times 2^4$ , donc  $N = 4$ . Comme on est sur 32 bits, la norme *IEEE 754* spécifie que l'exposant est codé sur 8 bits. D'où  $\text{biais} = 2^7 - 1 = 127$ . Alors  $N + \text{biais} = (\text{biais} + 1) + (N - 1) = 2^7 + 2^1 + 2^0 = (10000011)_2 > 0$  donc le nombre écrit sous forme normalisé. On a donc :

- signe : 0 car  $21.125 \geq 0$

9. La démonstration ne sera pas donnée mais la relation d'ordre sur les nombres réels représentés par les mots binaires est la même que celle sur les mots binaires représentant les nombres réels. C'est-à-dire qu'on peut indifféremment comparer représentants ou représentés.

- le nombre est normalisé donc  $e = (10000011)_2$
- le nombre est normalisé donc  $m = (010100100000000000000000)_2$

Finalement,  $x = 21.125$  est représenté selon la norme *IEEE 754* par le mot binaire 32 bits :

01000001101010010000000000000000

**Exemple (dénormalisé) :** Représentons le nombre  $x = -9.1688559 \times 10^{-39}$  selon la norme *IEEE 754* sur 32 bits.

Il existe  $p \in \mathbb{Z}$  tel que  $2^p \leq x < 2^{p+1}$ , c'est-à-dire  $x = m \times 2^{\lfloor \log_2(x) \rfloor}$ .

On cherche alors  $m = x \times 2^{-\lfloor \log_2(x) \rfloor} = -9.1688559 \times 10^{-39} 2^{126} \approx 0.78 \approx (1.10001111010111000010100)_2 \times 2^{-1}$

On veut donc écrire selon la norme *IEEE 754* le nombre  $x \approx 1.10001111010111000010100 \times 2^{-127}$

Sur 32 bits, *biases* = 127, donc  $N + \text{biases} = 0$ . Le nombre est donc dénormalisé<sup>10</sup>. On code alors :

- signe : 1 car  $x < 0$
- le nombre est dénormalisé donc  $e = (00000000)_2$
- le nombre est dénormalisé donc  $m = (11000111101011100001010)_2$

Finalement,  $x = -9.1688559 \times 10^{-39}$  est représenté selon la norme *IEEE 754* par le mot binaire 32 bits :

10000000011000111101011100001010

#### 1.4.4 Notation pour la suite du cours

On notera pour la suite  $\mathbb{R}_{f32}$  (resp.  $\mathbb{R}_{f64}$  et  $\mathbb{R}_{f128}$ ) est les nombres réels dont il existe une représentation binaire sur 32 bits (resp. 64 et 128 bits) exacte selon la norme *IEEE 754*.



## REPRÉSENTATION HEXADÉCIMALE



La représentation hexadécimale n'est qu'une compression de l'écriture de mots binaires en base 16 dans l'unique objectif de gagner de la place et de la lisibilité lors de l'écriture de mots binaires par des humains.

Le choix de la base s'explique par l'observation suivante :  $16 = 2^4$ , donc les seize chiffres peuvent représenter les nombres  $0, \dots, 15$ , c'est-à-dire en binaire  $0000, \dots, 1111$ . On pourrait choisir de manière équivalente n'importe quelle base qui soit une puissance de 2.<sup>11</sup> En effet, écrire en base 2 revient à décomposer selon les puissances de 2. En choisissant une puissance de 2 comme base d'écriture, on divise d'autant le nombre de chiffres requis.

**Chiffres en base 16 :** Les chiffres en base seize sont les suivants :

$$\Sigma = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F\}$$

Ainsi,  $(A)_{16} = 10$ ,  $(B)_{16} = 11$ ,  $(C)_{16} = 12$ ,  $(D)_{16} = 13$ ,  $(E)_{16} = 14$  et  $(F)_{16} = 15$ .

<sup>10</sup>. Comme par hasard...

<sup>11</sup>. En particulier la base octale (c'est-à-dire la base 8) a été utilisé pendant longtemps bien que maintenant assez obsolète.

Soit  $n = (c_{N-1} \dots c_0)_{16} \in \mathbb{N}$ , où  $\forall i \in \llbracket 0; N-1 \rrbracket, c_i \in \Sigma$

$$n = \sum_{i=0}^{N-1} (c_i 16^i)$$

Par exemple,  $(2FA)_{16} = 2 \times 16^2 + 15 \times 16^1 + 10 \times 16^0 = 762$ .

**Notation :** On note aussi  $n = 0xc_{N-1} \dots c_0$ . Par exemple,  $0x2FA = (2FA)_{16} = 762$

### Lien entre binaire et hexadécimale

Soit  $i \in \llbracket 0; N-1 \rrbracket$ . On observe que  $c_i \in \llbracket 0; 15 \rrbracket$ . C'est-à-dire que chaque  $c_i$  peut s'écrire avec 4 bits. On note alors  $c_i = (b_{i,3}, b_{i,2}, b_{i,1}, b_{i,0})_2$ .

$$\begin{aligned} \sum_{i=0}^{N-1} (c_i 16^i) &= \sum_{i=0}^{N-1} (c_i 2^{4i}) \\ &= \sum_{i=0}^{N-1} ((b_{i,3}2^3 + b_{i,2}2^2 + b_{i,1}2^1 + b_{i,0}2^0) \times 2^{4i}) \\ &= \sum_{i=0}^{N-1} (b_{i,3}2^{3+4i} + b_{i,2}2^{2+4i} + b_{i,1}2^{1+4i} + b_{i,0}2^{4i}) \\ &= (b_{N-1,3}b_{N-1,2}b_{N-1,1}b_{N-1,0} \dots b_{0,3}b_{0,2}b_{0,1}b_{0,0})_2 \end{aligned}$$

**Interprétation :** Pour passer d'une écriture binaire à une écriture hexadécimale, il suffit d'écrire chaque mot<sup>12</sup> de quatre bits en un chiffre hexadécimale. Inversement, pour passer d'une écriture hexadécimale à une écriture binaire, il suffit de convertir chaque chiffre hexadécimale en un groupe de quatre bits.

**Exemple :**  $0x2FA = \underbrace{0010}_2 \underbrace{1111}_F \underbrace{1010}_A$

**Remarque :** Cette transformation est aussi valable pour des nombres flottants, puisqu'il suffit de considérer l'interprétation entière du mot binaire qui par la norme *IEEE 754* représente un flottant.

### Exercice 9 (Conversion binaire-hexadécimale).

Trouver les représentations binaires puis hexadécimale des nombres suivants. On utilisera l'écriture en base 2 pour les nombres entiers et la représentation 32 bits de la norme *IEEE 754* pour les nombres à virgules :

- 713705
- 8.8
- 42
- -1.1
- 101

<sup>12</sup>. ou *mot*



# ALGORITHMES ET PROGRAMMATION

La définition rigoureuse d'un algorithme se trouve n'être absolument pas trivial, ni même simple. Une première raison à cela est qu'il s'agit d'un concept au départ considéré comme intuitif, et que la formalisation d'un concept qui semble humainement intuitif pose immédiatement la question du bien fondé et de la rigueur de cette formalisation : un mot est toujours interprété, et deux humains peuvent utiliser le même mot, et ne pas en avoir la même utilisation, voir la même compréhension fine.

La difficulté sous-jacente à une telle définition sera tout à fait esquivé ici puisqu'elle nécessiterait de poser une théorie du calcul pour définir ce qu'est un problème, ce que signifie la résolution d'un problème, ainsi que la résolution en temps fini d'un problème, ce qu'est une instruction élémentaire, etc...

On s'en passera en conservant une définition « avec les mains » qui manque cruellement de rigueur mais qui a le bénéfice d'être simple et suffisante dans un cadre purement pratique.

## Définition 1 : Algorithme

Un algorithme est une suite d'actions/d'instructions précises dont l'objectif est de résoudre un problème donné. Ces instructions doivent être les plus élémentaires possibles, et ne surtout pas être ambiguës. C'est-à-dire qu'en lisant l'algorithme, il n'y ait qu'une seule possibilité d'action à chaque étape (ce qui ne signifie pas que le résultat soit prédéterminé puisqu'une action peut potentiellement avoir un résultat aléatoire, comme le lancer d'un dé par exemple).

**Exemple :** "Changer la valeur d'un nombre" n'est pas une instruction possible pour un algorithme, puisque le lecteur peut toujours *interpréter* l'instruction, et ne sait en fait toujours pas exactement ce qu'il doit faire. Par contre l'instruction "Ajouter 1 à la valeur de la variable  $x$ " est une instruction, à condition que  $x$  soit connue, que 1 soit bien définie et que "Ajouter" soit une opération binaire bien définie sur 1 et  $x$ .

Tout simplement.

---

### Algorithme 1 : Premier exemple simple

---

```
1 Afficher("Bonjour!");  
2 Afficher("Au revoir!");
```

---

Cette algorithme n'est valable que si l'opération "Afficher" a été décrite de manière super précise avant. C'est-à-dire que même l'endroit où il faut afficher (par exemple, la feuille sur laquelle est écrite ce chapitre d'algorithmique) est indiqué auparavant au lecteur, *pour éviter l'interprétation*. La donnée d'une chaîne de caractère doit également être connue.

Un algorithme peut ensuite être exécuté par un *agent*, et il produira alors un résultat donné.

## Définition 2 : Agent algorithmique

Actionneur capable de communication qui lira l'algorithme écrit et effectuera les actions les unes à la suite des autres. Ce peut être un être humain, ou encore un chat ou un chien (par exemple en apprenant l'algorithme à haute voix au chien)

L'**Algorithme 1** produira donc le résultat suivant :

Bonjour !  
Au revoir !

L'intérêt d'algorithmes est notamment d'automatiser des tâches. On peut notamment imaginer l'algorithme suivant :

---

**Algorithme 2 :** Aller chercher le journal

---

```

1 Aller à la boîte aux lettres;
2 si la boîte est ouverte alors
3   | Ramasser le contenu de la boîte aux lettres;
4   | Fermer la boîte aux lettres;
5 sinon
6   | Ouvrir la boîte aux lettres;
7   | Ramasser le contenu de la boîte aux lettres;
8   | Fermer la boîte aux lettres;
9 Revenir à la maison;
```

---

On peut imaginer apprendre cet algorithme à notre corbeau de compagnie (c'est très intelligent les corbeaux...), et simplement lui ordonner d'exécuter cet algorithme par l'instruction "Va chercher le journal!".

**Remarque :** En petit malin, le lecteur attentif peut observer deux choses :

- que l'*agent* qui exécute cet algorithme doit déjà savoir de quelle boîte aux lettres et de quelle maison il est question, pour qu'il n'y ait pas d'interprétation (et pour aller plus loin, il faut aussi qu'il comprenne le langage dans lequel est exprimé l'algorithme et le sens de chaque verbe/action).
- que l'instruction "Fermer la boîte aux lettres" est présente deux fois, et surtout qu'on est certain qu'elle s'exécute. En effet, cette instruction est présente que la condition "*la boîte est ouverte*" soit vérifiée ou non! *idem* pour l'instruction "Ramasser le journal". On peut donc les mettre une seule fois à la fin.

---

**Algorithme 3 :** Aller chercher le journal

---

```

1 Aller à la boîte aux lettres;
2 si la boîte est ouverte alors
3   | ;                                     /* Il n'y a rien à faire ! */
4 sinon
5   | Ouvrir la boîte aux lettres;
6 Ramasser le contenu de la boîte aux lettres;
7 Fermer la boîte aux lettres;
8 Revenir à la maison;
```

---

L'algorithme est alors plus court à lire, mais aussi plus court à apprendre, à enregistrer. Si il faut le stocker sur un ordinateur par exemple, la quantité de mémoire de l'ordinateur n'est pas infinie! Il est donc **parfois**<sup>13</sup> important que cet algorithme soit le plus court possible. C'est ce qu'on appellera de l'*optimisation d'espace*<sup>14</sup>.

---

13. Mais pas systématiquement, cela dépend de si l'on cherche à optimiser la vitesse d'exécution ou la taille en mémoire

14. Dual de l'*optimisation temporelle* qui cherche à optimiser la vitesse d'exécution. Il n'est parfois pas possible d'avoir les deux, détail sur lequel on reviendra dans très très très longtemps

### 1.6.1 Algorithmes et programmes

#### Définition 3 : Programme

Un programme informatique est un algorithme pouvant être exécuté par un ordinateur (qui est donc l'agent exécutant). Il doit donc être écrit dans un langage compréhensible par l'ordinateur appelé *langage de programmation*.

Dans la suite du chapitre, l'entièreté des algorithmes seront exécutés sur ordinateur, pour la principale raison que cela permet de visualiser les résultats et d'expérimenter par soi-même.

On va donc considérer pour toute la suite des programmes. Il faut cependant garder à l'esprit la distinction entre les deux, puisqu'un algorithme englobe beaucoup plus de choses qu'un programme informatique (il est difficile de programmer un ordinateur pour aller chercher le journal dans la boîte aux lettres).

Pour qu'un programme puisse être exécuté par un ordinateur, il doit être connu de celui-ci, et est donc stocké dans la mémoire de l'ordinateur. L'ordinateur va ensuite le lire instructions par instructions et exécuter chacune de ces instructions.

La représentation de l'algorithme doit donc être accessible à un ordinateur. On peut considérer une écriture de cette algorithme en langage binaire, c'est-à-dire comme une suite finie de 0 et de 1. La traduction du langage naturel (c'est-à-dire humain) en langage binaire est appelée *compilation*. Elle est effectuée par des programmes appelées des *compilateurs*.

On peut cependant généraliser un peu la notion de *compilation* : on peut dire que compiler c'est lire une suite de caractères obéissant à une certaine syntaxe, en construisant une (autre) représentation de l'information que ces caractères expriment. De ce point de vue, beaucoup d'opérations apparaissent comme étant de la compilation ; à la limite, la lecture d'un nombre est déjà de la compilation, puisqu'il s'agit de lire des caractères constituant l'écriture d'une valeur selon la syntaxe des nombres décimaux et de fabriquer une autre représentation de la même information, à savoir sa valeur numérique « dans notre tête »

## CHAPITRE

# 2

# PROGRAMMER UN ORDINATEUR



## UN PEU DE VOCABULAIRE



Cette table de vocabulaire est donné en préambule pour ne pas trop surcharger le texte ci-après.

### Définition 4 : BIOS

Le BIOS (*B*asic *I*ntput *O*utput *S*ystem) est le programme qui s'exécute au démarrage de l'ordinateur (appelé le *boot*). C'est lui qui permet de le lancer véritablement pour être utilisé. En effet, son objectif est de "passer la main" au système d'exploitation, grâce à un petit programme appelé le *bootloader* (*loader* signifie chargeur en anglais).

### Définition 5 : Système d'exploitation

Un système d'exploitation est un programme exécuté par le BIOS juste après le lancement. Ce programme fournit à l'utilisateur de l'ordinateur toutes les fonctionnalités dont il pourrait avoir besoin : accès au disque dur, accès à la mémoire vive, aux périphériques entrée/sortie, possibilité d'exécuter des programmes utilisateurs, générateur de nombres pseudo-aléatoire, etc...

### Définition 6 : Répertoire

Un répertoire est le nom technique donné au dossier. Il contient d'autres répertoires ou fichiers. Il permet en effet de *répertorier* des fichiers ou d'autres répertoires. Sa fonction principale est donc la classification.



**Définition 7 : Fichier**

Un fichier est un ensemble d'informations numériques constituées d'une séquence d'octets. Ces informations peuvent représenter des données allant du programme informatique à la vidéo en passant par les informations d'une communication réseaux, un livre numérique, la mémoire d'un programme informatique, etc. . . Ces informations sont réunies sous un même nom et manipulées comme une unité, appelé le fichier.

**Définition 8 : Extension de nom fichier**

Une extension de nom de fichier (ou plus simplement "extension de fichier") est une suite de caractères à la fin du nom du fichier, séparée du radical du nom de fichier par un point, qui indique à l'utilisateur de l'ordinateur et aux logiciels le format des données qu'il contient. L'extension agit uniquement à titre *indicatif* ! La modifier sans modifier le fichier lui-même n'effectue **STRICTEMENT AUCUNE** conversion. Il ne suffit pas d'appeler un chat un chien pour que celui-ci se transforme subitement ! Le nom d'un fichier est seulement une étiquette. Il n'a aucune incidence sur ce que le fichier contient réellement.<sup>a</sup>

<sup>a</sup>. J'insiste car il s'agit là d'une  *croyance*  extrêmement répandue chez les utilisateurs non techniciens d'outils informatiques

**Définition 9 : Système de fichiers**

Afin de faciliter la localisation et la gestion des répertoires et des fichiers, ceux-ci sont organisés suivant un système de fichiers. C'est ce système qui permet à l'utilisateur de répartir les fichiers dans une arborescence de répertoires et de les localiser par un chemin d'accès.

**Définition 10 : Répertoire de travail**

Le répertoire de travail d'un exécutable est le répertoire dans lequel l'exécutable va effectuer ses instructions. Par exemple, si l'exécutable contient une instruction qui ouvre et lit un fichier sur le disque dur, l'ordinateur essaiera d'ouvrir ce fichier dans le répertoire de travail, renverra une erreur si ce fichier n'est pas présent dans le répertoire de travail. Il est possible de modifier le répertoire de travail d'un exécutable à l'intérieur de celui-ci (par certaines instructions).

**Définition 11 : Chemin d'accès**

Chaîne de caractères qui décrit la position du fichier sur son support de stockage au sein du système de fichiers. On distingue deux types de chemin d'accès :

- chemin absolu : décrit la position absolue du fichier depuis la racine de l'arborescence du système de fichiers (/ sous Linux, C : / sous Windows)
- chemin relatif : décrit la position relative du fichier par rapport au répertoire de travail. On considère alors le répertoire de travail comme la racine de l'arborescence

**Définition 12 : Instruction**

Une instruction est une séquence d'octets qui peut être lue par l'UCC pour exécuter une action.

**Définition 13 : Exécutable**

Un fichier exécutable (ou plus simplement un exécutable) est un fichier contenant un entête d'informations générales et une séquence d'instructions. L'entête peut par exemple définir le point d'entrée du programme, c'est-à-dire l'adresse (relative) dans la séquence d'instructions à laquelle l'ordinateur doit débiter sa lecture des instructions.

**Définition 14 : Niveau d'abstraction**

Les langages de programmation peuvent être plus abstraits, c'est-à-dire être proche ou non du fonctionnement technique de l'ordinateur. Un langage à haut niveau d'abstraction va cacher la technique associée à la manipulation du matériel de l'ordinateur (mémoire vive, périphériques, etc. . .) tandis qu'un langage à bas niveau d'abstraction va laisser la possibilité au programmeur de manipuler lui-même ce qui est "matériel". En fait, les langages à haut niveau d'abstraction écrivent les instructions de manipulation bas niveau à l'insu du programmeur.



### 2.2.1 Objectifs des langages de programmation

Les langages dits « de programmation » ont un objectif : la description de données.

Ces données peuvent être de multiples natures. Ce peut être :

- un programme
- une page internet
- une base de relations entre données
- un fichier PDF comme celui-ci
- de la musique
- etc. . .

Il est à noter par ailleurs qu'un programme est potentiellement capable de décrire lui-même n'importe quel type d'information. Cependant, certains langages de programmation décrivent directement les données et non pas le programme qui les génère. On appelle les langages de programmation décrivant les données directement des *langages de description*. Les langages décrivant des instructions exécutables par un ordinateur sont appelés des *langages impératifs*.

Parmi les langages de description les plus importants, on peut citer :

- HTML : description de pages internet
- CSS : description de l'apparence d'éléments (associé presque directement au HTML)
- LaTeX : description de fichiers PDF, que ce soient diaporamas, articles de recherche, livres, etc. . .
- les langages HDL (VHDL, SystemVerilog, . . .) : description de circuits électroniques intégrés
- Z : description de la spécification formelle de systèmes.<sup>1</sup>

Parmi les langages impératifs les plus importants, on peut citer :

- Scratch : langage à très haut niveau d'abstraction spécialisé pour l'éducation, c'est-à-dire destiné à l'apprentissage de l'algorithmie pour les enfants

1. Certains affirmeraient que comme personne ne le connaît, il est sans importance. . . je le mets pour attirer l'attention sur un langage atypique particulièrement intéressant et absolument pas dépourvu d'utilité.

- Python : langage généraliste à très haut niveau d'abstraction destiné au prototypage rapide d'applications
- Java : langage généraliste à très haut niveau d'abstraction, très utilisé dans le développement d'applications
- JavaScript : langage à très haut niveau d'abstraction permettant de dynamiser des pages internet. Sert au développement d'applications Webs. Il manipule et génère des pages internet sous forme de texte HTML.
- R : langage à très haut niveau d'abstraction spécialisé pour le calcul statistique et les visualisations de données
- PHP : langage à très haut niveau d'abstraction spécialisé pour la programmation de serveurs Web
- C : langage généraliste à niveau relativement bas d'abstraction (par rapport à la plupart des autres, mais très haut tout de même par rapport à l'Assembleur). Il est utilisé notamment dans le développement de programmes dans des systèmes embarqués, ou pour tout ce qui se rapproche de l'ordinateur, comme les systèmes d'exploitation par exemple.
- C++ : une extension du langage C qui introduit des concepts de plus haut niveau d'abstraction tout en conservant le rapprochement technique avec l'ordinateur. Ce langage est le plus utilisé dans le développement d'applications et de jeux-vidéos.
- C# (prononcer *C sharp*) : langage généraliste de haut niveau utilisé pour le développement d'applications et de jeux-vidéos. Le système d'exploitation Windows est programmé en grande partie avec ce langage.
- Rust : langage généraliste associant bas niveau et haut niveau. Nécessite une certaine compétence en matière de programmation.
- Assembleur : le langage généraliste du plus bas niveau d'abstraction accessible par un humain. Il s'agit d'une traduction presque littérale des actions effectués par un ordinateur. Nécessite une certaine compétence en matière de programmation.

### 2.2.2 Langages compilés et interprétés

Les langages de programmation peuvent être à nouveau divisés en deux catégories distinctes :

- les langages compilés
- les langages interprétés

La compilation consiste à traduire un texte écrit dans un langage de programmation sous une forme accessible par un ordinateur, c'est-à-dire en langage binaire généralement.

Certains langages de programmation nécessitent d'être compilés pour que la donnée décrite puisse être traitée par l'ordinateur. Par exemple, la description en LaTeX d'un document nécessite une compilation par un autre programme pour être codée au format PDF. Les langages C, C++ et Assembleur sont également des langages qui nécessitent un *compilateur* pour être transformés/traduits en code binaire qui puisse être exécuté par un ordinateur. Ces langages de programmation sont en général plus anciens que les langages dits "interprétés"

L'autre catégorie de langages, dits interprétés, n'est jamais traduite en langage binaire pour être exécuté par l'ordinateur directement. Il y a à la place une interface créée par un autre programme appelé *interpréteur*. Ce programme, qui lui est exécuté par l'ordinateur, va simuler l'exécution de l'ordinateur sur le code. Il va lire chacune des lignes du programme écrit, et va faire exécuter les instructions correspondantes à l'ordinateur. Cela permet notamment de créer un niveau d'abstraction supplémentaire pour le programmeur, qui n'a pas besoin de connaître sa machine pour écrire des programmes. C'est l'interpréteur qui s'occupe de l'aspect le plus technique. Le désavantage majeur de ce type de langages interprétés est que l'exécution d'un programme est beaucoup plus lente puisqu'une

étape de traduction “en direct” par l’interpréteur est nécessaire **à chaque exécution du programme**. Ce type de langages n’a pu se développer que lorsque les ordinateurs ont été assez puissants pour le permettre. Cela explique par exemple l’explosion de Python dans le monde du développement ces dix dernières années.<sup>2</sup>



## LE LANGAGE C

Le langage C est un langage de bas niveau. Il permet donc de mieux comprendre le fonctionnement interne de l’ordinateur puisqu’il permet la manipulation de la mémoire. D’un autre côté, il traduit assez bien dans sa syntaxe l’intuition algorithmique humaine et se trouve donc un moyen efficace d’apprendre à programmer. Il est de plus très simple et présente peu de technicité langagière pure (au contraire de langages comme le Rust ou le Java par exemple). En cela, l’apprentissage du C permet de basculer ensuite vers un très grand nombre d’autres langages de programmation.

Le C a été inventé dans les années 70 dans l’objectif particulier de développer des systèmes d’exploitation. En effet, son rapprochement avec la machine, son extrême modularité, sa syntaxe claire et précise et la possibilité d’inclure des fragments de programmes écrits en assembleur à l’intérieur de programmes écrits en C permettent de développer n’importe quel type d’application complexe avec une certaine facilité pour l’époque. Le C reste très utilisé dans les systèmes embarqués et le développement de systèmes d’exploitation. Pour tout un tas de raisons allant du manque de souplesse abstraite du langage à certaines failles de sécurité introduites par les programmeurs rêveurs<sup>3</sup>, le langage C a tendance à être remplacé pour le développement d’applications complexes par d’autres langages qui proposent certaines solutions abstraites<sup>4</sup>, tiennent plus la main au programmeur en terme de sécurité et de stabilité de développement<sup>5</sup>, proposent plus de fonctionnalités pré-écrites<sup>6</sup>, etc. . .

### 2.3.1 Installer un éditeur et un compilateur

Avant de pouvoir programmer en C, il est nécessaire d’installer certains outils de base.

#### Éditeur de texte

Le premier est l’éditeur de texte incluant la coloration syntaxique (*syntax highlighting* en anglais), nécessaire pour la programmation quelque soit le langage utilisé. En effet, la coloration syntaxique permet de reconnaître les éléments du langage facilement et rend le code beaucoup plus lisible.

#### Un premier programme

---

2. Avis personnel de l’auteur : utiliser des langages qui consomment beaucoup plus de ressources pour arriver au même résultat avec des performances moindres pour des seules raisons de facilité est un problème dans un monde qui a besoin d’une baisse drastique de la consommation énergétique pour sa survie. Le Python ne permet pas de comprendre en profondeur les choses ni d’optimiser les programmes écrits de manière réellement efficace. Il n’est donc pas adapté pour des projets de grande envergure mais reste parfois utile pour tester une idée en cinq minutes. Certains argueront que le langage est utilisé à profusion dans le monde de l’industrie. Il s’agit pour moi d’une erreur à but lucrative mais non pérenne. À discuter. . .

3. Qui se trouvent être suffisamment nombreux pour avoir une mauvaise influence sur la réputation du langage. . .

4. comme la programmation orientée objets

5. On pourra penser au *garbage collector* introduit dans une quantité phénoménale de langages

6. Il n’y a qu’à voir la taille de la bibliothèque standard du C++ par rapport à celle du C

```
1  #include <stdio.h>
2
3  int main(int argc, char **argv) {
4      printf("Hello World !\n");
5      return 0;
6  }
```

Les différents éléments du programme sont directement visibles.

Il existe deux grandes familles d'éditeurs de code :

- Les Environnements de Développement Intégrés (EDIs) : en général spécialisés dans un unique langage de programmation, ils incluent le compilateur/interpréteur de celui-ci et tous les outils nécessaires pour programmer dans ce langage.
- Les éditeurs de texte : permettent d'éditer n'importe quel langage de programmation, mais ne fournissent aucun outil de compilation/interprétation, qui doit être installé à part (*recommandé, car l'apprentissage de la compilation "à la main" est utile*)

Il existe plusieurs EDIs/éditeurs de code connus qui permettent d'éditer des programmes dans la plupart des langages de programmation :

- Sublime Text : un éditeur de texte simple mais complet, recommandé pour commencer (<https://www.sublimetext.com/download>).
  - Atom : une variante de Sublime Text (<https://github.com/atom/atom/releases/tag/v1.60.0>)
  - Code : :Blocks : un EDI pour programmer en langages C et C++ sous Windows (<https://www.codeblocks.org/downloads/binaries/>)
  - Visual Studio Code : un éditeur de texte pouvant être étendu en EDI, probablement le plus connu et le plus utilisé (<https://code.visualstudio.com/>)
  - Visual Studio Community : un EDI pouvant être étendu sur plusieurs langages (extrêmement lourd, seulement utile pour de très gros projets professionnels)
  - EMACS : un éditeur de texte pouvant être étendu en EDI pour n'importe quel langage de programmation, envoyer des mails, naviguer sur Internet, ouvrir des PDFs, etc. . . , etc. . . (nécessite un peu de pratique avant de pouvoir être utilisé à son plein potentiel) (<https://www.gnu.org/software/emacs/>).
- Il s'agit en vérité de beaucoup plus que d'un simple EDI/éditeur. On ne le sait pas au moment où on l'installe, c'est tout. . .
- vim : un éditeur de texte pour les systèmes d'exploitation Linux uniquement (nécessite un peu de pratique avant de pouvoir être utilisé à son plein potentiel) (installé par défaut)

## Compilateur

### Sous Windows :

Pour installer un compilateur indépendant sous Windows, il faut télécharger l'archive à l'adresse : [https://github.com/brechtsanders/winlibs\\_mingw/releases/download/14.1.0posix-18.1.5-11.0.1-ucrt-r1/winlibs-x86\\_64-posix-seh-gcc-14.1.0-llvm-18.1.5-mingw-w64ucrt-11.0.1-r1.zip](https://github.com/brechtsanders/winlibs_mingw/releases/download/14.1.0posix-18.1.5-11.0.1-ucrt-r1/winlibs-x86_64-posix-seh-gcc-14.1.0-llvm-18.1.5-mingw-w64ucrt-11.0.1-r1.zip) puis extraire l'archive dans C :/ de sorte à avoir un répertoire C :/mingw64/.

Le compilateur est alors dans le répertoire C :/mingw64/bin/ et pourra être utilisé pour compiler les programmes écrits en C ou en C++.

### Sous Linux :

Un compilateur du langage C est présent par défaut dans le système d'exploitation. Seul l'EDI/éditeur de texte doit être installé

### 2.3.2 Compiler le premier programme

Il faut pour compiler ce premier programme connaître le principe d'un *terminal de commande*. Un terminal de commande permet d'exécuter les fonctionnalités du système d'exploitation sans passer par une interface graphique (c'est-à-dire des boutons, des fenêtres, etc...). Toutes les actions sont effectuées par *lignes de commande*. À une époque où l'interface graphique n'existait pas, ces terminaux de commande étaient les seuls moyens de manipuler un ordinateur.

Le code du premier programme doit être écrit dans un fichier de nom quelconque (appelé "*main.c*" par convention).

Pour la suite, on pourra utiliser l'arborescence de fichiers suivante<sup>7</sup> :

```
Documents
├─ Apprendre_le_C
│   └─ src
│       └─ main.c
```

Le répertoire "src"<sup>8</sup> contiendra les fichiers de code C. Cette arborescence sera développée et étendue dans par la suite.

#### Sous Windows :

Il existe deux terminaux de commande sous Windows. Le plus connu et le moins utile est le terminal dit *CMD*. Il est hérité des anciennes versions de Windows, et ne permet pas de faire grand-chose. Le plus généraliste est le terminal plus récent *PowerShell* qui permet une souplesse de manipulation plus élevée. En vérité, PowerShell est lui-même un langage de programmation impératif interprété à très haut niveau d'abstraction spécialisé dans la manipulation du système d'exploitation Windows.

Cependant, le langage PowerShell demande plus de ressources à l'ordinateur, et dans notre cas, le CMD est suffisant.

On peut exécuter l'un ou l'autre en maintenant les touches **Windows + R** du clavier et en tapant dans la fenêtre qui apparaît soit `cmd`, soit `powershell`.

Dans les deux cas, comme PowerShell comprend les possibilités du CMD, on peut exécuter les mêmes lignes de commande pour compiler notre programme :

Compiler sous Windows

```
1 C:\Users\user> cd Documents/Apprendre_le_C
2 C:\Users\user\Documents\AppData\Local\Temp\1\GCC-32756\src> C:/mingw64/bin/gcc.exe src/main.c -o main.exe
3 C:\Users\user\Documents\AppData\Local\Temp\1\GCC-32756\src> main.exe
```

« user » désigne votre nom d'utilisateur. Si une erreur apparaît à la première ligne, il faut taper **Utilisateurs** au lieu de **Users** (le système peut être en français).

7. Vous êtes bien sûr libre de faire autrement, il s'agit simplement de poser une structure conventionnelle pour la suite du cours.

8. Pour "source"

**Sous Linux :**

Il n'existe qu'un seul type de terminal sous Linux<sup>9</sup>. On ouvre ce terminal par les touches CTRL + ALT + T. Il suffit ensuite d'écrire :

**Compiler sous Linux**

```
1 user@computer ~-> cd ~/Apprendre_le_C
2 user@computer ~/Apprendre_le_C> gcc src/main.c -o main
3 user@computer ~/Apprendre_le_C> ./main
```

**Dans les deux cas :**

La première ligne modifie le répertoire de travail du terminal.

La deuxième ligne compile notre programme sous la forme d'un exécutable appelé "main.exe" ou "main".

La troisième ligne commande à l'ordinateur d'exécuter le programme "main.exe" ou "main" au sein du terminal.

Le résultat devrait être l'affichage du texte "Hello World!" à l'écran.

### 2.3.3 Analyse du premier programme

On rappelle le premier programme écrit en C :

**Un premier programme**

```
1 #include <stdio.h>
2
3 int main()
4 {
5     printf("Hello World !\n");
6     return 0;
7 }
```

Bien qu'il s'agisse d'un programme très simple, il pose les fondements du langage par bien des aspects. L'une des premières tâches à effectuer lors de la découverte d'un langage est d'apprendre les nombreux *mot-clés* et symboles du langage de programmation. Une fois que vous aurez appris la signification sous-jacente au code, vous serez en mesure de "parler" au compilateur et de lui donner vos propres ordres et de construire n'importe quel type de programme que vous êtes assez inventif et ingénieux pour créer<sup>10</sup>. Le compilateur va ensuite le transformer en langage binaire et l'ordinateur l'exécutera.

Mais il est à noter que connaître la signification des symboles arcaniques du langage n'est pas tout ce qu'il y a à faire en programmation. Vous ne pouvez pas maîtriser une autre langue en lisant un dictionnaire de traduction. Pour parler couramment une autre langue, il faut s'entraîner à converser dans cette langue. L'apprentissage d'un langage de programmation n'est pas différent. Il faut s'entraîner à "parler" au compilateur avec le code source écrit. Tous les codes écrits dans ce cours doivent être retapés à la main (sans copié-collé qui n'apprend rien) et il ne faut pas hésiter à les expérimenter et les modifier avec curiosité.

9. Pour simplifier... que les puristes pardonnent.

10. Avec certaines contraintes théoriques et pratiques

## Analyse ligne par ligne

```
| #include <stdio.h>
```

Cette première ligne présente un aspect très puissant du langage C (toujours présent dans une certaine mesure dans les autres langages de programmation impératifs) : la *modulation*.

La modulation consiste à diviser un programme en plusieurs ensembles de sous-programmes appelés *modules* (*modules* en anglais) qui vont chacun définir des fonctionnalités. Ces fonctionnalités sont ensuite utilisés dans un programme principal. Ce découpage permet de structurer un programme. Dans le cas de très gros projets (de plusieurs milliers, dizaines de milliers voire centaines de milliers de lignes de code<sup>11</sup>), ce découpage est obligatoire pour pouvoir se retrouver dans le programme et savoir où les fonctionnalités ont été développés. On peut y penser comme à la fabrication d'un avion. L'entreprise Airbus ne fabrique pas l'intégralité de ses avions au même endroit. Il s'agit pour une part de l'assemblage de différents composants (ici les modules) construit à des endroits différents (parfois par des entreprises différentes).

Un *bibliothèque* (*library* en anglais) est une collection de modules.

Le langage C présente une très grande quantité de fonctionnalités qui ont déjà été programmés et qu'il suffit de ré-utiliser. Pour ne pas avoir à surcharger notre programme avec des fonctionnalités inutiles, il est possible de choisir les fonctionnalités incluses. Cela se fait par l'instruction `#include` qui permet d'inclure le contenu d'un module choisi. Par "inclure", il faut entendre : copier l'ensemble du code écrit dans le module dans notre programme.

Le module `stdio` est un module de la bibliothèque *standard* pour les entrées/sorties (Input/Output). Une bibliothèque dite standard est une bibliothèque présente par défaut dans le langage, présente à son installation. En C, les modules de cette bibliothèque sont indiqués entre chevrons `<nom_module.h>`. Un module non standard (c'est-à-dire définie par le programmeur) est indiqué entre guillemets `"nom_module.h"`.

Les informations nécessaires de ces modules sont stockés dans des fichiers dit d'entête (*headers* en anglais) qui finissent par l'extension `.h`. Ce sont ces informations qui sont données pour permettre l'inclusion du code. Ces fichiers sont à différencier des fichiers d'extension `.c` qui définissent les fichiers de code du langage C (l'extension d'un fichier de code est modifié en fonction du langage de programmation : *py* en Python, *java* en Java, etc...)

```
| int main()
```

Cette instruction permet de définir la fonction principale (*main* en anglais signifie principal(e)). Une fonction en informatique est un algorithme avec certains paramètres appelés des entrées et qui renvoie une certaine valeur appelée sortie. Le concept de fonction en informatique est donc le même qu'en mathématiques.

La fonction `main` est celle qui est appelée à l'exécution du programme. En ceci, son adresse en mémoire définit le *point d'entrée* du programme, c'est-à-dire l'adresse à laquelle débute l'exécution du programme.

---

11. Les systèmes d'exploitation comme Linux ou Windows, ou encore les très gros logiciels comme les éditeurs de jeux vidéos tapent plutôt dans les quelques millions de lignes de code



On appelle *type de retour d'une fonction* le type de la valeur en sortie de la fonction. Ici, ce type est `int`, c'est-à-dire *integer* en anglais, qui signifie "entier". La fonction `main` renvoie donc un entier. L'entier renvoyé par la fonction `main` représente l'état du programme à la fin de son exécution. En général, 0 signifie que tout s'est bien passé, et `-1` signifie qu'une erreur a eu lieu au cours de l'exécution du programme.

Les parenthèses contiennent les paramètres de la fonction définie. On observe que dans ce cas-ci, la fonction `main` n'a aucun paramètre.

```
{  
  ...  
}
```

Les accolades définissent un *bloc de code*. Ce bloc est associé à la fonction `main`. Il contient l'algorithme exécuté par cette fonction.

```
| printf("Hello World !\n");
```

Cette instruction appelle la fonction `printf` écrite dans le module d'entrée/sortie `stdio` incluse plus haut. Cette fonction sert à afficher du texte sur un terminal de commande. Le caractère "`\n`" représente le retour à la ligne.

```
| return 0;
```

L'instruction `return` renvoie la sortie de la fonction écrite. Ici, la sortie de la fonction `main` est 0, c'est-à-dire que tout s'est bien passé. En général, le `return` à la fin de la fonction `main` renvoie toujours 0. En effet, si on est arrivé à la fin du programme, c'est qu'il n'y a pas eu de problème. On renvoie un code d'erreur seulement si une erreur nous empêche d'aller plus loin.

Partie II

# DU LANGAGE C



# INTRODUCTION

Cette deuxième partie du cours d'informatique porte sur la programmation en langage C, pour les raisons susdites dans la partie précédente.

L'objectif est d'offrir une vue qui se veut assez complète mais surtout rigoureuse<sup>12</sup>, afin d'amener le lecteur à une maîtrise du langage suffisante pour l'écriture de programmes quelques sur un ordinateur personnel. Le cours ne cherche donc pas à se spécialiser dans un domaine quelconque et aborde de manière générale la programmation en C<sup>13</sup>.

Dans cette optique, la progression se veut tout à fait linéaire et progressive dans le sens où chaque concept présenté ne nécessite pour sa bonne compréhension que les concepts présentés précédemment.

La structure générale du texte est thématique et chaque thème est suivi d'exercices pour assurer l'assimilation des notions abordées. On distingue quatre classes de thèmes :

- les fondamentaux relatifs à la syntaxe très générale du langage C et à quelques points généraux nécessaires à une vue d'ensemble
- les bases qui présentent l'essentiel du langage C, c'est-à-dire ce qui suffit à écrire des programmes quelconques en langage C
- les concepts avancés qui abordent certaines subtilités et particularités de première nécessité douteuse mais d'utilité avérée
- des concepts divers qui ne concernent pas le langage C en tant que tel mais plutôt certains outils et certaines méthodes nécessaires pour une programmation stable éclaircie du plus gros des embûches menaçant le débutant

## De l'universalité ou de la particularité des langages et notamment du C

Il peut venir à l'esprit à la lecture de ce document deux idées opposées. D'abord que les concepts présentés à propos du langage sont présents dans d'autres langages comme le Python, et semblent par ailleurs présenter un caractère universel à la programmation (on pensera par exemple au concept de *variable*). Par ailleurs, certaines particularités pourraient n'être que spécifiques au C et généralisable à aucun autre langage, comme cela peut être pensé des *classes de stockage*.

On remarque d'abord qu'il est absurde penser qu'un langage puisse être tout à fait particulier sans jamais présenter aucun point commun avec quelque langage que ce soit. Cela semble intuitif et est par ailleurs démontré mathématiquement par l'équivalence stricte des langages impératifs découlant des modèles de la machine de Turing et du  $\lambda$ -calcul.

Nuançons malgré tout. Certains concepts apparaissent dans la *plupart* des langages impératifs car ils corroborent l'intuition algorithmique humaine. Pourtant, ces concepts ne sont absolument pas inhérents aux théories du calcul dont découlent ces langages. Par exemple, la notion de *variable à valeur dynamique* peut être ignorée dans certains langages comme le *Haskell*, dont le principe est très calqué sur le  $\lambda$ -calcul et se trouve ainsi purement fonctionnel. Par ailleurs, d'autres concepts ne sont pas des conséquences directes de la théorie mais se trouvent être inhérents à la pratique, c'est-à-dire à l'implantation des langages sur un ordinateur. Ainsi, la nécessité de manipuler des adresses mémoires sur un ordinateur classique rend le concept de pointeur presque universel. Si l'utilisateur ne les utilise

---

12. Rigueur entendue comme non nécessairement formelle, c'est-à-dire qu'on n'explicitera pas la théorie du langage et la théorie de la compilation sous-jacentes

13. En particulier, rien ne sera dit vis-à-vis des spécificités de la programmation de systèmes embarqués bien qu'il s'agisse d'une spécialité de la formation ISMIN.

pas consciemment, les pointeurs apparaissent ainsi systématiquement dans le fonctionnement interne du langage.<sup>14</sup>

---

---

14. Sauf dans le cas de langages ne permettant que l'utilisation des registres du processeur, mais je n'ai jamais entendu parlé de tels langages

## CHAPITRE

### 3

# FONDAMENTAUX DU LANGAGE



## IDENTIFIANTS



### Définition 15 : Identifiant

Un identifiant est un symbole qui identifie une entité du langage (variable, routine, structure, etc...), c'est-à-dire qui la désigne. Un identifiant est composé de caractères compris dans l'ensemble :

$$\Sigma = \{0, \dots, 9\} \cup \{\_ \} \cup \{a, \dots, z\} \cup \{A, \dots, Z\}$$

Le symbole `_` s'appelle l'*underscore* (ou tiret du "8" en français, on conservera toutefois l'appellation anglaise car plus commune).

Un identifiant ne peut pas débiter par un chiffre. Dit autrement, l'ensemble des identifiants est l'ensemble des mots formés d'une lettre ou d'un underscore suivi potentiellement d'un nombre quelconque de lettres, de chiffres ou d'underscores.

### Exemples :

- les mots `_`, `a`, `b`, `t` et `z` sont des identifiants
- le mot `_uN_1denTiflant5_Qu3lqU0nqU3` est un identifiant
- le mot `J3_su1s_Un_SymboLe_Qu31qu0nQu3` est un identifiant
- le mot `nombre_2` est un identifiant
- le mot `2_nombres` n'est pas un identifiant (car il commence par un chiffre)
- le mot `dire_#_bonjour` n'est pas un identifiant (car `#`  $\notin \Sigma$ )
- le mot `_____` est un identifiant
- le mot `Salut_0uille` est un identifiant

Il existe un certain nombre d'identifiant dit *réservés* par le langage qui ne peuvent pas être utilisés ou dont l'utilisation amène un comportement imprévisible du programme. On distingue deux catégories de ces symboles :

- les mot-clés
- les identifiants débutant par un underscore

Certains mot-clés ont été affublés d'une version du langage C. Ces mot-clés ne sont donc présents qu'à partir de cette version (correspondant à l'année de publication). .

Voici la liste des standards principaux du langage qui ont été publiés (par ordre chronologique croissant) :

- K&R C : standard des créateurs du langage
- ANSI-C (C89) : première spécification dite *standard* par l'ANSI (*American National Standards Institute*)
- C99 : première spécification de l'ISO (*International Organization for Standardization*)
- C11
- C17
- C23

Les mots clés du langage C sont les suivants :

alignas (C23)	extern	sizeof	__Alignas (C11)
alignof (C23)	false (C23)	static	__Alignof (C11)
auto	float	static_assert (C23)	__Atomic (C11)
bool (C23)	for	struct	__BitInt (C23)
break	goto	switch	__Bool (C99)
case	if	thread_local (C23)	__Complex (C99)
char	inline (C99)	true (C23)	__Decimal128 (C23)
const	int	typedef	__Decimal32 (C23)
constexpr (C23)	long	typeof (C23)	__Decimal64 (C23)
continue	nullptr (C23)	typeof_unqual (C23)	__Generic (C11)
default	register	union	__Imaginary (C99)
do	restrict	unsigned	__Noreturn (C11)
double	return	void	__Static_assert (C11)
else	short	volatile	__Thread_local (C11)
enum	signed	while	

**Remarque 1 :** Quelque soit le standard utilisé pour votre propre compilation, il est extrêmement conseillé, dans un souci de compatibilité quand la convention n'est pas officiellement posé dans le cadre du projet, de ne jamais utiliser un mot-clé de ce tableau comme identifiant personnel quel que soit le standard utilisé.

**Remarque 2 :** Dans un souci de compatibilité quand la convention n'est pas officiellement posé dans le cadre du projet, il est conseillé de n'utiliser que les outils fournis par les standards chronologiquement inférieurs ou égales à C99 (c'est-à-dire sans utiliser d'outils des standards C11 et supérieur). Cela inclut les bibliothèques ajoutées dans les standards suivants du langage.

Dans les deux cas, la transmission du code source à un tiers ne compilant pas selon le même standard peut être potentiellement source d'erreurs et de bogues à la compilation.

La plupart des IDEs permettent de choisir entre les différents standards du compilateur. Lors d'une compilation "à la main", le paramètre `-std` permet de spécifier le standard. Par exemple, `gcc main.c -o main -std=c23` va compiler selon le standard C23.

### Internationalité des identifiants

Le code d'un programme doit être lisible par n'importe qui sur Terre. Dans le cas des programmes libres de droit (dits *open-sources* en anglais) par exemple, il est nécessaire que des développeurs de plusieurs pays puissent se comprendre. Dans le cas d'une entreprise de taille internationale, cela a aussi son importance. Pour cette raison, tous les identifiants d'un code doivent être en anglais de préférence.<sup>1</sup>



Les commentaires sont présents dans l'immense majorité des langages de programmation. Ils permettent d'annoter un code pour en expliquer certaines composantes ou pour exprimer certaines métadonnées vis-à-vis de celui-ci. Il peut s'agir par exemple de copier la licence utilisée dans le programme, de noter le nom de l'auteur du programme, d'expliquer le but d'un bloc de code, d'expliquer l'implantation pratique d'un objet théorique ou encore d'expliquer un aspect technique particulier du programme.

Lorsque le compilateur détecte un texte indiqué comme un commentaire, il l'ignore purement et simplement. Ainsi, il s'agit uniquement d'informations destinées au lecteur du code. Il peut y avoir plusieurs utilités :

- un développeur qui n'a plus touché à son code depuis six mois peut trouver extrêmement utile de l'avoir commenté lorsqu'il revient dessus
- dans le cas d'un projet conséquent, le développement est effectué en équipe. Un développeur peut avoir besoin de comprendre/modifier ce qui a déjà été écrit par un autre.

### Internationalité des commentaires

Le code d'un programme doit être lisible par n'importe qui sur Terre. Dans le cas des programmes libres de droit (dits *open-sources* en anglais) par exemple, il est nécessaire que des développeurs de plusieurs pays puissent se comprendre. Dans le cas d'une entreprise de taille internationale, cela a aussi son importance. Pour cette raison, les commentaires d'un programme doivent tous être écrits en anglais.<sup>2</sup>

En C, on distingue deux types de commentaires :

- les commentaires sur une ligne
- les commentaires par bloc

#### 3.2.1 Commentaires sur une ligne

Les commentaires sur une ligne sont indiqués par le double slash : //

Tous les caractères d'une ligne qui suivent un double slash sont considérés comme texte du commentaire, et donc ignorés par le compilateur.

Exemple

```
| printf("Salut les gens !\n"); // \n represents the line feed symbol
```

---

1. C'est triste mais c'est la vie.

2. C'est triste mais c'est la vie... Comment ça je me répète ?

**Remarque :** Le commentaire ne peut pas être inséré à l'intérieur d'une chaîne de caractères. Ainsi :

```
printf("Je ne dirais // pas bonjour !\n");
```

compile sans erreur et produit la sortie : Je ne dirais // pas bonjour.

### 3.2.2 Commentaires par bloc

Les commentaires par bloc permettent d'écrire des commentaires sur plusieurs lignes. Ils sont indiqués par une entrée de commentaire et une sortie de commentaire : `/*` et `*/`.

Exemple

```
1  /*
2  MINITEL Association
3  An example to illustrate comments
4
5  #include <stdio.h> // this line will never be read
6  */
7
8  #include <stdio.h> // Include the 'stdio' library
9
10 int main() { // Defines the main function of the program
11     return 0; // Means a successful running of the program
12 }
```



## LE POINT-VIRGULE



Le point-virgule (*semicolon* en anglais) en langage C permet d'indiquer la fin d'une instruction. Les commentaires et les directives préprocesseurs ne finissent pas par un point-virgule car ils ne sont pas considérés comme des instructions du langage (voir sections **Commentaires** et **Directives préprocesseurs**).

Le point-virgule ne peut en aucun cas être remplacé par un retour à la ligne (comme on le ferait en Python). Cependant, il permet d'écrire plusieurs instructions sur une seule et même ligne (comme en Python) :

```
1  #include <stdio.h> // preprocessor directives do not end with a semicolon character
2
3  int main() {
4      // instructions end with semicolon but not comments
5      printf("Salut !\n"); printf("Deuxieme Salut !\n");
6      return 0;
7  }
```





## POINT D'ENTRÉE DU PROGRAMME

Lorsqu'un programme est chargé en mémoire rapide pour être exécuté par l'ordinateur, il commence à une adresse  $a_0$  et fini à une adresse  $a_0 + l$ , où  $l$  est la taille du programme, c'est-à-dire le nombre d'octets du fichier binaire généré par le compilateur.

Cependant, le programme lui-même ne débute pas nécessairement à  $a_0$ . À cette adresse peut commencer la déclaration de données globales au programme, d'instructions tierces, de fonctionnalités autres, etc. . . C'est pourquoi le fichier enregistré sur le disque dur contient aussi un entête qui n'est pas le programme lui-même mais contient certaines informations sur le programme.

En particulier, cet entête contient une adresse relative  $a_{start} < l$  qui désigne le *point d'entrée* du programme. Lorsque le programme est chargé en mémoire rapide à l'adresse  $a_{start}$ , l'ordinateur commence son exécution à l'adresse  $a_0 + a_{start}$ .

Lors de la compilation, pour que le compilateur sache quel est le point d'entrée du programme, une convention a été posée : une fonction d'entrée du programme doit être codée, qui est désignée comme point d'entrée du programme. Cette fonction se nomme par défaut *main* en langage C. À la compilation, le compilateur posera comme point d'entrée l'adresse de la fonction *main* dans le programme. Elle renvoie un entier qui représente l'état de sortie du programme, et traduit l'événement : "l'exécution du programme a réussi" selon les conventions suivantes :

- EXIT\_SUCCESS : l'exécution du programme a réussi
- EXIT\_FAILURE : l'exécution du programme a échoué

**Remarque :** Ces deux identifiants sont définis dans la bibliothèque `stdlib` qu'il faut donc inclure par la ligne :

```
1 | #include <stdlib.h>
```

Les valeurs de ces identifiants diffèrent selon le système d'exploitation (voir section suivante). Sous Windows et Ubuntu, ces valeurs sont :

- EXIT\_SUCCESS = 0
- EXIT\_FAILURE = -1

On peut donc écrire :

### Exemple

```
1 | // the order of libraries includes doesn't matter
2 | #include <stdlib.h> // EXIT_SUCCESS and EXIT_FAILURE (among others)
3 | #include <stdio.h> // printf (among others)
4 |
5 | int main() {
6 |     return EXIT_SUCCESS; // Means a successful running of the program
7 | }
```



## DIRECTIVES PRÉPROCESSEURS (1)



Les directives préprocesseurs sont des instructions qui seront exécutées par le compilateur. Elles ne font donc pas partie du programme lui-même mais permettent de “diriger” la compilation du programme. Elles permettent par exemple de ne compiler certaines parties du code que sur un système d’exploitation spécifique, où plus généralement d’écrire dans un programme plusieurs versions d’un même bloc de code dont seule celle compatible avec le système d’exploitation sera choisie. Les directives préprocesseurs permettent également d’inclure du code d’autres programmes dans le nôtre, ou de définir des symboles représentant des valeurs constantes.

Les directives de préprocesseurs débutent par le caractère `#`. Seules les directives de préprocesseur commencent par ce caractère. Ainsi, `#include` est une directive de préprocesseur, qui va copier *avant la compilation proprement dite* le contenu de la bibliothèque ciblé dans le code du fichier qui effectue l’inclusion.

On peut aussi considérer la directive `#define` qui va permettre de définir des constantes (comme `EXIT_SUCCESS` et `EXIT_FAILURE`) :

Exemple

```
1  #include <stdio.h>
2
3  #define RETURN_CONSTANT 0
4
5  int main()
6  {
7      return RETURN_CONSTANT;
8  }
```

**Remarque 1 :** La définition d’un symbole par la directive `#define` remplace la valeur définie par ce symbole dans l’entièreté du code analysé par le compilateur avant que celui-ci ne compile réellement le programme.

**Remarque 2 :** Toutes les directives de préprocesseur ne sont pas données ici (et en vérité très peu). Il ne s’agit que des plus récurrentes dans la pratique.

**Remarque 3 :** Mis-à-part `#include` et `#define`, aucune directive de préprocesseur ne devrait être utilisée à tort et à travers. L’objectif des directives telles que celles-ci est la portabilité du code pour la compilation, et la facilitation d’écriture pour de très gros projets. Ainsi, en incluant d’autres codes, il devient possible de partitionner des programmes très long<sup>3</sup>. Ces directives ne font pas partie *en soi* du langage.

Un approfondissement technique sera effectué ultérieurement.

---

3. De l’ordre de la centaine de milliers voire de quelques millions de lignes de code par exemple, et sans aller jusque-là, de quelques milliers de lignes

## CHAPITRE

### 4

# BASES DU LANGAGE



Les variables constituent la brique de base du langage. Une variable au sens informatique est un conteneur d'une donnée dont la valeur peut évoluer au cours de l'exécution du programme.

Cette donnée représente de l'information. Elle est donc numérique, bien que l'interprétation puisse être de toute nature : image, vidéo, chaîne de caractère, nombre entier ou à virgule, arbre informatique, etc... On peut par exemple penser à la représentation d'une couleur comme un triplet  $(r, g, b, a) \in \llbracket 0, 255 \rrbracket^4$ , c'est-à-dire un mot binaire de quatre octets, et donc définir une variable de quatre octets qui stocke ces informations.

### Rapport entre variable informatique et variable mathématique :

Une variable en mathématique désigne un objet indéterminé. Cela semble au premier abord faux car il peut arriver qu'on écrive les phrases suivantes :

- “Soit  $e \in E$ . ...”, où  $E$  désigne un ensemble quelconque
- “ $\forall x \in E, \mathcal{P}(x)$ ”, où  $E$  désigne un ensemble quelconque et  $\mathcal{P}$  un prédicat quelconque à un seul paramètre

En y regardant de plus près, on observe qu'il s'agit en fait simplement de facilités d'écriture équivalentes aux phrases suivantes :

- “Soit  $e$  un objet indéterminé. Supposons  $e \in E$ . ...”, où  $E$  désigne un ensemble quelconque
- “ $\forall x, x \in E \implies \mathcal{P}(x)$ ”, où  $E$  désigne un ensemble quelconque et  $\mathcal{P}$  un prédicat quelconque à un seul paramètre

Cet objet finit par ne plus être indéterminé du fait d'hypothèses à son propos.

D'un autre côté, les variables informatiques n'évoluent pas dans le contexte d'une démonstration. Elles sont en fait directement liés au concept physique d'un ordinateur comme des identifiants associées à des adresses mémoire. Elles peuvent donc potentiellement évoluer et changer de valeur au cours du temps. Cette notion de temporalité est tout à fait absente du concept de variable mathématique.

### Approximation de stockage des variables

On considère dans un premier temps que les variables sont caractérisés par trois informations :

- une *étiquette*, aussi appelée le nom de la variable
- une adresse mémoire
- une donnée sur  $N$  octets (ou  $8N$  bits) à cette adresse

Une variable peut être vue alors comme une “case” de  $N$  octets dans la RAM, positionnée à une certaine adresse mémoire. Le langage C nous permet d'associer à cette adresse une étiquette, appelée le nom de la variable. Cette étiquette est oubliée après la compilation, et ne subsiste que l'adresse mémoire dans le programme en langage machine.

Visuellement :

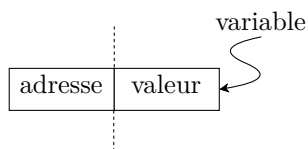


FIGURE 4.1 – Schéma du stockage d'une variable

### Type d'une variable

Pour définir le nombre d'octets sur lequel est stockée une variable, le langage C propose un certain nombre de “types” élémentaires directement intégrés au langage. Ces types représentent des sous-ensembles de  $\mathbb{Z}$  ou de  $\mathbb{R}$ <sup>1</sup>. Définir une variable selon un type du langage C consiste donc à limiter les données qu'elle contient à un sous-ensemble de  $\mathbb{Z}$  ou  $\mathbb{R}$  en définissant une taille maximum, en octets, de la donnée. Le type permet donc « d'interpréter » le mot binaire de  $N$  octets sur lequel la variable est définie avec une fonction *btoi* (*b*inary *t*o *i*nteger ou *btof* (*b*inary *t*o *f*loat).

Le langage C propose les types de base suivants :

- **char** : donnée  $d$  sur 1 octet (8 bits)  $d \in \llbracket -128; 127 \rrbracket$
- **short int** : donnée  $d$  sur 2 octets (16 bits)  $d \in \llbracket -32768; 32767 \rrbracket$
- **int** : donnée  $d$  sur 4 octets (32 bits)  $d \in \llbracket -2147483648; 2147483647 \rrbracket$
- **long int** : donnée  $d$  sur 8 octets (64 bits)  $d \in \llbracket -9223372036854775808; 9223372036854775807 \rrbracket$
- **float** : donnée  $d$  sur 4 octets (32 bits)  $d \in \mathbb{R}_{f32}$
- **double** : donnée  $d$  sur 8 octets (64 bits)  $d \in \mathbb{R}_{f64}$
- **long double** : donnée  $d$  sur 16 octets (128 bits)  $d \in \mathbb{R}_{f128}$
- **TYPE\*** : donnée  $d$  sur 8 octets qui est l'adresse mémoire d'une donnée de type TYPE quelconque

La syntaxe du langage C pour déclarer une variable d'un type quelconque TYPE est la suivante :

```
TYPE any_variable; // uninitialized variable : undertermined value
TYPE any_variable = ANY_VALUE; // initialized with value ANY_VALUE
```

Il n'est pas possible en C de déclarer deux fois un même identifiant. Ainsi, le code suivant provoque une erreur à la compilation :

1. C'est-à-dire indiquent soit une représentation entière soit une représentation flottante par la norme *IEEE 754*

```
| int age = 20; // declare age as an int with value 20
| char age = 19; // ERROR : age is already in use
```

Ainsi, un identifiant déclaré avec un certain type ne peut pas en changer jusqu'à ce qu'il ne soit plus défini (c'est-à-dire souvent jusqu'à la fin de l'exécution du programme). Après ne plus être défini, il pourra l'être à nouveau avec un autre type. Cela sera vu avec les *espaces de noms*.

**Remarque 1 :** L'exécution d'un programme en C est séquentielle. Pour cette raison, on ne peut référer à une variable qu'après l'avoir déclarée :

```
| test = 2; // ERROR : 'test' is not declared yet
| int test;
```

Pour cette raison, toutes les variables doivent être déclarées avant tous les lieux du code où elles se trouvent référencées.

**Remarque 2 :** le symbole `=` est un symbole d'*assignment*. Contrairement aux conventions mathématiques, il ne définit par une fonction binaire renvoyant *Vrai* si les deux objets sont égaux. Il permet cependant de changer à tout instant la valeur d'une variable déjà définie :

```
| int nombre_de_feuilles = 2000;
| nombre_de_feuilles = 1000; // Assigne à la variable nombre_de_feuilles la valeur 1000
```

On observe une équivalence stricte entre les expressions suivantes :

```
| // First expression :
| TYPE variable = valeur;
```

```
| // Second expression :
| TYPE variable; // undetermined value
| variable = valeur;
```

La première expression est une facilité d'écriture très largement utilisée<sup>2</sup>

En conséquence, le code suivant est tout à fait correct, bien qu'inutile sous cette forme :

```
| int a = a; // a's value is still undetermined
```

En effet, le symbole *a* est défini avant l'assignation.

### 4.1.1 Taille et type d'une variable

Il est possible de récupérer la taille d'une variable après l'avoir déclarée, ainsi que son type (depuis la version C23)

Cela est effectué par les opérateurs unaires `sizeof` et `typeof`. L'opérateur d'alignement `_Alignof` sera détaillé dans une section ultérieure.

2. Appelée *sucre syntaxique* dans certains milieux ;-)

### Opérateur sizeof

L'opérateur `sizeof` est particulièrement commun. Son exécution est quasiment systématiquement effectuée à la compilation. En effet, il retourne une valeur numérique précalculable. La valeur du résultat est donc remplacée dans le code à la génération de l'exécutable.

`sizeof` renvoie le nombre d'octets  $N$  nécessaires au stockage de l'espace mémoire associé à une étiquette ou un type. On utilise en général `sizeof` sur les types :

```
sizeof(char); // 1
sizeof(short int); // 2
sizeof(int); // 4
sizeof(long int); // 8
sizeof(float); // 4
sizeof(double); // 8
sizeof(long double); // 16

TYPE a;
sizeof(a); // Bytes needed to save 'a' in memory
sizeof(TYPE); // same
```

**Remarque :** L'opérateur `sizeof` renvoie une valeur de type `size_t`, équivalente à un `long int`.

### Opérateur typeof

L'opérateur `typeof`, tout comme `sizeof`, est calculé au moment de la compilation. D'ailleurs, effectuer un tel calcul durant l'exécution du programme n'a aucun sens. En effet, `typeof` renvoie le type d'une variable. Les deux codes suivants sont donc strictement équivalents :

<code>typeof(int) a;</code>	<code>int a;</code>
<code>int b;</code>	<code>int b;</code>
<code>typeof(b) c;</code>	<code>int c;</code>

## 4.1.2 Entiers signés et non signés

Il est possible en langage C de choisir pour les nombres entiers entre l'interprétation signée de sa représentation binaire, et l'interprétation non signée. Cela se fait par les mot-clés `signed` et `unsigned`. Par défaut, les variables entières sont interprétés comme signées. Ainsi, le mot-clé `signed` n'est pas utile au moment de la déclaration de la variable. Cela peut être cependant utile pour une réinterprétation ultérieure (voir **Projection de type**). La syntaxe est la suivante :

```
char v = 130; // signed by default, truth is v = 2 - 128 = -126
unsigned short int v2 = -1; // unsigned interpretation, truth is v2 = 65535
signed int v3 = -4; // 'signed' is useless because it's implicit
// etc...
```

**Remarque :** Le langage C ne propose pas de nombres flottants non signés car le standard de représentation des nombres flottants implique directement la présence du signe.

### 4.1.3 Introduction à la portée des variables : les blocs

Les variables sont déclarés dans des blocs de portée. Ces blocs de portée sont indiqués par les accolades gauche et droite { et }. Ils indiquent dans quelle région du programme les variables sont déclarés. En particulier, il est possible d'imbriquer les blocs entre eux. Cela permet de remplacer dans une certaine partie du code une variable par une autre du même. Cela peut être intéressant pour rester lisible et utiliser des variables qui correspondent toujours à leur utilité :

```

1  #include <stdlib.h>
2
3  int main()
4  { // First bloc
5      int a = 5; // declared in the upper 'main' bloc
6      { // Second bloc nested in the first one
7          // at this point, 'a' is still equal to 5
8          double a = 3.14; // Not the same 'a'
9          // at this point, 'a' is equal to 3.14
10     }
11     // 'a' get back to 5
12     return EXIT_SUCCESS;
13 }
```

Ces blocs peuvent être définis partout dans un code. Il est possible de déclarer des variables dans ou hors de ces blocs.

En particulier, il est possible de définir des variables hors du bloc `main`. Ces variables sont dites globales car accessibles dans tous les blocs du programmes même ceux qui ne sont pas le bloc de la fonction `main`<sup>3</sup> :

```

1  #include <stdlib.h>
2
3  int some_global_variable = VALUE;
4  int main() {
5      char some_global_variable = OTHER_VALUE;
6      // some code
7  }
8
9  // Some other code may access to 'some_global_variable'
```

**Remarque :** Maîtriser la portée des variables et ne pas garder déclarée une variable alors qu'elle n'est plus utile permet au compilateur d'optimiser le code machine produit. Par exemple, dans le code suivant :

```

1  #include <stdlib.h>
2
3  int main() {
4      {
5          int v = VALUE;
6          // using 'v'
7      }
8      // Some big code
```

3. On verra comment définir d'autres fonctions dans la section sur les **Routines** et on détaillera les notions d'espaces locaux et d'espace global dans le prochain chapitre, dans la section sur les classes de stockage.

```

9  {
10     int v = ANOTHER_VALUE;
11     // using 'v'
12 }
13 return EXIT_SUCCESS;
14 }

```

$v$  possède une valeur particulière d'initialisation dans une région du code, puis une autre valeur particulière d'initialisation dans une autre région du code très éloignée de la première. Indiquer cela avec les blocs de portés permet d'indiquer au compilateur qu'il est inutile de stocker la valeur de  $v$  entre les deux régions.

#### 4.1.4 Exercices

**Exercice 10 (Intervention de variables par effet de bord).** Écrire un programme en C qui initialise deux variables  $a$  et  $b$  de valeurs différentes. En initialisant au maximum une seule variable supplémentaire, échanger les valeurs des deux premières variables. Ainsi, si  $a = 8$  et  $b = 6$  à leurs initialisations, alors en fin d'exécution il faut avoir  $b = 8$  et  $a = 6$  sans que les assignations soient explicites. C'est-à-dire que le code ne doit pas à être modifié si les valeurs de  $a$  et  $b$  sont changées.



Cette section ne détaille pas ce qu'est techniquement une chaîne de caractères. Elle présente seulement le *formatage d'une chaîne de caractères*, c'est-à-dire la mise en forme. Il est sous-entendu en programmation que le formatage d'une chaîne de caractères est effectué selon des variables. De manière générale, le système de formatage d'une chaîne de caractères en C permet d'inclure dans une chaîne de caractères les valeurs de variables quelconques.

Le formatage d'une chaîne est effectué à l'aide de deux caractères spéciaux :

- le caractère de formatage des caractères spéciaux \
- le caractère de formatage de variables %

Aucun de ces deux caractères ne peut être affiché directement dans une chaîne de caractère :

```

1  #include <stdio.h>
2  #include <stdlib.h>
3
4  int main() {
5      // produces a compilation warning and prints "'est pas affichable." :
6      printf("% n'est pas affichable.");
7      // compilation error :
8      printf("\ non plus.");
9      return EXIT_SUCCESS;
10 }

```

Pour afficher réellement un des ces deux caractères, il faut l'*échapper* en le répétant. On appelle cette opération l'échappement car elle permet au caractère d'échapper au formatage de la chaîne de caractères.



```

1  #include <stdio.h>
2  #include <stdlib.h>
3
4  int main() {
5      printf("%% n'est pas affichable.\n");
6      printf("\\ non plus.\n");
7      return EXIT_SUCCESS;
8  }

```

### 4.2.1 Formatage des caractères spéciaux

Les caractères considérés comme spéciaux sont :

- caractère de passage à la ligne suivante : `\n`<sup>4</sup>
- caractère retour au début de la ligne actuelle : `\r`<sup>5</sup>
- caractère de tabulation : `\t`
- caractère de chaîne de caractère " : `\"`

Exemples

```

1  #include <stdio.h>
2  #include <stdlib.h>
3
4  int main() {
5      printf("Pas de retour a la ligne apres cette phrase.");
6      printf("Alors que la oui.\n");
7      printf("Recommencer au \"debut\" : \rdebut\n");
8      printf("Une tabulation : \t c'etait une tabulation.\n");
9      return EXIT_SUCCESS;
10 }

```

### 4.2.2 Formatage de variables

Le formatage de variables nécessite d'indiquer le type de variable qui va être affichée. En effet, en fonction de son type, l'interprétation de sa représentation binaire change.

On distingue les principaux modificateurs d'affichage :

- signed char OU signed short int OU signed int : `%d`
- signed long int : `%ld`
- unsigned char OU unsigned short int OU unsigned int : `%u`
- unsigned long int : `%lu`
- float : `%f`
- double : `%lf`
- long double : `%Lf`

4. Ce caractère est désigné en anglais par le symbole LF pour “Line Feed”

5. Appelé aussi *retour chariot*, en référence aux machines à écrire. Ce caractère est désigné en anglais par le symbole CR pour “carriage return”

Ces modificateurs sont insérés dans la chaîne de caractères. Il devient ensuite possible de donner en argument à la fonction `printf` les valeurs à insérer en lieu et place des modificateurs :

```

1  #include <stdio.h>
2  #include <stdlib.h>
3
4  #define N 3
5
6  int main() {
7      printf("Valeur du nombre N : %d", N);
8      return EXIT_SUCCESS;
9  }
```

Il est possible de mettre autant de modificateurs qu'on le souhaite dans la chaîne de caractère. Il faut cependant respecter le nombre de modificateurs et mettre précisément le même nombre de variables :

```

// Error : no integer is given :
printf("%u est un entier non signé");
// Error : 42 should not be there :
printf("%lf : 64 bits, %f : 32 bits.", 3.1415926358, 2, 718281828, 42);
```

Bien sûr, les entrées peuvent être des variables :

```

1  #include <stdio.h>
2  #include <stdlib.h>
3
4  int main() {
5      unsigned short int a = -7;
6      int b = 10;
7      printf("%u + %d = %d\n", a, b, a+b);
8      return EXIT_SUCCESS;
9  }
```

### 4.2.3 Exercices

**Exercice 11 (Taille des types).** Écrire un programme en C qui affiche la taille en octets des différents types de base du langage.



## OPÉRATEURS SUR LES VARIABLES



### Définition 16 : Opérateurs et opérandes

Les opérateurs sont des fonctions d'arité strictement positive. Les opérandes sont les “termes” des opérateurs, c'est-à-dire les entités sur lesquels les opérateurs effectuent leur opération.

Il existe quatre familles principales d'opérateurs sur les variables en C :

- les opérateurs arithmétiques
- les opérateurs bit-à-bit
- les opérateurs logiques/relationnels
- les opérateurs d'assignation

La syntaxe générale des opérateurs sur les variables est la suivante (selon l'arité de l'opérateur) :

```
TYPE v1 = VALEUR1;
TYPE v2 = VALEUR2;

// unary operator :
<OPERATION> v1; // result is not saved

// binary operator :
v1 <OPERATION> v2; // result is not saved
```

Par ailleurs, certains opérateurs sont prioritaires sur d'autres. Pour forcer une opération à être évaluée avant une autre, il est possible de parenthéser les expressions :

```
TYPE v1 = VALEUR1;
TYPE v2 = VALEUR2;

TYPE v3 = (v2 <OP2> (v1 <OP1> v2)); // operator OP1 is evaluated before OP2
```

Par ailleurs, le langage organise les opérations selon une priorité par défaut. Si on écrit :

```
TYPE v3 = v2 <OP2> v1 <OP1> v2;
```

, alors dans le cas où <OP1> est prioritaire sur <OP2>, il n'y a pas modification par rapport à l'expression parenthésée. Dans le cas où <OP2> est prioritaire sur <OP1>, cela diffère.

Pour tous les opérateurs présentés dans la suite, on pourra se référer à la table de priorités suivante <sup>6</sup> :

---

6. Copiée de [https://en.cppreference.com/w/c/language/operator\\_precedence](https://en.cppreference.com/w/c/language/operator_precedence) parce-que sur ce genre de chose, inutile d'être original.

Priorité	Opérateur	Description	Associativité
1	++ - ( [] . → (type){list}	Incrémentation et décrémentation postfixes Appel de fonction Indexation de tableau Accès à un membre de structure Accès à un membre de structure par un pointeur Construction de littéral	Gauche à droite
2	++ - + - ! ~ (type)object * & sizeof _Alignof	Incrémentation et décrémentation préfixes Signes Non logique et bit à bit Projection de type Indirection Récupération de l'adresse Récupération de la taille Alignement	Droite à gauche
3	* / %	Multiplication, division et modulo	Gauche à droite
4	+ -	Addition et soustraction	Gauche à droite
5	<< >>	Fonctions << et >>	Gauche à droite
6	< <= > >=	Relations < et ≤ Relations > et ≥	Gauche à droite
7	== !=	Opérateurs relationnels d'égalité et de différence	Gauche à droite
8	&	ET bit-à-bit	Gauche à droite
9	^	OU exclusif bit-à-bit	Gauche à droite
10		OU inclusif bit-à-bit	Gauche à droite
11	&&	ET logique	Gauche à droite
12		OU logique	Gauche à droite
13	? :	Condition ternaire	Droite à gauche
14	= += -= *= /= %= <<= >>= &= ^=  =	Opérateur d'assignement simple Assignement par somme et différence Assignement par produit, quotient et reste Assignement par décalages bit-à-bit gauche ou droit Assignement par OU inclusif, OU exclusif, ET bit-à-bit	Droite à gauche
15	,	Virgule	Gauche à droite

TABLE 4.1 – Priorités des opérateurs en C

La priorité indique l'ordre de consommation des opérandes. Ainsi, un opérateur de priorité 1 effectuera son calcul sur ses opérandes avant les opérateurs de priorité 2, 3, etc...

### 4.3.1 Opérateurs arithmétiques

On liste les opérateurs arithmétiques suivants :

Opération	Arité	Symbole mathématique	Symbole en C
Addition	2	+	+
Soustraction	2	−	−
Multiplication	2	×	*
Division entière	2	÷	/
Modulo	2	%	%

TABLE 4.2 – Opérateurs arithmétiques

Ces opérateurs sont les plus communs du langage avec les opérateurs logiques. Ils permettent en particulier de modifier les variables assignés en effectuant des calculs.

```

1  #include <stdio.h>
2  #include <stdlib.h>
3
4  int main() {
5      int a = 18;
6      int b = 67;
7      printf("a = %d; b = %d\n", a, b);
8      printf("a + b = %d\n");
9      printf("a - b = %d\n");
10     printf("a * b = %d\n");
11     printf("a / b = %d\n");
12     printf("b / a = %d\n");
13
14     return EXIT_SUCCESS;
15 }
```

On observe que la division est bien entière en posant les divisions euclidiennes :

- $\left\lfloor \frac{18}{67} \right\rfloor = 0$  car  $18 = 67 \times 0 + 18$
- $\left\lfloor \frac{67}{18} \right\rfloor = 3$  car  $67 = 18 \times 3 + 13$

### 4.3.2 Opérateurs bit-à-bit

On liste les opérateurs bit-à-bit suivants :

Opération	Arité	Symbole mathématique	Symbole en C
ET	2	$\wedge$	<code>&amp;</code>
OU inclusif	2	$\vee$	<code> </code>
OU exclusif	2	$\oplus$	<code>^</code>
NON	1	$\neg$	<code>~</code>
Décalage à droite signé (ou arithmétique)	2	$\gg_a$	<code>&gt;&gt;</code>
Décalage à droite non signé (ou logique)	2	$\gg_l$	<code>&gt;&gt;</code>
Décalage à gauche	2	$\ll$	<code>&lt;&lt;</code>

TABLE 4.3 – Opérateurs bit-à-bit

Ces opérateurs sont décrits en détails dans la partie 1 du cours.

### 4.3.3 Opérateurs logiques, ou relationnels

#### Définition 17 : Vérité d'une expression

On dit qu'une expression est *fausse* si sa valeur est égale à 0. On dit qu'elle est *vraie* sinon.

Les opérateurs logiques, contrairement aux opérateurs bit-à-bits, ne travaillent pas sur les valeurs de variables mais sur leur valeur de vérité. En particulier, ils ne renvoient que 1 ou 0, c'est-à-dire *vrai* ou

*faux.*

On liste les opérateurs logiques suivants :

Opération	Arité	Symbole mathématique	Symbole en C
Égalité	2	=	==
Différence	2	≠	!=
ET logique	2	∧	&&
OU logique	2	∨	
NON	1	¬	!

TABLE 4.4 – Opérateurs logiques

**Remarque :** Il s’agit d’un OU *inclusif*. Le OU exclusif n’a pas de version “logique” en C.

Quelques exemples :

```

1  int a = 5;
2  int b = 7;
3  printf("%d\n", !(b-a)); // b-a = 2 is true so !(b-a) is false : prints 0
4  printf("%d\n", !(a + b)); // prints !0, so prints 1
5  printf("%d\n", a != b); // prints 1
6  printf("%d\n", a == b); // prints 0
7  printf("%d\n", !(a+b) == (a != b)); // prints 1 == 1 so prints 1
8  printf("%d\n", a != b && a == b); // prints 0
9  printf("%d\n", a == b || (a + b)); // prints 1
10 printf("%d\n", a-b && (a != b)); // prints 1 because both are true

```

**Remarque :** On observe que pour tout entiers  $a$  et  $b$ ,  $a - b \equiv a \neq b$ . En effet,  $a \neq b \Leftrightarrow a - b \neq 0$

### 4.3.4 Opérateurs d’assignation

Les opérateurs d’assignation sont à la fois les éléments les plus connus par les débutants et à la fois les opérateurs dont ces mêmes débutants ignorent tout des spécificités. Voici un exemple qui pourrait surprendre même des programmeurs ayant plusieurs années d’expérience :

```

1  #include <stdio.h>
2  #include <stdlib.h>
3
4  int main() {
5      int a;
6      printf("Initialement, a = %d\n", a = 0);
7      printf("Puis a = %d\n", ++a);
8      printf("Et enfin, a = %d\n", a <= 3);
9      return EXIT_SUCCESS;
10 }

```

Les différentes opérations présentées dans ce code sont expliquées dans la suite, bien qu’une exécution et quelques tests accompagnés d’un peu d’intuition pourraient suffire à comprendre.

### Opérateur élémentaire d'assignation =

L'opérateur binaire `=`, dit d'assignation, n'effectue pas que l'assignation d'une valeur à une variable. Il renvoie également la valeur assignée.

Ainsi, l'opération `a = 2` est égale à 2. On peut donc écrire de manière équivalente :

<pre>int a; int b = (a = 3) + 1;</pre>	<pre>int a = 3; int b = a + 1;</pre>
--	--------------------------------------

En particulier, il est possible d'initialiser plusieurs variables à une même valeur par le code suivant, tout à fait illisible :

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  int main() {
5      int a, b, c, d, e = (d = (c = (b = (a = 3))));
6      printf("a : %d, b = %d, c = %d, d = %d, e = %d\n", a, b, c, d, e);
7      return EXIT_SUCCESS;
8  }
```

Il n'est pas particulièrement utile d'utiliser cette particularité de l'opérateur d'assignation sauf dans des cas très spécifiques. Mais c'est toujours bon à savoir lorsqu'on tombe sur le code d'un tiers qui l'utilise.

### Combinaison de l'opérateur d'assignation et des opérateurs binaires de calcul

Les opérateurs binaires arithmétiques et bit-à-bits peuvent être combinés avec l'opérateur d'assignation pour contracter le code :

```
int a = ...; // any value
a += 3; // a = a + 3
a -= 3; // a = a - 3
a *= 3; // a = a * 3
a /= 3; // a = a / 3
a %= 3; // a = a % 3
a |= 3; // a = a | 3
a &= 3; // a = a & 3
a ^= 3; // a = a ^ 3
a <<= 3; // a = a << 3
a >>= 3; // a = a >> 3
```

L'équivalence à l'expression en commentaire est totale. Ainsi, ces combinaisons renvoient la nouvelle valeur de `a`.

### Incrémentation et décrémentation

En raison de présence extraordinairement récurrente des deux opérations d'incrément et de décrémentation (c'est-à-dire l'ajout de 1 à la valeur d'une variable, ou sa diminution de 1), deux opérateurs spécifiques existent pour performer de manière optimisée ces opérations :

■ `++variable;`<sup>7</sup>

7. Ce qui a d'ailleurs amené au nommage du langage *C++* comme une version "incrémentée", étendue, du langage *C*

■ `-variable`;

Il y a équivalence entre les expressions suivantes :

```
int a = 0;
a += 1;
a -= 1;
```

```
int a = 0;
++a; // returns 1
--a; // returns 0
```

Ainsi, `++a` renvoie la valeur de *a* après incrémentation et `--a` renvoie la valeur de *a* après décrémentation.

Toutefois, la forme la plus connue de l'incrément et de la décrémentation est la suivante :

```
int a = 0;
a++; // returns 0
a--; // returns 1
a; // returns 0
```

Dans ce cas, la valeur renvoyée est celle *avant* l'incrément ou la décrémentation. Ce qui explique le décalage dans le code juste ci-dessus.

**Remarque :** Chaque opération d'assignation renvoie une valeur numérique, et non une variable. Ainsi, chacune des lignes du code suivant provoque une erreur car n'a aucun sens :

```
(a = N)++;
(a++) = N;
a += 2 += 2;
a = b = 3;
```

### 4.3.5 Exercices

Voir [11] pour approfondir les calculs binaires optimisés.

**Exercice 12 (Valeur).** Quelle est la valeur de *i* après la suite d'instructions :

```
int i = 10;
i = i - (i--);
```

Quelle est la valeur de *i* après la suite d'instructions :

```
int i = 10;
i = i - (--i);
```

**Exercice 13 (Calcul d'expressions).**

Évaluer à la main les valeurs de *a*, *b*, *c* et *d* écrites en langage C :

```
int a = 57 + (4 - 6);
int b = a - 2 * 7;
int c = b / 3 * 4 + b;
unsigned short int d = c - (100 % c + 100);
```



**Exercice 14 (Priorité des opérateurs).** Enlever les parenthèses des expressions suivantes lorsqu'elles peuvent être retirées (c'est-à-dire que le résultat du programme reste le même) :

```
int a = 6, b = 12, c = 24;
a = (25*12) + b;
printf("%d", ((a > 4) && (b == 18)));
((a >= 6) && (b < 18)) || (c != 18);
c = (a = (b + 10));
```

**Exercice 15 (Interversion sans effet de bord (1)).** Écrire un programme en C qui initialise deux variables  $a$  et  $b$  de valeurs différentes. *Sans initialiser une seule variable supplémentaire*, échanger les mots binaires des deux variables. Ainsi, si  $a = a_{N_a-1} \dots a_0$  et  $b = b_{N_b-1} \dots b_0$  à leurs initialisations, alors en fin d'exécution il faut avoir  $b = a_{N_a-1} \dots a_0$  et  $a = b_{N_b-1} \dots b_0$  sans que les assignations soient explicites. C'est-à-dire que le code ne doit pas à être modifié si les valeurs de  $a$  et  $b$  sont changées. Justifier que ce programme est toujours correct pour des entiers codés chacun sur  $N$  bits, quelques soient les signatures de  $a$  et  $b$ . Ce programme est-il toujours correct si  $a$  et  $b$  ne sont pas écrits sur autant de bits ? Justifier.

**Exercice 16 (Interversion sans effet de bord (2)).** *idem* que pour l'Exercice 15. mais l'unique opérateur autorisé est l'opérateur de OU exclusif (noté  $\wedge$  en C).

**Exercice 17 (Multiplication par décalage).**

On remarque que pour entiers  $n, m \in \mathbb{N}$ ,  $n \ll m = n \times 2^m$ . En utilisant uniquement des additions ou soustractions de décalages, effectuer la multiplication  $57 \times 14$  :

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  int main() {
5      int a = 57;
6      int r;
7      // code to determinate
8      printf("57x14 = %d", r);
9      return EXIT_SUCCESS;
10 }
```

Minimiser le nombre de décalages et d'additions/soustractions (objectif : 2 décalages et une addition ou soustraction)

**Exercice 18 (Valeur absolue).** En utilisant uniquement des opérations arithmétiques et bit-à-bits, écrire un programme qui calcule la valeur absolue d'un entier signé :

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  int main() {
5      int x = ...; // any positive or negative value
6      unsigned int abs_x;
7      // code to determinate
8      printf("|%d| = %u", x, abs_x);
9      return EXIT_SUCCESS;
10 }
```



# PROJECTION DE TYPE



La *projection de type* (*static cast* en anglais) est une opération unaire du langage C qui permet la conversion d'un variable d'un type à un autre.

L'exemple suivant, extrêmement simple, va servir d'illustration à la projection de type : on souhaite diviser un nombre stocké comme un entier par un diviseur stocké comme un entier, et obtenir le nombre flottant résultant. C'est-à-dire ne pas effectuer une division entière mais réelle.

On essaie dans un premier temps le code suivant :

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  int main() {
5      int some_integer = 7851;
6      int divider = 100;
7      double a = some_integer/divider;
8      printf("%.2lf\n", a);
9      return EXIT_SUCCESS;
10 }
```

Après exécution, on a le résultat : 78.00

Le problème se situe au calcul `some_integer/100`. En effet, l'opération effectuée est une division entière. Le résultat de cette division est donc 78, qui est ensuite convertit implicitement en `double` par l'opérateur `=`.

Il faudrait que la division effectuée ne soit pas une division entière. Pour cela, il est nécessaire de modifier l'interprétation d'au moins une des deux variables (`some_integer` ou `divider`) pour qu'elle soit interprétée comme un `double` AVANT la division. On ne souhaite cependant pas modifier le type d'une des deux variables de manière définitive.

La solution à cela est la *projection de type*. Il s'agit de forcer le compilateur à réinterpréter/convertir la variable en un autre type que celui auquel elle a été assignée. La syntaxe est la suivante :

```
TYPE1 variable;
TYPE2 reinterpreted_variable = (TYPE2)(variable);
```

Dans notre cas, il suffit d'écrire :

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  int main() {
5      int some_integer = 7851;
6      int divider = 100;
7      double a = (double)(some_integer)/divider;
8      printf("%.2lf\n", a);
9      return EXIT_SUCCESS;
10 }
```

Le résultat après exécution est bien : 78.51

**Remarque :** Le mot binaire désigné par `nombre_entier` est différent de celui de `(double)(nombre_entier)` du fait de l'écriture des nombres flottants. Il s'agit donc d'une conversion. Il est possible aussi que la projection de type n'induisse pas de réécriture du mot binaire, comme dans le cas d'une réinterprétation de la signature d'un nombre.

```

1  #include <stdio.h>
2  #include <stdlib.h>
3
4  int main() {
5      unsigned short int v = -1;
6      printf("%d\n", v); // prints 65535
7      printf("%d\n", (signed short int)(v)); // prints -1 but it's the same binary word
8      return EXIT_SUCCESS;
9  }
```

### 4.4.1 Exercices

#### Exercice 19 (Quelques évaluations).

Dire pour chaque expression si elle est vraie ou fausse (respectivement : différente ou égale à 0) :

```

int e1 = 0xFFFFFFFF00 != 0xFFFFFFFF - (char)(-1);
int e2 = 0xFFFFFFFF - (unsigned char)(-1) - 0xFFFFFFFF00;
int e3 = !(e1 == e2) || (1 ^ e2);
int e4 = (unsigned char)(!(((64 ^ e3) % 8) - 1) + 256) - 2;
int e5 = e1 && e3;
```



## STRUCTURES DE CONTRÔLE



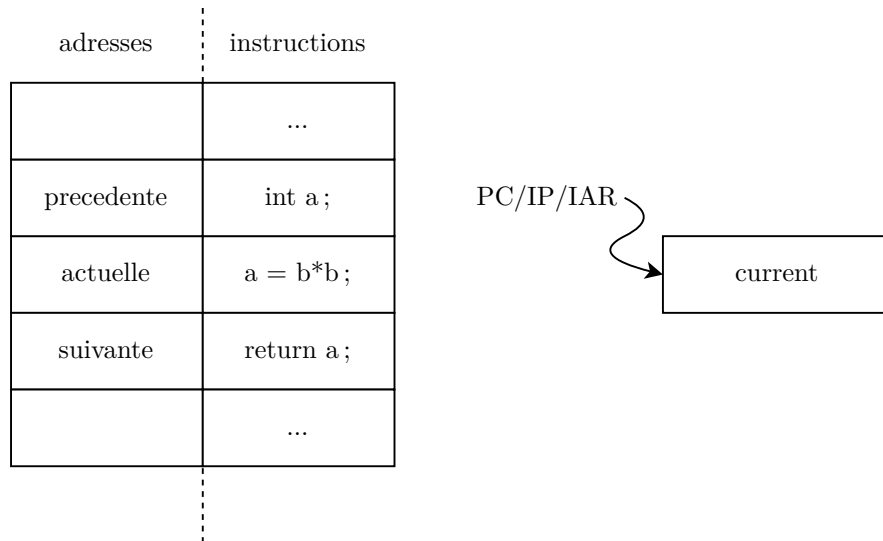
### Définition 18 : Flot d'exécution/de contrôle

Le flot d'exécution ou flot de contrôle est l'ordre dans lequel les instructions d'un programme impératif sont exécutés. Il est possible de modifier cet ordre grâce à des structures de contrôle de flot.

Un programme informatique est constitué d'une succession d'instruction binaires chacune chargée en mémoire. Il existe dans le processeur un registre qui stocke l'adresse de l'instruction à exécuter. Ce registre est appelé par différents noms :

- *PC* pour *Program Counter*
- *IP* pour *Instruction Pointer*
- *IAR* pour *Instruction Address Register*

La manipulation de la valeur ce registre permet de se “déplacer” dans le programme. Ces modes de déplacements sont décrits dans les sections suivantes.



La première sous-section se veut une introduction plus bas-niveau du contrôle du flot d'exécution, c'est-à-dire détaillant de manière moins abstraite et plus technique le contrôle du flot d'exécution.<sup>8</sup> Il est possible d'aller directement à la sous-section décrivant les structures complexes, majoritairement utilisées en langage C.

### 4.5.1 Structures élémentaires

Voir [3] pour plus de détails.

Un ordinateur utilise les structures élémentaires de contrôle de flot suivantes :

- la séquence
- l'arrêt de programme
- le saut incondtionnel
- le saut conditionnel

Les structures de contrôle plus abstraites sont ensuite construites par l'utilisation de ces structures élémentaires.

#### Séquence

La séquence est une structure de contrôle implicite, dont la gestion est réservée au processeur de l'ordinateur. Il s'agit simplement du passage d'une instruction à la suivante. En langage C, c'est le point-virgule indiquant la fin d'une instruction qui se charge d'indiquer le passage à la suivante.

La fin d'un instruction va donc amener à l'incréméntation du registre  $IP$  de la taille de l'instruction. Imaginons que le processeur exécute la  $i^e$  instruction de taille  $t_i$  octets à l'adresse  $a_i$ . À l'exécution de cette instruction,  $IP = a_i$ . À la fin de l'exécution de l'instruction,  $IP = a_i + t_i = a_{i+1}$ . Il s'agit de l'adresse de l'instruction suivante, puisque toutes les instructions sont contigües en mémoire.

8. Il s'agit d'une introduction légère au cours d'architecture des processeurs

Certaines architectures de processeur utilisent une taille fixe pour les instructions. Dans ce cas,  $\forall i, t_i = k \in \mathbb{N}$  est une constante. Cela accélère l'accès aux instructions par le processeur, mais limite le nombre possible d'instructions.<sup>9</sup>

### Arrêt de programme

La mise en pause d'un programme à un certain point de son exécution et sa reprise est nécessaire pour un système multi-tâches qui doit exécuter plus de programmes au même moment qu'il ne possède de processeurs. Il peut ainsi alterner l'exécutions des programmes pour simuler leur exécution parallèle aux yeux de l'utilisateur. On peut imaginer *par exemple* deux programmes utilisant chacun pendant 10 *ms* les ressources de l'ordinateur. Ils ne s'exécuteront que 500 *ms* chacun pendant une seconde, mais l'utilisateur pensera du fait de l'alternance très rapide d'exécution des programmes que ceux-ci s'exécutent en même temps (deux fois plus lentement cependant que si ils avaient été seuls à utiliser les ressources du processeur).

### Saut inconditionnel

Le saut inconditionnel consiste à attribuer au registre *IP* une valeur précisée dès que l'instruction est lue, sans aucune condition. Le processeur passe donc immédiatement à l'instruction souhaitée.

En langage C, l'instruction qui permet d'effectuer ce genre de saut est **goto**. Il faut lui préciser une adresse d'instruction du code C qui sera indiquée par un *label* définie selon la syntaxe suivante :

```
| mon_label:
```

Un exemple :

```
1  #include <truc.h>
2
3  int main() {
4      goto end;
5      // operations never executed
6  end:
7      some_end_operations();
8      return EXIT_SUCCESS;
9  }
```

Oubliez ça, c'était pour la culture...

### Saut conditionnel

Le processeur contient un registre appelé le registre des *drapeaux* (*flags* en anglais), ou registre de statut. Ces *drapeaux*, ou statuts, permettent d'indiquer des états du processeur vis-à-vis des actions qu'il a effectué. Chaque statut est stocké dans un bit du registre. On note  $N$  le nombre de bits du registre ( $N$  peut varier selon le processeur).

$$R_{flags} = s_{N-1} \dots s_0$$

Chaque instruction du processeur va modifier certains bits du registre. Par exemple, lors de l'addition de deux nombres non signés  $a$  et  $b$  stockés dans des registres 32 bits, si  $a + b > 2^{32} - 1$ , le drapeau dit de retenue (*carry flag* en anglais, noté CF dans la littérature) sera mis à 1.

9. Par exemple, les processeurs Intels utilisent des instructions de taille variable tandis que les processeurs ARM utilisent des instructions de taille fixe

Le saut conditionnel consiste à attribuer au registre *IP* une valeur précisée lorsque l'instruction est lue, à la condition qu'un certain drapeau du registre des drapeaux ait une certaine valeur (0 ou 1).<sup>10</sup> On peut ainsi vérifier :

- l'égalité entre deux nombres  $a$  et  $b$  (en vérifiant  $a - b = 0$  ou  $a - b \neq 0$ )
- l'inégalité entre deux nombres  $a$  et  $b$  (en vérifiant  $a - b > 0$  ou  $a - b < 0$ )
- *et caetera...*

Il est ainsi possible de former des conditions pour contrôler le flot d'exécution. L'exemple suivant est écrit en NASM pour un processeur Intel 64 bits. Il s'agit d'un langage assembleur, dont chaque instruction correspond donc de manière directe à une instruction exécutée par le processeur :

```

1  ; semicolons are comments
2
3  section .text ; beginning of the code
4  global _start
5
6  _start: ; same as main function in C
7
8  mov al, 200 ; al is a 8 bits register, and 2^8 = 256
9  add al, 100 ; 200 + 100 > 256, so CF = 1
10 jc end ; JC stands for Jump if Carry
11     ; all of the code here will be missed because CF != 0
12 end: ; following code ends the program running
13 mov rax, 231
14 mov rdi, 0
15 syscall

```

`end` est une étiquette choisie par le programmeur. Elle permet de nommer l'adresse mémoire d'une instruction. Elle est traduite à la "compilation" du code ci-dessus en adresse mémoire. Le code en NASM est équivalent au pseudo-code suivant :

---

#### Algorithme 4 : Traduction en pseudo-code

---

```

1   $al \leftarrow 200$ ;
2   $al \leftarrow al + 100$ ;
3  si la dernière opération a effectué un dépassement de capacité alors
4  |   Aller à la fin du programme;
5  sinon
6  |   Les instructions ici sont sautées
7  Fin du programme;

```

---

On pourrait alors imaginer la boucle suivante : (voir *boucle for* dans les structures abstraites)

```

1  ; ...
2  mov al, 10
3  simple_loop:
4  ; things to repeat 10 times
5  sub al, 1
6  jnc simple_loop ; jump if al - 1 >= 0
7  loop_end:
8  ; ...

```

10. voir [https://wiki.osdev.org/CPU\\_Registers\\_x86#EFLAGS\\_Register](https://wiki.osdev.org/CPU_Registers_x86#EFLAGS_Register) et [https://en.wikipedia.org/wiki/FLAGS\\_register](https://en.wikipedia.org/wiki/FLAGS_register) pour une liste des drapeaux des processeurs Intels

### 4.5.2 Structures complexes

Grâce aux structures élémentaires de contrôle de flot d'exécution, le langage C propose des structures de contrôle beaucoup plus intuitives dans leur utilisation :

- les conditions
  - le branchement `if-else`
  - l'aiguillage
- les boucles
  - la boucle à pré-condition
  - la boucle à post-condition
  - la boucle itérative

Les détails de la construction de ces structures abstraites ne seront pas décrits ici. Cela peut en effet dépendre du compilateur. <sup>11</sup>

Les *conditions* permettent au programme d'effectuer des choix quant à son flot d'exécution et ainsi de ne pas être entièrement séquentiel.

Les *boucles* permettent d'effectuer un bloc d'instructions plusieurs fois de suite. Par exemple, trouver le minimum d'un ensemble de nombres nécessite de parcourir tous ces nombres, et il serait un peu ennuyant d'avoir à d'écrire autant d'instructions qu'il y a d'éléments de cet ensemble, surtout si celui-ci contient des milliers voire des millions de nombres.

#### Branchement `if-else` :

Le branchement `if-else` consiste à n'exécuter une partie du code que si une certaine condition est satisfaite, et en exécuter une autre si cette condition n'est pas satisfaite. On observe ainsi le pseudo-code suivant : En langage C, cela donne :

---

**Algorithme 5** : Branchement conditionnel

---

```
1 si condition ≠ 0 alors
2   | Instructions si condition ≠ 0
3 sinon
4   | Instructions si condition = 0
5 Suite du programme
```

---

```
if (condition) {
    printf("La condition est verifiee.\n");
} else {
    printf("La condition n'est pas verifiee.\n");
}
printf("Suite du programme\n");
```

On remarque que `≠ 0` est implicite. Ainsi, les deux codes suivants sont équivalents :

---

11. Un exemple d'une possibilité d'implantation est cependant donné au dessus en langage assembleur

```
int a = 4;
int b = 6;
if (a - b) {
    printf("a different de b.");
} else {
    printf("a = b");
}
```

```
int a = 4;
int b = 6;
if (a - b != 0) {
    printf("a different de b.");
} else {
    printf("a = b");
}
```

On pourrait aussi écrire de manière plus lisible :

```
int a = 4;
int b = 6;
if (a != b) {
    printf("a different de b.");
} else {
    printf("a = b");
}
```

```
int a = 4;
int b = 6;
if ((a != b) != 0) { // or a != b != 0
    printf("a different de b.");
} else {
    printf("a = b");
}
```

Par ailleurs, l'instruction *else* est tout à fait optionnelle. L'omettre signifie simplement qu'il n'y a rien à exécuter si la condition n'est pas vérifiée :

```
int a = 4;
int b = 6;
if (a > b) { // or a != b != 0
    printf("?");
}
printf("texte par défaut"); // sera toujours exécuté
```

Il est également possible d'enchaîner plusieurs blocs de branchements conditionnels :

```
int a = 4;
int b = 6;
if (a > b) { // or a != b != 0
    printf("%d > %d\n", a, b);
} else if (a == b) {
    printf("egaux\n");
} else {
    printf("%d > %d\n", b, a);
}
printf("texte par défaut"); // always executed
```

### Opérateur ternaire

L'opérateur ternaire permet d'écrire certaines conditions de manière compacte :

```
variable = (condition) ? expression_1 : expression_2;
```

```
if (condition) {
    variable = expression_1;
} else {
    variable = expression_2;
}
```



Il y a stricte équivalence sémantique entre la syntaxe à gauche et celle à droite. Cela questionne évidemment les raisons de l'existence d'une telle syntaxe en C, mise à part la compacité de l'écriture.

La raison, assez technique,<sup>12</sup> sera simplement réduite à ses conséquences : l'utilisation d'une condition ternaire permet d'optimiser la vitesse d'exécution des instructions par l'UCC.

Un petit exemple :

```

1  // <math.h> needs you to compile this code with -lm option
2  // at the end of gcc command line :
3  // gcc main.c -o main -lm
4  #include <stdlib.h>
5  #include <math.h> // NaN : Not A Number, sqrt
6
7  int main() {
8      double a = ...; // any value
9      double b = ...;
10     double c = ...;
11
12     // Resolve the polynom equation
13     double d = b*b - 4*a*c;
14     double x1 = (d < 0) ? NAN : (-b - sqrt(d))/(2*a);
15     double x2 = (d < 0) ? NAN : (-b + sqrt(d))/(2*a);
16
17     return EXIT_SUCCESS;
18 }
```

## Aiguillage

L'aiguillage est une autre forme de branchement conditionnel. Il permet de diriger le flot d'exécution selon la valeur d'une variable entière. On peut vouloir associer à plusieurs nombres un bloc d'instructions différent. L'aiguillage permet de faire correspondre à des entiers des blocs d'instructions.

Il possède quelques avantages et inconvénients :

### ■ avantages :

- très lisible
- permet une optimisation plus poussée du code par le compilateur
- facilite l'écriture des conditions sur des structures complexes (parce-que personne n'a envie d'écrire 40 if-else enchaînés les uns dans les autres)

### ■ inconvénients :

- ne teste que des égalités
- les valeurs à évaluer doivent être écrites en dur

Cet aiguillage sert en général à effectuer un choix, qu'il s'agisse du parcours d'une structure informatique complexe<sup>13</sup>, ou d'un simple menu dans l'application d'un restaurant.

En langage C, l'aiguillage est introduit par le mot-clé `switch`. Les mot-clés `case` et `break` viennent avec.

12. Pour les curieux, voir [https://drive.google.com/file/d/1bkLb30ByL\\_UaBNz-ksr03WliZ6mnuiy-/view?usp=drive\\_link](https://drive.google.com/file/d/1bkLb30ByL_UaBNz-ksr03WliZ6mnuiy-/view?usp=drive_link)

13. Patience et longueur de temps font plus que force ni que rage, a écrit un jour... Faut attendre un peu avant de faire des trucs impressionnants, z'inquiétez pas ça va arriver

```

unsigned int x;
switch (x) {
case 0:
    printf("Choix 0");
    // and other instructions
    break;
case 1:
case 2:
    printf("Choix 1 ou 2");
    // and other instructions
    break;
case 3:
    printf("Choix 3 puis 4");
    // and other instructions
case 4:
    printf("Choix 4");
    // and other instructions
    break;
default:
    printf("Autre");
    // and other instructions
    break;
}
printf("apres le switch");

```

L'exemple ci-dessus se veut exhaustif des cas possibles de l'instruction `switch`. On observe que le flot d'exécution va "sauter" au `case` correspondant à la valeur de  $x$ , et à `default` si cette valeur n'est pas présente dans l'aiguillage. *L'exécution redevient séquentielle à partir de cet instant!* Une erreur régulièrement faite par les débutants est de penser qu'à la fin de l'exécution du `case`, le flot vient tout de suite après le `switch`. Cela n'est pas vrai sans le mot-clé `break` qui "casse" le bloc d'instructions en cours d'exécution et saute à la fin.

Dans l'exemple ci-dessus, l'exécution du code pour  $x = 3$  s'ensuit immédiatement par l'exécution du code pour  $x = 4$  car il n'y a pas de `break`. De la même manière, si  $x = 1$ , le flot va sauter à la ligne `case 1` puis va séquentiellement avancer au bloc d'instructions pour  $x = 2$ .

### Boucle à pré-condition (ou boucle *while*) :

La boucle à pré-condition suit le pseudo-code suivant :

---

#### Algorithme 6 : Boucle *While*

---

```

1 tant que condition ≠ 0 faire
2   | Instructions

```

---

Il s'agit d'une boucle qui vérifie *avant* l'exécution du bloc d'instructions qu'une certaine condition est vérifiée, et exécute ainsi le bloc d'instruction jusqu'à ce que la condition ne soit plus vérifiée. C'est le contenu de ce bloc d'instructions qui va amener à ce que la condition ne soit plus satisfaite.

```

while (condition) {
    // instructions block executed while condition != 0
}

```

Comme pour le branchement if-else, la différence à 0 de la condition est implicite.

On peut par exemple imaginer le code suivant :

```

1  #include <stdio.h>
2  #include <stdlib.h>
3
4  #define N ...
5
6  int main() {
7      int u = 3;
8      unsigned int n = 0;
9      while (n < N) {
10         u = 3*u*u - 5*u + 2;
11         n++; // n++ <=> n += 1
12     }
13     printf("u(%u) = %d\n", n, u);
14     return EXIT_SUCCESS;
15 }
```

Ce programme calcule l'élément de rang  $N$  précisé de la suite  $(u_n)_{n \in \mathbb{N}}$  définie telle que :

$$\begin{cases} u_0 &= 3 \\ \forall n \in \mathbb{N}, u_{n+1} &= 3u_n^2 - 5u_n + 2 \end{cases}$$

**Exercice 20 ([COURS]).** Modifier le programme donné en exemple pour qu'il affiche toutes les valeurs  $u_1, \dots, u_N$ .

Expliquer pourquoi les valeurs deviennent fausses à partir d'un certain rang que l'on précisera.

### Boucle à post-condition (ou boucle *do-while*)

La boucle à post-condition est pratiquement identique à la boucle à pré-condition, à une nuance qui a son importance : la condition est effectuée à la fin de l'exécution de la boucle. Cela signifie qu'au contraire de la boucle à pré-condition, le bloc d'instructions d'une boucle à post-condition s'exécute au moins une fois. Les deux codes suivants sont donc équivalents :

<pre>do {     // Instructions block } while (condition);</pre>	<pre>// Instructions block while (condition) {     // Same instructions block }</pre>
--	---

Ce type de boucle est extrêmement utile quand on veut être certain qu'une action est effectuée au minimum une fois. Par exemple, il peut s'agir de demander à un utilisateur d'entrer des nombres jusqu'à ce que l'utilisateur entre le nombre 0.

### Boucle itérative (ou boucle *for*)

La boucle itérative est un cas particulier de boucle à pré-condition. Elle est construite selon le format suivant en langage C :

```
for (initial; condition; pas) {
    // Instructions block
}
```

et est équivalente au pseudo-code suivant :

---

**Algorithme 7 : Boucle *for***


---

```

1 Exécuter l'instruction initiale;
2 tant que condition faire
3   | Instructions ;                               /* Bloc d'instructions */
4   | Instruction de pas ;                         /* Puis le pas */

```

---

L'idée est en générale d'utiliser l'instruction de *pas* pour parcourir une structure. Il s'agit d'une itération sur un élément de parcours, raison qui donne son nom à cette syntaxe de boucle. Son application va être vue dans la section sur les *tableaux*.

**Remarque 1 :** *initial* et *pas* ne peuvent être au maximum constitués que d'une instruction.

**Remarque 2 :** *initial*, *condition* et *pas* sont optionnels, tout comme le bloc d'instruction. Si la condition est oubliée, elle est considérée comme toujours vraie.

Quelques exemples :

```

// Infinite loop :
for (;;) {

}

// Infinite loop incrementing a variable :
for (int a = 0;;a++) { // a++ <=> a += 1

}

// Finite loop which starts printing a text :
int a;
for (printf("Initialement, a = %d\n", a = 1); a < 10; printf("À présent : a = %d\n", ++a));

```

Mais pourquoi on utilise jamais **goto** ? C'est plus basique donc c'est mieux ?



Plus sérieusement, il est vraiment recommandé de ne jamais faire cela. En effet, ce genre de saut est effectué sans contrôle (donc augmente les risques de bogues) et permet moins d'optimisations de la part du compilateur car son fonctionnement ne possède pas de logique intrinsèque, au contraire des boucles *for* et *while* qui suivent des règles contraintes par des conditions.

### 4.5.3 Exercices

**Exercice 21 (Question d'âge).** Écrire un programme C contenant une variable *age* de type `signed char` dont la valeur est initialisée. Le programme affiche :

- “Soyez patient(e), votre tour arrive” si  $age < 0$
- “Mineur” si  $0 \leq age < 18$
- “Tout juste majeur” si  $age = 18$
- “Majeur” si  $18 < age \leq 100$
- “Et la surpopulation alors?” si  $100 < age$ <sup>14</sup>

Essayer de minimiser le nombre de branchements conditionnels (objectif : trois `if` maximum, et seule la phrase correcte doit s'afficher<sup>15</sup>).

**Exercice 22 (Suite de Fibonacci).** Écrire un programme qui calcule les  $N$  premiers éléments de la suite de Fibonacci  $(f_n)_{n \in \mathbb{N}}$  définie telle que :

$$\begin{cases} f_0 &= 0 \\ f_1 &= 1 \\ \forall n \in \mathbb{N}, & f_{n+2} = f_{n+1} + f_n \end{cases}$$

**Exercice 23 (Nombres premiers).** Écrire un programme qui demande à l'utilisateur de saisir un nombre  $N$  et vérifie si  $N$  est premier. On essaiera d'avoir un maximum de  $\sqrt{N}$  tours de boucle. On justifiera la correction du programme.<sup>16</sup>



La modulation d'un programme est probablement le point le plus important qu'il puisse y avoir en programmation. Pour de très petits projets, de très petites applications, il est possible de programmer toute l'application d'un seul tenant. Mais au fur-et-à-mesure que l'application grandit, il devient très compliqué, voire impossible de modifier correctement le programme sans introduire une quantité phénoménale de bogues en tous genres.

L'idée est alors simple : il faut compartimenter, diviser le programme en sous-programmes qui effectuent chacun une action bien précise. C'est ensuite la fonction *main* qui va s'occuper d'agencer ces sous-programmes. Le programme vu de manière globale devient alors :

- très flexible, puisque chaque fonctionnalité est identifiée à un sous-programme précis et qu'il suffit d'appeler dans la fonction *main* les sous-programmes utiles
- facilement déboguable puisqu'il suffit de corriger chaque sous-programme, tous très simples, et de corriger l'agencement de ces sous-programmes dont on sait que chacun est correct

Nous allons voir un cas particulier de sous-programme appelé la *routine*<sup>17</sup>, dont il n'existe que deux formes<sup>18</sup> en langage C :

- la procédure : effectue une action dépendante de ses entrées et ne renvoie rien

14. Enfin moi je dis ça je dis rien...)\*

15. Juré, c'est possible! Croix de bois, croix de fer, si j'mens j'vais en enfer!

16. C'est-à-dire une démonstration que ce programme résout bien le problème.

17. Un autre type de sous-programme notable apparaissant par exemple en Python (et en C) est la *coroutine*, qui possède la propriété de pouvoir être suspendu au cours de son exécution puis reprise là où elle s'est arrêtée.

18. Nativement. Il est possible d'en simuler d'autres, comme les méthodes par exemple en programmation orientée objet, qui ont comme particularité d'être partie d'une entité appelée un objet.

■ la fonction : effectue une action dépendante de ses entrées et renvoie une valeur de sortie  
Ces deux formes en langage C se voient écrites de manière semblable par les syntaxes suivante :

<pre>void procedure_name(parametres) {     // Instructions block     return; }</pre>	<pre>TYPE function_name(parametres) {     // Instructions block     return VALUE; // VALUE of type TYPE }</pre>
--	---

Les paramètres sont indiqués comme des variables non initialisées, séparées par des virgules :

```
int addition(int a, int b) {
    int c = a + b;
    return c;
}
```

Il est alors possible d'appeler cette fonction *dans la suite du programme*. En effet, les instructions précédentes à la définition de la procédure/fonction n'en ont pas connaissance. C'est d'ailleurs pour cette raison que les directives `#include` sont écrites au début du programme.

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  int addition(int a, int b) {
5      int c = a + b;
6      return c;
7  }
8
9  int main() {
10     printf("%d + %d = %d\n", 2, 3, addition(2, 3));
11     return EXIT_SUCCESS;
12 }
```

Il faut donc que les routines soient définies *avant* tous leurs appels. Pour permettre une plus grande flexibilité dans l'écriture du code, il est cependant possible d'*annoncer* la définition d'une routine avant de l'écrire réellement, grâce aux *prototypes*.

**Remarque :** L'instruction `return` peut en fait être placé n'importe où dans la routine. Elle y mettra fin :

```
void afficher_diff_0(int x) {
    if (x == 0) {
        return; // ends the routine
    }
    printf("x != 0\n"); // never executed
}
```

### 4.6.1 Signature et prototype

#### Définition 19 : Prototype

Le prototype d'une routine est la donnée de :

- son type de retour
- son nom
- les types de ses paramètres
- l'ordre des paramètres

En fait, un prototype se déclare en C comme une routine absente de code :

```
int addition(int a, int b); // Prototype of the 'addition' function

int addition(int a, int b) { // Declaration of the 'addition' function
    return a + b;
}
```

Le prototype permet de déclarer l'existence de la routine avant sa définition pour que le compilateur la reconnaisse avant sa définition. Il est ainsi possible de l'appeler avant la définition de la routine :

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  // Both indicates the existence of the 'addition' function to the compiler
5  int addition(int a, int b); // Prototype of the 'addition' function
6  // OR
7  int addition(int, int); // Parameters names are not necessary in the prototype
8
9  int main() {
10     printf("%d + %d = %d", 2, 3, addition(2, 3));
11     return EXIT_SUCCESS;
12 }
13
14 int addition(int a, int b) { // Declaration of the 'addition' function
15     return a + b;
16 }
```

#### Définition 20 : Signature

La signature d'une routine consiste en son nom, les types de ses paramètres et l'ordre de ceux-ci. À la différence du prototype, le type de retour de la routine n'est pas donné.

**Remarque :** Si le concept de *signature* de routine est général à une grande quantité de langages, celui de *prototype* de routine tel que définit ci-avant est spécifique aux langages C, C++ et à ceux dérivant de C et C++.

## 4.6.2 Note relative à la copie des arguments

Notons que les identifiants à l'intérieur d'une routine évoluent dans un *espace local* :

```

1  #include <stdio.h>
2  #include <stdlib.h>
3
4  void proc(int c) {
5      v = 48; // ERROR : v not declared in this function
6      return c;
7  }
8
9  int main() {
10     int v = 5;
11     proc(v+1);
12     v = c; // ERROR : c not declared in this function
13     return EXIT_SUCCESS;
14 }
```

En particulier :

```

1  #include <stdlib.h>
2
3  int polynome(char p) { // Declaration of the 'addition' function
4      p = (p*p - p + 1);
5      return (int)p;
6  }
7  int main() {
8      char p = 3;
9      polynome(p);
10     return EXIT_SUCCESS;
11 }
```

ne modifie jamais la valeur de la variable *p* située dans le *main*.

On arrive alors à la différenciation entre la variable passée en *argument* de la routine et le *paramètre* de la routine :

### Définition 21 : Argument

Un argument est la valeur passée lors de l'appel de la routine pour un certain paramètre.

Ainsi, dans le code suivant :

```

1  #include <stdlib.h>
2
3  int polynome(char p) { // Declaration of the 'addition' function
4      return (int)(p*p - p + 1);
5  }
6  int main() {
7      int a = 3;
8      polynome(a);
9      return EXIT_SUCCESS;
10 }
```



$p$  désigne le paramètre de la fonction `polynome` tandis que  $a$  est l'argument passé au paramètre  $p$ . On dit aussi que  $a$  est passé en argument à la fonction `polynome` pour le paramètre  $p$ .

Pour être techniquement plus précis, la valeur de  $a$  est copiée en mémoire au moment de son passage à `polynome`. Ainsi, la modification de  $p$  dans  $f$  ne modifie jamais la valeur de  $a$  :

```

1  #include <stdlib.h>
2
3  int polynome(char p) { // Declaration of the 'addition' function
4      p = (p*p - p + 1);
5      return p; // implicit static casting
6  }
7  int main() {
8      int a = 3;
9      polynome(a);
10     if (a == 3) {
11         printf("a non modifie\n");
12     }
13     a = polynome(a); // a = 3*3 - 3 + 1 = 7
14     if (a != 3) {
15         printf("a modifie\n");
16     }
17     return EXIT_SUCCESS;
18 }
```

On trouve alors une limitation à l'action des fonctions, puisque celles-ci semblent ne pouvoir agir que sur une unique valeur, celle de retour. Grâce aux pointeurs, on pourra cependant abroger cette limitation...

### 4.6.3 La pile d'exécution

Lorsque le système charge en mémoire un programme informatique et l'exécute, il ne contient pas que le code. Il faut aussi pouvoir stocker les variables et enregistrer l'historique des appels de routines. En effet, il faut bien gérer en mémoire le passage d'un argument à une routine, et également stocker l'adresse mémoire à laquelle le `return` doit revenir. On pourrait imaginer le code suivant :

```

1  #include <stdio.h>
2  #include <stdlib.h>
3
4  #define X ... // any value
5  #define Y ... // any value
6
7  unsigned int addition(int a, int b) {
8      for (; a-- > 0; b++);
9      return b;
10 }
11
12 unsigned int multiply(unsigned int a, unsigned int b) {
13     int sum = 0;
14     for (; a > 0; a--) {
15         sum = addition(sum, b);
16     }
17     return sum;
18 }
```

```
19 |
20 | int main() {
21 |     printf("%u x %u = %u\n", X, Y, multiply(X, Y));
22 |     return EXIT_SUCCESS;
23 | }
```

La *pile d'exécution* est une zone mémoire allouée à un processus qui lui permet de stocker les paramètres de fonctions et les variables internes à la fonction. Il faut voir cette pile comme une pile d'assiettes (qui sont les paramètres, etc...). Les éléments ne peuvent être ajoutés à la pile d'exécution qu'en les empilant ou en les dépilant du dessus de la pile.

Ainsi, l'appel à `multiply` va *empiler* l'adresse mémoire de l'instruction suivant l'appel à la fonction, va empiler les arguments sur la pile puis va sauter à l'adresse mémoire de la fonction `multiply`. Cela sera réitéré pour chaque appel à la fonction `addition`.

La pile contient *entre autre* :

- la mémorisation des appels de routine, des arguments des paramètres lors des appels et de l'adresse de retour. Il faut bien que le programme sache où revenir lors du `return`
- les variables assignées dans les routines : celles-ci sont *empilées* lors de leur assignation

Les variables contenues dans la pile ne sont accessibles que dans la routine qui les a créés. Par ailleurs, dès l'instant où une routine exécute l'instruction `return`, les variables utilisées dans cette routine sont libérées, c'est-à-dire que l'espace mémoire qui était réservé pour elles devient à nouveau libre de contenir n'importe quelle autre valeur (opération de *dépilage* de la pile d'exécution).

C'est pour cette raison qu'on essaye de compartimenter les programmes en une multitude de routines, surtout lorsque les calculs effectués par ces routines utilisent beaucoup de variables temporaires. En effet, au lieu de stocker toutes ces variables dans l'espace mémoire du programme principal même lorsqu'elles sont inutiles, l'espace mémoire n'est utilisé pour stocker ces variables que lors de l'exécution de la routine.

#### Remarque :

L'allocation de mémoire de variables est entièrement gérée par le compilateur. En effet, les variables créées dans un programme sont analysées avant la compilation et le programme en binaire finale sera en vérité une alternative au code écrit par le programmeur qui sécurisera l'accès à la mémoire pour éviter certains bogues.

Par exemple, la taille de la pile au chargement du programme est généralement de taille fixe (cela dépend du système d'exploitation, mais on considère ici des systèmes classiques comme Linux ou Windows). Ainsi, empiler de trop nombreux appels de routines les uns sur les autres peut amener à une erreur de dépassement de pile (*stack overflow* en anglais). De la même façon, il ne devrait pas être possible d'allouer de trop nombreuses variables sur la pile dans une seule fonction. Certaines techniques permettent toutefois d'éviter des erreurs à ce niveau en stockant les variables les plus anciennes/les moins utilisées ailleurs que sur la pile, comme par exemple sur le tas ou même sur le disque dur (pour d'immenses programmes) malgré la perte de vitesse que cela engendre. On suppose en effet que ces variables ne sont utilisées que rarement et qu'une baisse de vitesse pour leur accès est négligeable.

Il est cependant possible de gérer par soi-même l'allocation des variables en mémoire. C'est ce qu'on appelle l'*allocation dynamique*. La pile n'est jamais<sup>19</sup> utilisée pour ce type d'allocation.

---

19. à deux trois détails près...

### 4.6.4 Exercices

#### Exercice 24 (Encore Fibonacci).

Écrire une fonction de prototype `unsigned int fibo(int n);` qui à  $n$  renvoie  $f_n$  où  $(f_n)_{n \in \mathbb{N}}$  est définie telle que :

$$\begin{cases} f_0 &= 0 \\ f_1 &= 1 \\ \forall n \in \mathbb{N}, & f_{n+2} = f_{n+1} + f_n \end{cases}$$

Tester ensuite cette fonction pour toutes les valeurs de  $n \in \llbracket 0; 100 \rrbracket$ .

#### Exercice 25 (Puissances entières). Écrire une routine équivalente à la fonction :

$$\begin{aligned} f : \llbracket -2^{31}, 2^{31} \rrbracket \times \llbracket 0, 2^{32} \rrbracket &\rightarrow \mathbb{Z} \\ (x, n) &\mapsto x^n \end{aligned}$$

Dans quelle mesure cette fonction est-elle correcte ?



Probablement la section la plus simple de ce document, l'utilisation des pointeurs n'induita jamais de bogues étranges et inexpliqués dans un programme C. Tout un chacun peut affirmer sans hésitation qu'il n'y a rien de plus *trivial* que la programmation avec des pointeurs.

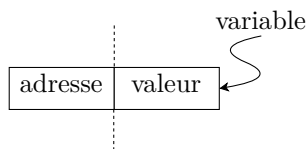
Tout va bien se passer...

Maintenant que le lecteur est *rassuré*, commençons...

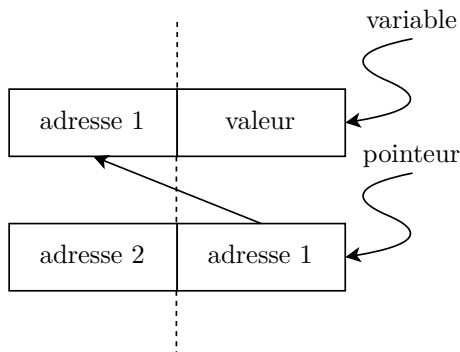
**Rappel :** Une variable peut être approximée comme la collection de trois informations :

- une étiquette, aussi appelée le nom de la variable
- une adresse mémoire
- une donnée sur  $N$  octets à cette adresse

Visuellement :



L'idée des pointeurs est simple : au lieu de stocker directement une valeur, on stocke l'adresse mémoire d'une autre variable :



Les pointeurs sont fondamentaux en programmation. En particulier, ils permettent une flexibilité très importante des programmes écrits. En effet, une variable est définie dans un certain espace. Ainsi, une variable d'une routine est locale à cette routine, et n'existe pas en dehors. Une adresse mémoire peut potentiellement être utilisée hors de routines spécifiques. Les possibilités des routines sont donc décuplées puisque celles-ci peuvent interagir avec leur environnement autrement que par la valeur de retour (dans le cas d'une fonction). Cela donne également une plus grande importance aux procédures.

Cela amène cependant un inconvénient : en perdant un contrôle absolu sur l'existence de données, il devient possible *par erreur* d'accéder à des zones mémoires qui ne sont plus utilisés. Il s'agit là de l'erreur la plus classique en programmation en C <sup>20</sup>.

On dispose pour la manipulation des pointeurs de deux opérateurs en langage C : l'étoile \* et l'esperluette &. On a également la constante NULL qui représente un pointeur ne pointant sur rien du tout. Un pointeur est défini selon la syntaxe suivante :

```
TYPE* ptr;
```

On sait alors que *ptr* contiendra l'adresse d'autres variables.

Viens alors l'utilisation de l'esperluette. Ce caractère en tant qu'opérateur binaire permet de signifier l'opération *ET*, qu'elle soit logique ou bit-à-bit. Il est cependant possible de l'utiliser comme opérateur unaire. Sa signification est alors tout autre. *&v* renvoie l'adresse mémoire à laquelle se situe la variable *v*. On peut donc écrire :

```
TYPE v = ...;
TYPE* ptr = &v;
```

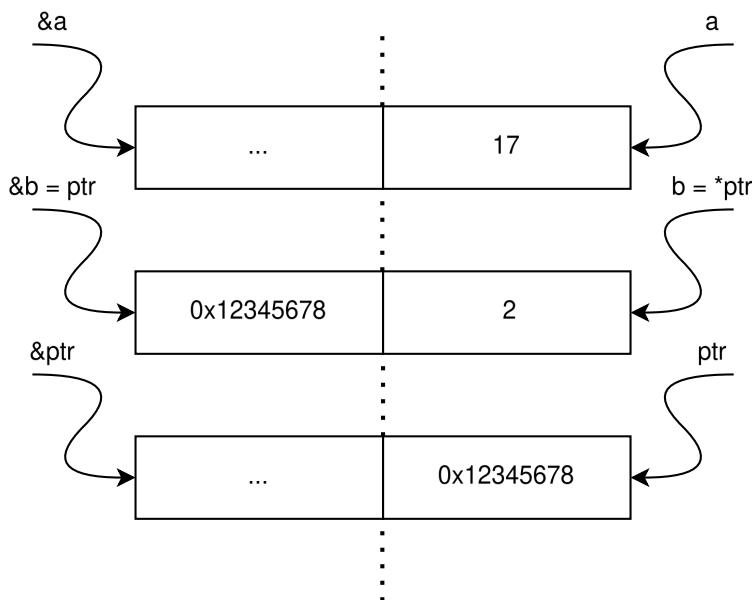
On remarquera que *ptr* doit être un pointeur de même type que le type de *v*. En effet, il devient possible depuis le pointeur *ptr* de lire et de modifier la valeur de *v*. La pleine connaissance de *v* est donc nécessaire, et l'information de sa taille est stockée comme la taille du pointeur.

C'est l'opérateur \* qui lui aussi, lorsqu'il est utilisé comme opérateur unaire, change de signification. *\*ptr* désigne la valeur de *v*. Modifier la valeur de *\*ptr* modifie donc la valeur de *v*. En langage C cela donne :

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  int main() {
5      int a = 58;
6      int b = 67;
7      int* ptr = &a; // points on a
8      *ptr = 17; // same as a = 17
9      printf("a = %d\n", a);
10     ptr = &b; // points on b
11     *ptr = 2; // same as b = 2
12     printf("b = %d", b);
13     return EXIT_SUCCESS;
14 }
```

Après exécution du code, on a le schéma suivant :

20. traduite par le fameux message d'erreur *Segmentation Fault*



L'utilisation de `*` pour accéder *indirectement* à la valeur d'une variable a donné à cet opérateur unaire le nom d'*opérateur d'indirection*<sup>21</sup>.

On peut décider également qu'un pointeur ne pointe sur rien :

```
int *pointeur = NULL; // points on nothing
```

**Remarque/Avertissement :** Lorsqu'un pointeur ne pointe sur rien, ou pointe sur une adresse dont l'accès n'est pas autorisé, c'est-à-dire qui est potentiellement utilisé par un autre programme, l'accès à la valeur en mémoire conduira très probablement le système d'exploitation, pour des raisons de sécurité, à arrêter prématurément l'exécution du programme en indiquant le code d'erreur `-11` dit d'*erreur de segmentation* (*segmentation fault* en anglais) :

```
int *pointeur = NULL; // points on nothing
char *pointeur2 = (char *)0x12345678; // a priori an invalid address

printf("%d\n", *pointeur); // -11 error code
printf("%d\n", *pointeur2); // -11 error code
```

### 4.7.1 Formatage en chaîne de caractères

Une adresse mémoire peut être affichée grâce au formatage des chaînes de caractères. Il faut pour cela utiliser le caractère spécial `"%p"` :

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  int main() {
5      int a = 58;
```

21. En toute originalité.

```

6   int b = 67;
7   int* ptr = &a; // points on a
8   printf("&a = %p et a = %d\n", &a, a);
9   printf("&ptr = %p, ptr = %p et *ptr = %d\n", &ptr, ptr, *ptr);
10  ptr = &b; // pointe vers b
11  printf("&b = %p et b = %d\n", &b, b);
12  printf("&ptr = %p, ptr = %p et *ptr = %d\n", &ptr, ptr, *ptr);
13  return EXIT_SUCCESS;
14 }

```

## 4.7.2 Les pointeurs en paramètres de routines

Venons en à une utilité possible des pointeurs : le passage d'arguments à des routines. La limitation des routines est qu'elles ne peuvent agir sur le reste du programme que par au maximum une unique valeur, celle de retour dans le cas des fonctions, aucune dans le cas des procédures. Imaginons par exemple une routine qui doit effectuer l'échange des valeurs de deux variables. Cela n'est pas possible tant que la routine n'a aucun moyen d'action direct sur les variables elles-même.

C'est-ici qu'interviennent les pointeurs :

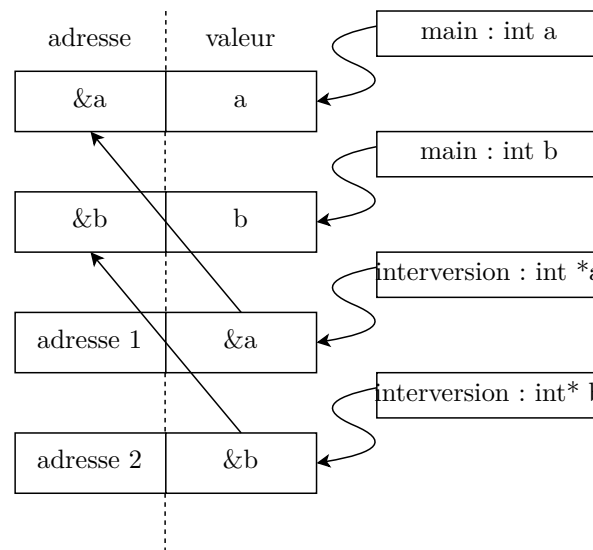
```

1   #include <stdio.h>
2   #include <stdlib.h>
3
4   void interversion(int* a, int* b) {
5       int tmp = *a;
6       *a = *b;
7       *b = tmp;
8   }
9
10  int main() {
11      int a = 5;
12      int b = 7;
13      interversion(&a, &b);
14      printf("a = %d et b = %d\n", a, b);
15      return EXIT_SUCCESS;
16  }

```

Analysons un peu cette procédure. Les paramètres sont de type pointeurs sur des int. *a* et *b* à l'intérieur de la procédure sont donc les adresses des arguments passés à la procédure.

Ainsi, on observe le schéma suivant durant l'appel de la procédure `interversion(&a, &b)` :



Alors, *\*a* dans *intversion* est la valeur de *a* dans la fonction *main* et *\*b* dans *intversion* est la valeur de *b* dans la fonction *main*. On procède alors à une interversion avec effet de bord, comme dans l'**Exercice 10**.

### 4.7.3 Projection de type

Un point important n'a pas été abordé sur les pointeurs<sup>22</sup> : la projection de type.

Considérons le code suivant :

```

1  #include <stdio.h>
2
3  int main() {
4      short int a = 7187;
5      short int *int_ptr = &a;
6      printf("%d\n", *int_ptr);
7      char *char_ptr = (char *)int_ptr; // explicit static casting
8      printf("%d\n", *char_ptr);
9      printf("%d\n", *(char_ptr + 1));
10 }

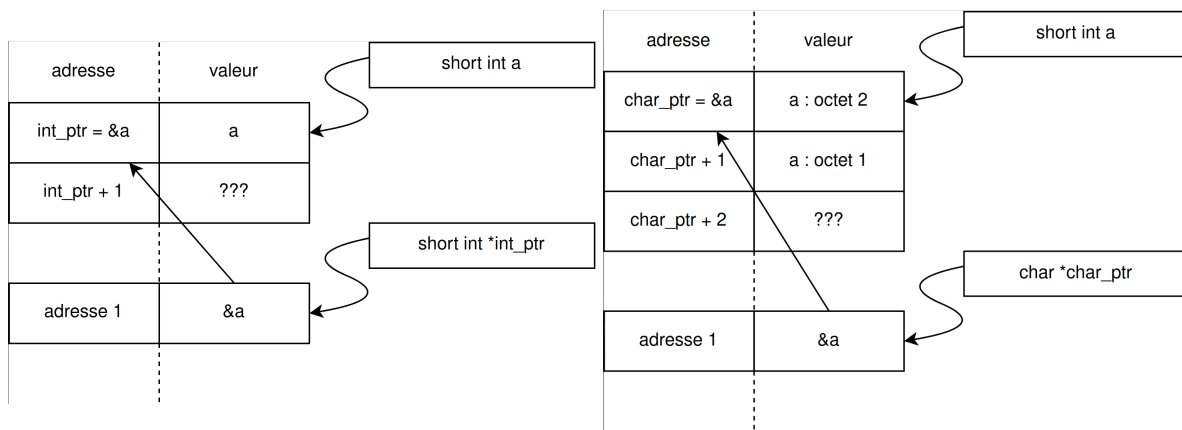
```

On observe que :

- *\*char\_ptr* = 19
- *\*(char\_ptr + 1)* = 28
- $a = 7187 = (0001110000010011)_2 = 0x1C13 = 28 * 256 + 19$

La projection de type en *char\**, qui pointe vers des valeurs de 1 octet, permet d'accéder aux valeurs de chacun des octets pris séparément d'un mot binaire. En effet, incrémenter un pointeur le fait pointer vers l'adresse de la case mémoire suivante, *de même taille que le type du pointeur* ! On observe les deux cas suivants :

22. Sans jeux de mots aucun... puisque les jeux de mots laids font des gambettes;)



Ainsi, toutes les égalités du code suivant sont vraies :

```
char *char_ptr = char_ptr;
short int *sint_ptr = char_ptr;
int *int_ptr = char_ptr;
long int *lint_ptr = char_ptr;

// every following equality test is true :
char_ptr == sint_ptr;
char_ptr == int_ptr;
char_ptr == lint_ptr;
char_ptr + 4 == int_ptr + 1;
char_ptr + 2 == sint_ptr + 1;
sint_ptr + 2 == int_ptr + 1;
int_ptr + 2 == lint_ptr + 1;
```

Toutefois, un détail reste troublant... l'inversion des octets 1 et 2 sur le schéma ci-dessus, que l'on observe à l'exécution du programme sur n'importe quel ordinateur possédant un UCC *Intel*. Aucune reformulation ici, l'explication sur Wikipédia suffit : <https://fr.wikipedia.org/wiki/Boutisme><sup>23</sup>.

**Notation :** Avant de partir dans quelques exercices, voyons simplement une facilité d'écriture du langage C, valide pour tout pointeur `ptr` et tout entier `i` :

```
1 // result of the equality test is always true :
2 ptr[i] == *(ptr + i);
```

Plus un petit *trick* inutile mais rigolo : comme l'addition est commutative on peut aussi écrire :

```
1 // result of the equality test is always true :
2 (i - 1)[ptr + 1] == *(ptr + i);
```

Les raisons de l'existence de cette facilité d'écriture seront détaillées dans la section sur les tableaux.

23. Ce document n'est pas un cours d'histoire de l'informatique :'



### Le pointeur quelconque : `void*`

Un dernier cas d'utilisation du projecteur de type est celui du pointeur quelconque `void*`. En soi, un pointeur contient seulement une adresse, et il peut être intéressant dans certains programmes (voir la section sur les tableaux dynamiques) de ne conserver que l'adresse et d'"oublier" le reste, c'est-à-dire le type du pointeur.

Le type spécial `void*` représente exactement cela : un pointeur non typé vers une adresse.

La projection de type sur un pointeur non typé `void*` permet de faire retrouver au pointeur un type :

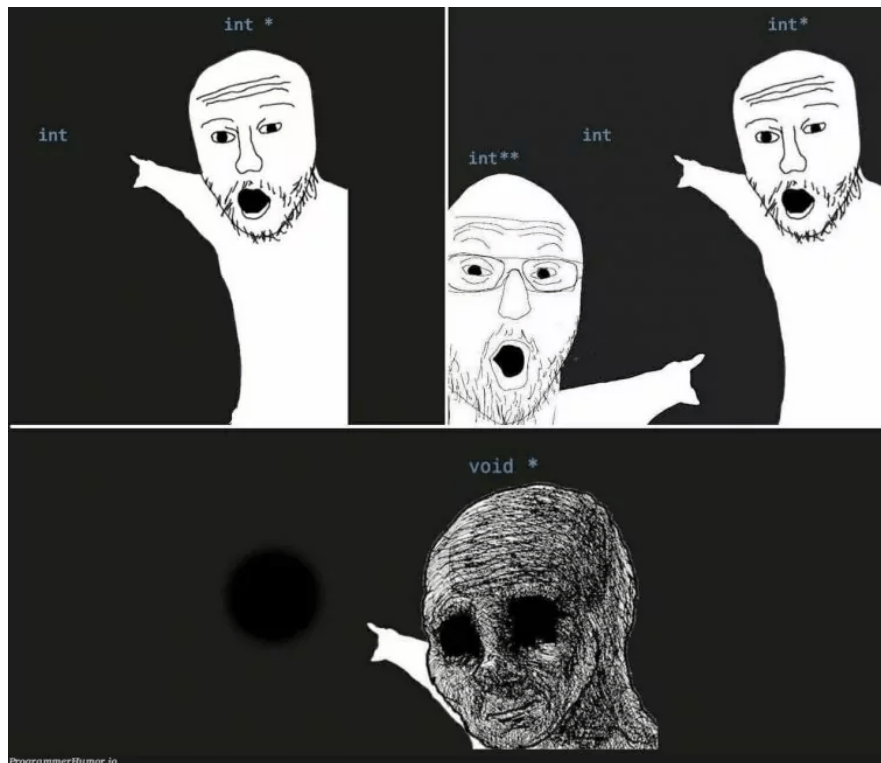
```
1 void *pointeur_quelconque = NULL;
2
3 // explicit static casting to avoid warnings at compile time :
4 int *int_ptr = (int*)pointeur_quelconque;
```

Par ailleurs, l'addition sur un pointeur non typé est "classique" :

```
1 void *pointeur_quelconque = NULL;
2 printf("%p\n", pointeur_quelconque + 5); // prints 0x5
```

Si pour l'instant l'utilisation de tels pointeurs peut rester assez absconse, cela s'éclairera dans le chapitre sur les concepts avancés du langage C. On trouvera déjà un premier exemple dans la section sur les tableaux dynamiques qui montre une application essentielle des pointeurs.

#### 4.7.4 Pointeurs itérés



### 4.7.5 Exercices

**Exercice 26 (Quelques procédures inutiles pour devenir un bot efficace).** Programmer en C les procédures suivantes, et les tester sur quelques valeurs :

- `void mov(int *x, int val);` : assigne à la variable pointée par  $x$  la valeur  $val$
- `void add(int *a, int *b);` : assigne à la variable pointée par  $a$  la somme des valeurs des variables pointées par  $a$  et  $b$
- `void mul(int *y, int *z);` : assigne à la variable pointée par  $y$  le produit des valeurs des variables pointées par  $y$  et  $z$
- `void pow(int *x, int n);` : assigne à la variable pointée par  $x$  la valeur de la variable pointée par  $x$  à la puissance  $n$

**Exercice 27 (Interversion sans effet de bord (3)).** Écrire une procédure `void swap(int *a, int *b);` qui échange les valeurs des variables pointées par  $a$  et  $b$  sans utiliser de variable temporaire. Quel est le comportement du programme si  $x = y$ ? (c'est-à-dire que la même adresse est passé en argument pour les deux paramètres)

Si cela induit une erreur de comportement de la procédure, corriger cette erreur.

**Exercice 28 (Distance de Manhattan).**

1. Écrire une fonction `double coords_to_point(float x, float y);` qui stocke les coordonnées d'un point dans une seule variable de type `double` à l'aide d'une projection de type.
2. Écrire une fonction `double subtract(double pA, double pB);` qui à deux points  $p_A = (x_A, y_A)$  et  $p_B = (x_B, y_B)$  renvoie :

$$p_A - p_B = (x_A - x_B, y_A - y_B)$$

3. Écrire une fonction `float norme1(double p)` qui calcule la norme 1 d'un point  $p = (x, y)$  :

$$\|\vec{p}\| = |x| + |y|$$

4. En déduire une fonction `float distance(double pA, double pB);` qui renvoie la distance entre deux points  $p_A$  et  $p_B$  associée à la norme 1. Cette distance est appelée *distance de Manhattan*<sup>24</sup>

**Remarque :** On pourra éviter les avertissements à la compilation en effectuant des projections de type explicites.

---

<sup>24</sup>. Souvent utilisée en informatique du fait de sa vitesse d'exécution.

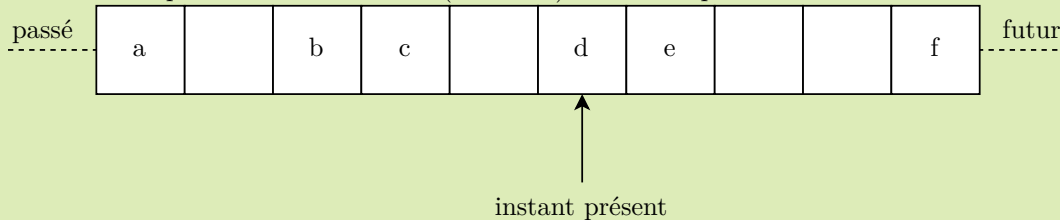


## INTERAGIR AVEC LES FLUX STANDARDS



### Définition 22 : Flux

En informatique, un flux (*stream* en anglais), est une séquence infinie discrète d'éléments indicés par le temps. On peut y penser comme une bande transporteuse d'objets quelconques, dont on n'aurait accès qu'à un certain élément (ou à rien) à l'instant présent :



Il s'agit d'un concept *purement abstrait* qui permet de penser à certains comportements en ignorant les détails techniques. Ces détails techniques seront vus ultérieurement avec la manipulation des *flux de fichiers*.

Un exemple “utile” dans notre cas est celui des flux d'entrée et de sortie d'un ordinateur. On peut penser par exemple :

- au flux d'images d'une vidéo en *streaming*
- au flux de sortie d'un texte sur un terminal
- au flux d'entrée du clavier
- au flux des positions du curseur, manipulé par une souris ou un pad
- au flux d'une communication internet entre un client et un serveur
- au flux résultant de l'écriture et de la lecture d'un fichier sur un disque
- au flux résultant de l'écriture et de la lecture d'une variable en mémoire
- etc. . .

La manipulation des flux standards passe par la bibliothèque `<stdio.h>`, qui est la bibliothèque standard des entrées et sorties.

De manière générale, un flux peut être de deux types :

- *textuel* : suites de caractères terminées par le caractère de fin de ligne ‘\n’ qui forment des lignes
- *binaire* : suites de mots binaires regroupés en blocs de multiplets

**Remarque :** On peut aussi considérer qu'un flux textuel est un flux binaire dans le sens où du texte est représenté par des mots binaires dans un ordinateur. Cependant, considérer de manière abstraite qu'il s'agit de texte au sens humain permet de faciliter l'analyse théorique.<sup>25</sup>

En particulier, trois flux *textuels* sont créés par défaut au moment de l'ouverture d'un terminal :

- le flux de sortie standard représenté en C par *stdout*
- le flux d'entrée standard représenté en C par *stdin*

25. Voir la théorie du langage dans la bibliothèque Minitel

■ le flux d'erreur standard représenté en C par *stderr*

Certains flux ont déjà été expérimentés dès à présent, comme *stdout* grâce à la fonction `printf`, ou de manière abstraite les flux *binaires* d'accès à la mémoire par la manipulation de variables.<sup>26</sup>

### 4.8.1 Flux d'entrée standard

On s'intéresse ici à la manipulation du flux d'entrée *stdin*. Il est possible de manipuler le flux des entrées clavier dans le terminal par la fonction `scanf` :

```
int x;
printf("Entrer un nombre : ");
int counter = scanf("%d", &x); // press enter to valid the entry
printf("%d valeurs saisies.", counter);
printf("Le nombre saisi est %d", x);
```

La fonction `scanf` ne fait pas que lire le flux des entrées clavier, elle effectue un formatage de ces entrées tout comme le fait la fonction `printf`. Par ailleurs, on peut observer qu'elle bloque l'exécution du programme jusqu'à l'appui sur la touche *Entrée* après avoir saisi une chaîne non vide de caractères.

Ce formatage diffère un peu du formatage de `printf`. On observe ainsi qu'au lieu de prendre la variable vers laquelle écrire en argument, `scanf` demande l'adresse de cette variable. En effet, écrire *x* fournirait la valeur de *x* (qui est indéfinie) et la fonction `scanf` n'a alors aucun moyen d'écrire dans la variable (puisque'elle ne sait pas où elle se trouve). On lui donne donc son adresse pour que `scanf` puisse y écrire.

Par ailleurs, on peut également formater plusieurs données d'un coup avec `scanf` :

```
int x, y;
printf("Entrer deux nombres : ");
int counter = scanf("%d %d", &x, &y); // press enter to valid the entry
printf("%d valeurs saisies.", counter);
printf("Les deux nombres saisis sont %d et %d", x, y);
```

### 4.8.2 Flux de sortie standard et flux d'erreur standard

Il est possible en C de choisir sur quel flux on veut afficher les données grâce à la fonction `fprintf`. Cette fonction prend un argument en plus, qui est le flux vers lequel envoyer les données :

```
fprintf(stdout, "Hello World !\n");
fprintf(stderr, "Une erreur est survenue !\n");
```

L'existence de *stderr* en plus de *stdout* peut sembler tout à fait superflue. Un premier intérêt est exhibé par l'exécution des deux codes suivants :

<pre>fprintf(stdout, "Hello World !"); while (1);</pre>	<pre>fprintf(stderr, "Petite erreur !"); while (1);</pre>
---	---

26. Dont les fonctionnements techniques ne reposent pas tout à fait sur le concept de flux ceci étant dit...

```
> ./stdout
```

```
> ./stderr
Petite erreur !
```

Le texte envoyé sur *stderr* s'affiche alors que le texte envoyé sur *stdout* non. La question, simple et bête, est la suivante : *pourquoi* ?

La réponse, quant-à-elle, n'est pas particulièrement compliqué, mais tout de même moins évidente que la question.

Pour des raisons d'optimisation, les flux peuvent utiliser un *tampon mémoire* (*memory buffer* en anglais). En effet, l'accès en écriture à des supports comme la carte vidéo ou un disque dur est en général beaucoup plus lent que l'écriture dans la RAM. Pour cette raison, les fonctions d'affichage tentent de minimiser le nombre d'accès effectués par le programme à ces supports. Au lieu d'envoyer directement à travers le flux les données textuelles, on les stocke d'abord dans une zone mémoire appelée un *tampon* et on vide ensuite d'un seul coup l'entièreté du tampon.

Il existe plusieurs stratégies de manipulation des tampons :

- Flux sans tampon (*unbuffered stream*) : les données textuelles sont transmises individuellement dès que possible
- Flux tamponné par ligne (*line buffered stream*) : les données textuelles sont transmises par blocs dès qu'un caractère de retour à la ligne est rencontré
- Flux complètement tamponné (*fully buffered stream*) : les données textuelles sont transmises par blocs d'une taille arbitraire déterminée précédemment

Le flux *stdout* est par défaut tamponné par ligne tandis que le flux *stderr* est par défaut sans tampon. On peut facilement obtenir la taille du bloc de tampon de *stdout* et de *stderr* :

```
1  #include <stdlib.h>
2  #include <stdio.h>
3  #include <stdio_ext.h>
4
5  int main() {
6      printf("Size of the default stdout buffer : ");
7      printf("%ld\n", __fbufsize(stdout));
8      fprintf(stderr, "Size of the default stderr buffer : ");
9      fprintf(stderr, "%ld\n", __fbufsize(stderr));
10     return EXIT_SUCCESS;
11 }
```

**Remarque :** Comme la fonction `__fbufsize` est appelée avant l'affichage de son résultat et que le tampon est initialisé sur la première écriture du flux *stdout*, oublier d'écrire le premier `printf` affiche une taille de tampon de 0.

Pour revenir à notre premier problème, on veut un affichage par défaut efficace mais on veut que dans le cas où le programme s'arrête du fait d'un dysfonctionnement, on puisse tout de même avoir le message d'erreur (par exemple, si on essaie d'accéder à une zone mémoire qui n'existe pas). La disjonction en deux flux de sorties différents est utile.

Par ailleurs, il est possible de rediriger les flux vers d'autres sorties que la console. On peut donc imaginer conserver *stdout* pour afficher du texte dans notre programme, mais rediriger *stderr* vers un fichier sur le disque dur qui servira de *log*<sup>27</sup>.

27. [https://fr.wikipedia.org/wiki/Historique\\_\(informatique\)](https://fr.wikipedia.org/wiki/Historique_(informatique))

### 4.8.3 Exemple pratique d'utilisation de *stderr*

On rappelle que `scanf` renvoie le nombre de valeurs saisies valides. Ainsi, à l'exécution du programme `ci-dessus`, si l'utilisateur entre des valeurs incorrectes il faut pouvoir avertir et donner un message d'erreur si la suite devient potentiellement bloquante :

```

1  #include <stdlib.h>
2  #include <stdio.h>
3
4  int main() {
5      int x, y, counter;
6      printf("Entrer deux nombres : ");
7      if ((counter = scanf("%d %d", &x, &y)) != 2) // press enter to valid the entry
8      {
9          fprintf(stderr, "Erreur saisie incorrecte.\n"); // \n only for readability
10     } else {
11         printf("%d valeurs saisies.", counter);
12         printf("Les deux nombres saisis sont %d et %d", x, y);
13     }
14     return EXIT_SUCCESS;
15 }
```

### 4.8.4 Exercices

**Exercice 29 (Distance Euclidienne).** Écrire un programme qui demande à l'utilisateur d'entrer deux points  $A = (x_A, y_A), B = (x_B, y_B) \in \mathbb{R}_{f64}^2$ , les affiche et donne le carré de la distance euclidienne entre ces deux points :

$$\|A - B\|_{euclide}^2 = (x_A - x_B)^2 + (y_A - y_B)^2$$

**Exercice 30 (Somme d'entiers).** Écrire un programme qui demande à l'utilisateur d'entrer  $N$  entiers et renvoie la somme de ces entiers. Ce programme ne devra pas utiliser plus de 3 variables.



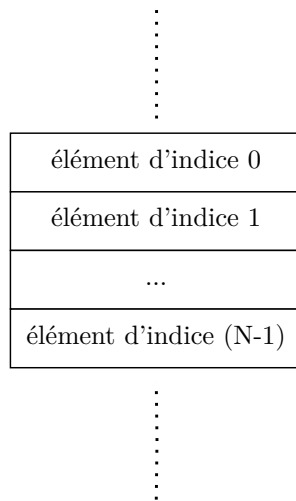
## TABLEAUX STATIQUES



### Définition 23 : Structure de donnée

Une structure de donnée est une manière de stocker les données. Il s'agit de la collection d'un ensemble de valeurs, de relations entre ces valeurs et de fonctions/d'opérations qui peuvent être appliquées à ces valeurs. Une structure de donnée peut être comparée ainsi à une structure algébrique en mathématiques.

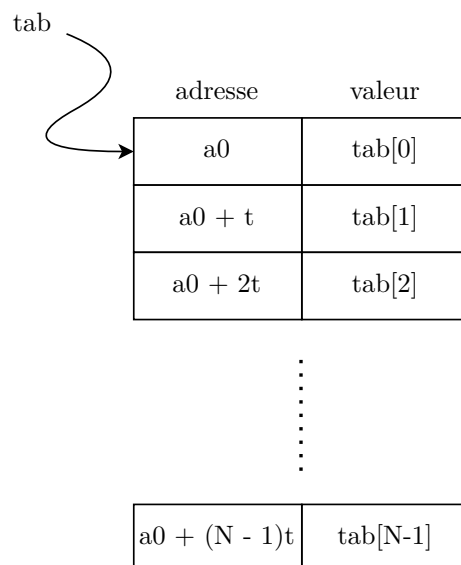
Le *tableau* est une des structures de donnée les plus simples. Il s'agit de l'équivalent informatique des  $N$ -uplets en mathématiques. Un tableau permet de stocker plusieurs variables sous une même étiquette de manière contigüe en mémoire :



Un tableau est dit *indexé*, c'est-à-dire que tous ses éléments sont numérotés, dans l'ordre, à partir de 0 :

- élément 1 : indice 0
- élément 2 : indice 1
- ...
- élément  $N$  : indice  $N - 1$

Si on considère que chaque élément est codé sur  $t$  bits, et que le premier élément est à l'adresse mémoire  $a_0$ , alors le  $i^e$  élément est situé à l'adresse  $a_i = a_0 + (i - 1)t$ . En particulier, vouloir accéder au  $N^e$  et dernier élément du tableau par l'adresse  $a_0 + Nt$  provoque une erreur, car cette adresse mémoire est située hors du tableau et est potentiellement interdite d'accès.



En langage C, il existe deux manières principales d'*implanter* un tableau :

- Par allocation statique : allocation par le compilateur d'un tableau de taille fixé avant l'exécution du programme
- Par allocation dynamique : allocation par le programmeur un tableau dont la taille dépend d'une variable du programme

### 4.9.1 Définition

L'allocation sur la pile, dite statique, est effectuée en langage C par la syntaxe suivante :

```
TYPE array[SIZE]; // uninitialized array
TYPE array[SIZE] = {v1, v2, ..., vSIZE}; // initialized array
```

On appelle *statique* ce type d'allocation de mémoire car *SIZE* ne peut être une variable ! Ainsi, il est incorrect d'écrire :

```
int n = 5;
int array[n]; // incorrect
```

Cette syntaxe provoque une erreur de compilation pour tous les compilateurs conformes aux normes du langage C avant la version C99. Certaines versions de certains compilateurs plus récents peuvent cependant accepter cette syntaxe et remplacer le code par une *allocation dynamique* (voir section suivante). Pour des raisons de compatibilité et pour éviter certains bugs, il est expressément recommandé d'éviter cette pratique douteuse.

Il est par contre tout à fait possible de préciser la taille d'un tableau par un symbole défini par une directive pré-processeur :

```
#define N 5
int array[N]; // before being compiled, every N in the code will be replaced by 5
```

Voyons simplement quelques exemples pour se familiariser avec la notation :

```
unsigned int chiffres[10] = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9};
char just_a_zero[1] = {0};
short int numbers[4] = {-78, 52, 17, -10};
```

On rappelle que les tableaux sont indicés de 0 à  $(TAILLE - 1)$ . On accède à un élément d'un tableau par la syntaxe suivante :

```
array[index];
```

Ainsi, le code suivant ajoute 1 à tous les éléments d'un tableau de taille 50 :

```
#define N 50
int main() {
    int array[N] = {...}; // any values
    for (int i = 0; i < N; i++) {
        array[i]++;
    }
}
```

**Remarque 1 :** tenter d'accéder au tableau par un indice supérieur ou égal à sa longueur provoque souvent une erreur. En effet, il n'est pas certain que la case juste hors du tableau ne soit pas utilisée par un autre programme de l'ordinateur et ne soit donc pas autorisée d'accès pour le programme.



**Remarque 2 :** La syntaxe permettant d’initialiser *toutes* les valeurs d’un tableau n’est valide qu’à sa déclaration. Il n’est ensuite possible de modifier les valeurs du tableau que *une à une!!!*

<pre>long int array[7];  // ERROR : array = {3141526535, 0xBEEF, 747, 0xC4, 713705, 666, 1414};  // STILL ERROR : array[7] = {3141526535, 0xBEEF, 747, 0xC4, 713705, 666, 1414};  // Python is not C : array = [i for i in range(7)]; // ERROR !!!</pre>	<pre>long int array[7];  // no errors : array[0] = 3141526535; array[1] = 0xBEEF; array[2] = 747; array[3] = 0xC4; array[4] = 713705; array[5] = 666; array[6] = 1414;</pre>
--	--

### 4.9.2 Les tableaux statiques, kékoï pour de vrai ?

Les tableaux statiques sont à un certain point de vue très proches comportementalement des pointeurs. Ils en diffèrent pourtant du tout au tout.

Pour mieux comprendre la nature d’un tableau, il faut exécuter le code suivant :

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  int main(int argc, char **argv) {
5      int array[3] = {1, 2, 3};
6
7      printf("array = %p | &array = %p | &(array[0]) %p\n", array, &array, &(array[0]));
8
9      return EXIT_SUCCESS;
10 }
```

Le résultat est assez surprenant, puisqu’on observe l’égalité des trois formules. Cela semble tout à fait étrange. En particulier, `tab` semble être égal à son premier élément, puisque `&(tab[0]) == &tab`, mais puisque `tab == &tab`, cela ne peut pas être le cas, puisqu’alors on aurait alors un premier élément toujours égal à l’adresse de `tab`.

Cette étrangeté tient en une phrase, d’apparence barbare : **les tableaux, en C, ne sont pas des entités de première classe**<sup>28</sup>. On appelle, en programmation, une entité de première classe une entité informatique régit par les mêmes règles générales que les autres entités. Il peut s’agir par exemple d’être renvoyé par une fonction, d’être assigné à une variable, ou d’être passé en argument à une fonction par exemple.

Il est absolument fondamentale de comprendre les conséquences de cette affirmation : les tableaux statiques ne sont pas régis par les mêmes règles que les autres entités du langage C. En particulier, il n’est pas possible de manipuler un tableau comme un tout de la même manière qu’une variable “classique”. En particulier, un tableau suit la règle suivante :

#### Règle de la conversion : <sup>29</sup>

28. Voir [https://www.gnu.org/software/c-intro-and-ref/manual/html\\_node/Limitations-of-C-Arrays.html](https://www.gnu.org/software/c-intro-and-ref/manual/html_node/Limitations-of-C-Arrays.html) ainsi que [https://en.wikipedia.org/wiki/First-class\\_citizen](https://en.wikipedia.org/wiki/First-class_citizen)

29. L’appellation n’a rien d’officiel, mais je trouvais qu’elle décrivait bien l’état de fait.

Tout identifiant référant à un tableau dans un code C subit une conversion comme pointeur vers l'adresse du premier élément du tableau. Aux exceptions suivantes :

- l'opérateur `&` renvoie l'adresse du tableau au sens d'entité, pas l'adresse du tableau au sens de la première case de celui-ci. Ainsi, `*(&array)` est égal à `array`, c'est-à-dire à l'adresse de la première case. Pour autant, on a tout de même `&array == array` car l'entité tableau se situe en mémoire au même endroit que les valeurs du tableau.<sup>30</sup>
- les opérateurs `sizeof`, `typeof` et `_Alignof` ne convertissent pas l'identifiant en pointeur mais travaillent uniquement sur le type du tableau, puisque cela est suffisant.

Cette règle amène à plusieurs propriétés.

### Accès aux éléments par incrémentation du pointeur

En considérant la déclaration d'un tableau quelconque :

```
| TYPE array[N] = {...};
```

L'égalité suivante est vraie pour tout indice positif strictement inférieur à  $N$  :

```
| array[indice] == *(array + index);
```

En effet, `tab` est ici interprété comme un pointeur `TYPE*` vers la première case. Il est donc possible d'accéder aux cases du tableau comme pour un pointeur. En fait, il faut plutôt penser à cela dans le sens inverse : l'opération a été définie sur les pointeurs, et on retrouve cette facilité d'écriture pour manipuler les tableaux.

**Remarque :** L'addition est une opération commutative... donc le code suivant est tout à fait correct :

```
| index[array] == array[index]; // == *(array + index)
```

Ainsi, le code suivant est correct :

```
1 | #include <stdio.h>
2 | #include <stdlib.h>
3 |
4 | int main() {
5 |     int some_array[10];
6 |     for (unsigned int i = 0; i < 10; i++) {
7 |         i[some_array] = i;
8 |         printf("%d;", some_array[i]);
9 |     }
10 |    printf("\b \n");
11 |    return EXIT_SUCCESS;
12 | }
```

La notation `n` n'est pas utilisée car assez illisible.<sup>31</sup>

30. Si ça vous paraît tordu... À moi aussi pour être honnête.

31. Question d'habitude...

### Passage d'un tableau en argument à une routine

Le langage C autorise à passer un tableau en argument à une routine (voir **Exercice 32.** et **Exercice 33.**). Mais cela cache en vérité une subtilité : ce n'est pas un tableau qui est passé en argument, mais un pointeur vers le tableau !

Un exemple :

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  void procedure(int array[]) { // seems to be an array
5      printf("%ld", sizeof(array));
6  }
7
8  int main() {
9      int test[5] = {1, 2, 3, 4, 5};
10
11     printf("%ld", sizeof(test)); // prints sizeof(int) * 5 = 20
12     procedure(test); // prints sizeof(int*) = 8
13
14     return EXIT_SUCCESS;
15 }
```

`sizeof` renvoie la taille mémoire calculée à la compilation. Ainsi, dans la fonction `main`, `sizeof(test)` est calculable car le compilateur fait le lien avec le tableau de cinq éléments défini précédemment, de type `int`.

Par contre, l'appel à `procedure` utilise une référence à un tableau, qui est donc interprétée comme un pointeur vers la première case. `procedure` recevra donc toujours un pointeur vers le premier élément du tableau. Un pointeur est stocké sur huit octets.

Par ailleurs, cela signifie également que le tableau n'est pas copié en mémoire au passage en argument, et que les modifications d'un tableau dans une routine sont permanents :

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  void incr_all(int array[], unsigned int length) {
5      for (int i = 0; i < length; i++) {
6          array[i]++;
7      }
8  }
9
10 int main() {
11     int test[5] = {1, 2, 3, 4, 5};
12
13     incr_all(test, 5);
14     incr_all(test, 5);
15
16     printf("%d = %d\n", test[2], *(test + 2));
17
18     return EXIT_SUCCESS;
19 }
```

L'incrémentation de tous les éléments du tableau est permanente et subsiste dans la suite du programme. Cela est logique, puisque c'est un pointeur vers les éléments du tableau qui est donné à la procédure.

### Renvoi d'un tableau par une fonction

Il a déjà été dit qu'un tableau ne pouvait être renvoyé par une fonction. Cependant, un pointeur peut l'être. Une première intuition serait donc de créer un tableau dans la fonction, puis de renvoyer un pointeur vers celui-ci :

```

1  #include <stdio.h>
2  #include <stdlib.h>
3
4  char *test_function() {
5      char test_array[10] = {3, 1, 4, 1, 5, 9, 2, 6, 5, 4};
6      return test_array;
7  }
8
9  int main() {
10     // Declared as pointer because test_function returns a pointer :
11     char *ret_array = test_function();
12
13     /* following declaration produces an error at compile time :
14     char ret_array[10] = test_function();
15     */
16
17     printf("Premier element (?) : %d\n", *ret_array);
18     return EXIT_SUCCESS;
19 }
```

Aucune erreur de compilation à signaler, mais à l'exécution... patatras. Le classique code -11 d'erreur de segmentation apparaît sur la console, dans l'incompréhension la plus totale...

Il faut se rappeler d'un point technique pour comprendre l'erreur ici : les variables initialisés statiquement, dont l'allocation en mémoire est effectuée par le compilateur, sont libérées par le compilateur à la sortie de la routine !

Le tableau déclaré et initialisé dans `fonction_test` a été libéré dès la sortie de la fonction. L'accès mémoire à son adresse est donc interdit par le système d'exploitation puisque cette zone mémoire n'est plus "fournie" au programme.

Une solution ? Oui, avec les tableaux dynamiques !

### 4.9.3 Exercices

**Exercice 31 (Affichage d'un tableau).** Écrire une procédure `void afficher(int tab[], int taille)` ; qui affiche les éléments d'un tableau d'entiers.

*Note : cela pourra être utile pour vérifier le résultat de routines agissant sur des tableaux*

**Exercice 32 (Somme d'un tableau).** Écrire une fonction `int somme(int tab[], int taille)` ; qui renvoie la somme des éléments du tableau.

**Exercice 33 (Maximum et minimum d'un tableau).** Écrire deux fonctions `int max(int tab[], int taille)` ; et `int min(int tab[], int taille)` ; qui renvoient respectivement le maximum et le minimum des éléments d'un tableau.



## TABLEAUX DYNAMIQUES



### 4.10.1 Introduction au tas

Un programme informatique, lorsqu'il est chargé en mémoire, comporte un troisième espace mémoire nommé *le tas* (*heap* en anglais). Il se nomme ainsi car il correspond à un tas de mémoire accessible au programmeur.<sup>32</sup>

Le tas est un espace mémoire de taille extensible<sup>33</sup>. Il peut être agrandi au fur-et-à-mesure des allocations, c'est-à-dire lorsque trop de variables ont été allouées. Ainsi, il est virtuellement possible d'utiliser l'entièreté de la RAM avec le tas. Il possède cependant certains défauts :

- les variables allouées ne sont pas toutes contiguës en mémoire, contrairement à la pile. Cela ralentit l'accès à celles-ci.
- les espaces mémoires alloués aux variables ne sont pas libérés automatiquement à la sortie des routines, **c'est au programmeur d'être responsable**

Le deuxième point est particulièrement important, puisqu'il peut être la source de graves bogues connus sous le nom de fuites de mémoire (*memory leaks* en anglais). La fuite de mémoire apparaît lorsqu'une variable est allouée dynamiquement sur le tas, et que le programmeur oublie de la libérer avant la fin de la routine. Le système d'exploitation continue de considérer que la zone mémoire est utilisée jusqu'à ce que le programme se termine. Dans le cadre d'applications conséquentes, cela peut être vraiment problématique.<sup>34</sup>

Un autre bogue très récurrent qui empêche totalement le fonctionnement du programme est l'accès à une variable dont l'espace mémoire a été libéré : s'ensuit l'erreur de segmentation, de code -11, provoqué par le système d'exploitation qui perçoit cela comme une attaque du système!<sup>35</sup> Ce bogue est rencontré en quasi-permanence chez les débutants, dès lors que des structures complexes sont manipulées en mémoire. Il arrive aussi très fréquemment<sup>36</sup> chez les programmeurs expérimentés. En effet, il est d'une facilité extraordinaire de rencontrer ce bogue dès lors que la programmation manque un tout soit peu de rigueur.<sup>37</sup>

### 4.10.2 Les routines malloc et free

Pour gérer "soi-même"<sup>38</sup> de la mémoire sur le tas, deux fonctions de la bibliothèque *stdlib.h* fournissent les fonctionnalités suivantes :

32. Au sens qu'il y en a beaucoup, et qu'on pioche "dans le tas" lorsqu'il s'agit de trouver une adresse pour allouer de la mémoire. L'appellation ne fait pas référence à la structure de donnée homonyme

33. Avec comme limite la taille de la RAM de l'ordinateur.

34. J'ai joué y a un bail à *Nehrim : At fate's edge*, un jeu amateur utilisant le moteur de jeu de *The Elder Scrolls IV : Oblivion*, et du fait de ces fuites de mémoire, le jeu *freezait* après une heure ou deux. Il était alors nécessaire de l'éteindre et de le relancer. Excellent néanmoins, je conseille !

35. Si il était possible d'accéder à n'importe quelle case mémoire d'un ordinateur sans protection, le *hacking* serait presque trivial.

36. Quoi que ceux-ci en disent ;)

37. C'est cette erreur qui crée la peur déraisonnée des pointeurs et la "haine" du C par quantité de programmeurs. Plus sérieusement, d'autres langages de programmation assez bas niveau qui assurent malgré tout en interne la sécurisation de la mémoire, comme le *Rust*, sont souvent préférés en entreprise pour une question de stabilité et d'assurance qualité. Ce qui ne change rien au fait que savoir programmer en C assure une compétence bien plus grande que de ne pas savoir, en raison de l'affrontement direct avec le "mal"

38. C'est les routines qui font le gros du boulot hein ! Faudrait pas se mettre à voler par les chevilles x)

- allouer un espace mémoire, grâce à `malloc`<sup>39</sup>
- libérer un espace mémoire, grâce à `free`

Le détail du fonctionnement interne des deux fonctions ne sera pas explicité.<sup>40</sup>

Voici les prototypes de ces deux fonctions issus de la documentation officielle :

```
#include <stdlib.h>
void *malloc(size_t size);
void free(void *ptr);
```

La fonction `malloc` prend en argument un nombre d'octets, et renvoie un pointeur de type quelconque vers la première case de l'espace mémoire alloué. La procédure `free` prend en argument un pointeur quelconque vers un espace mémoire, et libère l'entièreté de cet espace mémoire.

On observe ici une première utilisation des pointeurs quelconques `void*` : `free` prend simplement une adresse, et libère la mémoire associée en interne. Elle n'a pas à connaître le type de la variable libérée.

En particulier, tout accès ultérieur à cet espace mémoire peut **potentiellement** provoquer l'erreur de segmentation de code `-11`. Il ne s'agit que d'une potentialité car tant que l'espace mémoire libéré n'est pas réutilisé par un autre programme, le système d'exploitation peut ne pas relever d'erreur.

```
1  #include <stdio.h> // printf
2  #include <stdlib.h> // malloc and free
3
4  int main() {
5      int* array = (int*)malloc(sizeof(int) * 5); // Alloties 4 * 5 = 20 octets.
6
7      for (int i = 0; i < 5; i++) {
8          printf("%d\n", *(array + i));
9      }
10
11     free(array); // authorize future allocation of this address
12 }
```

Il s'agit d'un pointeur vers un espace mémoire de 20 octets. On peut y accéder de la même manière que pour un tableau.

**Remarque 1 :** La projection de type peut être conservé comme implicite :

```
short int* array = malloc(sizeof(short int) * 10); // Alloties 2 * 10 = 20 octets
```

**Remarque 2 :** Il est à présent possible d'allouer en mémoire un tableau dont la taille dépend des variables du programme, et de le renvoyer par une fonction !

```
1  #include <stdio.h> // printf
2  #include <stdlib.h> // malloc and free
3
```

39. Pour *Memory Allocation*

40. Mais pour le plaisir : [https://wiki.osdev.org/Memory\\_management](https://wiki.osdev.org/Memory_management), [https://wiki.osdev.org/Memory\\_Allocation](https://wiki.osdev.org/Memory_Allocation), <https://codebrowser.dev/glibc/glibc/malloc/malloc.c.html> et la G.O.A.T. <https://gee.cs.oswego.edu/dl/>

```

4  int* make_array(unsigned int n) {
5      int *array = (int*)malloc(sizeof(int) * n); // Allocates (4 * n) bytes
6      for (unsigned int i = 0; i < n; i++) {
7          array[i] = i;
8      }
9      return array;
10 }
11
12 int main() {
13     int* array = make_array(10); // 4 * 10 = 40 bytes array
14
15     for (unsigned int i = 0; i < 10; i++) {
16         printf("%d\n", array[i]);
17     }
18
19     free(array); // memory always have to be freed
20 }

```

### 4.10.3 Exercices



## TABLEAUX MULTIDIMENSIONNELS



Maintenant que sont acquis les concepts de tableaux statiques et dynamiques, il devient possible de construire la structure de donnée un peu plus complexe qu'est le tableau multidimensionnel, au sens informatique de tableaux imbriquées. Voici l'exemple d'un tableau 2-dimensionnel quelconque :

$$T_N = \begin{bmatrix} t_{n_0} \\ \vdots \\ t_{n_N} \end{bmatrix} = \begin{bmatrix} t_0[0] & \dots & t_0[n_0] \\ t_1[0] & \dots & t_1[n_1] \\ \vdots & & \vdots \\ t_N[0] & \dots & t_N[n_N] \end{bmatrix} \in \mathbb{R}^S \text{ où } S = \sum_{k=0}^N (n_k + 1)$$

**Remarque :** Un tableau 2-dimensionnel n'est pas nécessairement rectangulaire, puisque chaque ligne peut être de taille quelconque. Par exemple, on peut imaginer le tableau 2-dimensionnel suivant :

$$\begin{bmatrix} 1 \\ 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 \\ 9 \end{bmatrix} \quad (4.1)$$

### 4.11.1 Tableaux multidimensionnels statiques

Commençons par le moins utile. On ne peut définir statiquement que des tableaux multidimensionnels rectangulaire, c'est-à-dire dont chacune des lignes possède autant de colonnes que les autres.

Dans le cas statique, la déclaration d'un tableau  $D$ -dimensionnel de taille  $N_1 \times N_2 \cdots \times N_D$  revient exactement à allouer statiquement en mémoire un "grand" tableau de  $N_1 \times N_2 \cdots \times N_D$  cases. Et c'est seulement lors de l'indexation du tableau que l'aspect multidimensionnel ressort.

La syntaxe pour déclarer un tableau  $D$ -dimensionnel de taille  $N_1 \times N_2 \cdots \times N_D$  est la suivante :

```
#define N1 ...
#define N2 ...
...
#define ND ...

TYPE array[N1][N2]...[ND]; // D is the array's dimension
```

Ainsi, un tableau 2d d'entiers de 5 lignes et 8 colonnes est déclaré de la manière suivante :

```
int array[5][8];
// OR
int array[8][5];
```

**Remarque 1 :** L'ordre des tailles n'importe pas. En effet, il suffit de poser comme convention :

- dans le premier cas que la première dimension représente les lignes et la seconde les colonnes
- dans le second cas que la première dimension représente les colonnes et la seconde les lignes

On accède ensuite naturellement aux éléments du tableau par une double indexation :

```
array[2][3]; // value on line 3 and column 4 of the array
```

**Exercice 34 (Afficher un tableau 2d [COURS]).** Écrire un programme (et pas une routine) qui affiche les éléments d'un tableau 2d d'entiers.

Dans le cas d'un tableau à  $D$  dimensions, on accède à un élément par  $D$  indexations. Par exemple pour un tableau à 4 dimensions :

```
double spacetime_points[10][10][10][10];
spacetime_points[1][4][2][7] = 3.14;
printf("%lf", spacetime_points[1][4][2][7]);
```

**Avantages des tableaux multidimensionnels statiques :**

- Vitesse de lecture et d'écriture : comme la mémoire est allouée statiquement, il n'y a pas à accéder au tas pour créer le tableau, le lire ou l'écrire. On gagne ainsi en vitesse et on laisse au compilateur la possibilité d'optimiser le programme.
- Simplicité de la déclaration

**Limitations/Inconvénients des tableaux multidimensionnels statiques :**

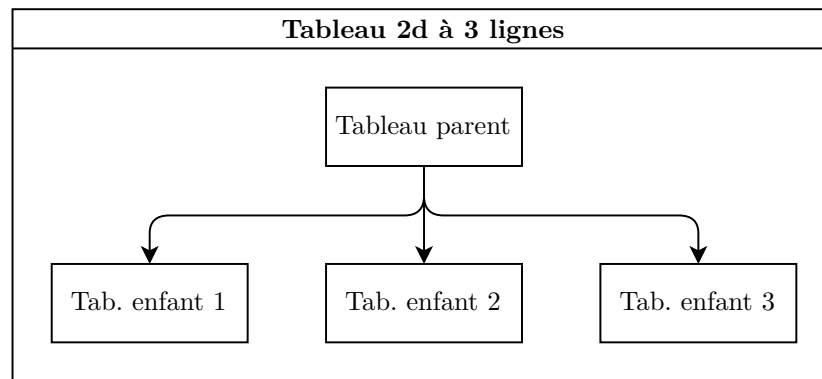
- Impossibilité de passer le tableau en argument à une routine : le compilateur ne sait pas comment convertir en pointeur les sous-tableaux à l'intérieur du premier tableau.
- Impossibilité de déclarer un tableau dont le nombre de colonnes est variable.

Pour outrepasser les deux limitations des tableaux multidimensionnels statiques, on utilise l'allocation dynamique.

### 4.11.2 Tableaux multidimensionnels dynamiques

L'idée de l'allocation dynamique de tableaux multidimensionnels est la suivante : on veut allouer dynamiquement un tableau qui va contenir des sous-tableaux que l'on va également allouer dynamiquement.





Si les sous-tableaux contiennent des valeurs de type `TYPE`, alors chacun d'entre-eux sera de type `TYPE*`. Par conséquent, le tableau parent doit être de type `TYPE**`. On en déduit le code suivant pour définir un tableau rectangulaire à  $N_{\text{lines}}$  lignes et  $N_{\text{columns}}$  colonnes :

```

TYPE** array2d = (TYPE**)(malloc(sizeof(TYPE*) * N_LINES));
for (int i = 0; i < N_LINES; i++) {
    array2d[i] = (TYPE*)(malloc(sizeof(TYPE) * N_COLUMNS));
}

```

Pour un tableau non rectangulaire, comme le tableau 4.1, on peut imaginer le code suivant :

```

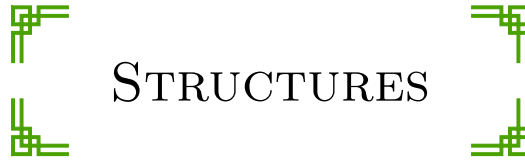
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  int min(int a, int b) {
5      return (a > b) ? b : a;
6  }
7
8  int main() {
9      int** triangle = (int**)(malloc(sizeof(int*) * 5));
10     for (int i = 0; i < 5; i++) {
11         triangle[i] = (int*)(malloc(sizeof(int) * (1 + min(5 - 1 - i, i))));
12
13         // initialize the array to 0
14         for (int j = 0, j < (1 + min(5 - 1 - i, i)); j++) {
15             triangle[i][j] = 0;
16         }
17     }
18
19     return EXIT_SUCCESS;
20 }

```

**Exercice 35 (Affichage du triangle [COURS]).** Compléter le programme ci-dessus pour remplir le tableau 4.1. Écrire ensuite une procédure `print_triangle(int **triangle, int n_lines)`; qui affiche ce tableau (où `n_lines` désigne le nombre de lignes du tableau)

**Remarque :** L'exercice ci-contre met en évidence la possibilité de passer en argument des tableaux multidimensionnels.

### 4.11.3 Exercices



Pour l'instant, seules les types élémentaires ont été vues, c'est-à-dire les nombres entiers et flottants, codés sur un, deux, quatre, huit ou dix octets. Il a aussi été vu comment créer des tableaux de ces types, statiques ou dynamiques.

Supposons cependant que l'on veuille modéliser informatiquement des structures complexes, comme par exemple :

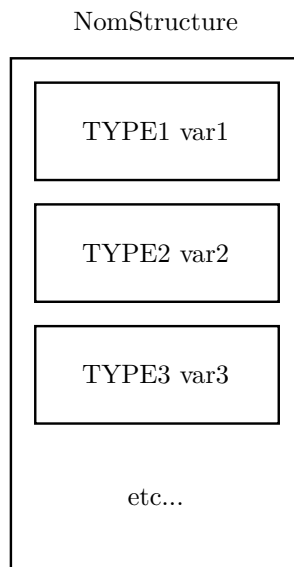
- la modélisation d'une lentille dans une simulation physique par une taille, la modélisation de sa forme par des courbes paramétriques, les propriétés du matériau, etc. . .
- le bouton d'une application par une taille, une position dans l'espace, une couleur, un texte, etc. . .
- un personnage d'un jeu vidéo. Celui-ci possède, par exemple, des points de vie, une vitesse, un modèle 3D qui est (grossièrement) un tableau de sommets, d'arêtes et de faces, etc. . .
- etc. . .

On pourrait stocker tout ceci dans des tableaux toujours plus grands d'entiers dont on peut interpréter les valeurs. Il faudrait alors noter dans une documentation que pour un tableau de `double` dont on conviendrait d'une convention de nommage pour se souvenir qu'il s'agit d'une lentille, la valeur `tableau[37]` correspondrait à telle ou telle propriété de la lentille.

C'est possible. Mais cela semble de manière évidente complètement tordu, et d'un sens pratique assez. . . comment dire. . . limité.

C'est ici que les structures entrent en jeu.

Une structure est un moyen de stocker dans une seule entité plusieurs types d'informations. Chacune de ces informations possédera sa propre étiquette au sein de cette entité, ainsi que son propre type. C'est un peu comme une boîte qui contient plusieurs variables :



La syntaxe de définition d’une structure en C est la suivante :

```

struct NomStructure { // No objects are created, it's just a description
    TYPE1 var1;
    TYPE2 var2;
    TYPE3 var3;
    // etc...
};
  
```

Il s’agit d’une définition, et non d’une déclaration. Pour cette raison, aucune variable interne, ou *membre*, de la structure ne peut être initialisée. Par ailleurs, une définition d’une structure est comme la définition d’un nouveau type de variable. Pour utiliser une structure, il faut au final créer des variables dont le type est cette structure :

```

struct NomStructure { // No objects are created, it's just a description
    TYPE1 var1;
    TYPE2 var2;
    TYPE3 var3;
    // etc...
};
struct NomStructure x; // 'x' is created with no members initialized
  
```

Il est possible d’y penser un peu comme à la construction d’une maison : on commence par dessiner le plan de la maison et une fois cela fait il suffit de fabriquer plein de maisons grâce à ce plan.

On peut ensuite accéder aux membres d’une structure via un nouvel opérateur binaire, l’opérateur d’accès à un membre noté “.” :

```

x.var1 = ...; // access to the 'var1' member of 'x'
x.var2 = ...; // access to the 'var1' member of 'x'
x.var3 = ...; // access to the 'var1' member of 'x'
// etc...
  
```

Un exemple :

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  struct Point3D {
5      double x;
6      double y;
7      double z;
8  };
9
10 int main() {
11     struct Point3D p;
12     p.x = 1.0;
13     p.y = 2.0;
14     p.z = -3.0;
15     printf("p = (%lf, %lf, %lf)\n", p.x, p.y, p.z);
16
17     return EXIT_SUCCESS;
18 }
```

**Remarque 1 :** Une structure ne peut contenir au maximum que 127 membres.

**Remarque 2 :** Au contraire des tableaux statiques, toute structure définie est considérée comme une entité de première classe. Ainsi, il est possible d'écrire des routines dont les paramètres sont des structures, et qui prennent en argument des structures identiques :

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  struct Point3D {
5      double x;
6      double y;
7      double z;
8  };
9
10 void afficher(struct Point3D p) {
11     printf("(%lf, %lf, %lf)\n", p.x, p.y, p.z);
12 }
13
14 int main() {
15     struct Point3D p;
16     p.x = 1.0;
17     p.y = 2.0;
18     p.z = -3.0;
19     afficher(p);
20
21     return EXIT_SUCCESS;
22 }
```

**Remarque :** Il est aussi possible qu'un membre d'une structure soit lui-même une structure :

```
1  #include <stdio.h>
2  #include <stdlib.h>
```

```
3
4 struct Couple {
5     double x;
6     double y;
7 };
8
9 struct Line {
10     struct Couple point;
11     struct Couple direction;
12 };
13
14 // or
15
16 struct Line {
17     struct Couple {
18         double x;
19         double y;
20     } point;
21     struct Couple direction;
22 };
23
24 int main() {
25     struct Line l;
26     // . is an associative operator :
27     l.point.x = 0.0;
28     l.point.y = 1.0;
29     l.direction.x = 1.0;
30     l.direction.y = 2.5;
31
32     return EXIT_SUCCESS;
33 }
```

Comme les structures définies sont considérés comme des entités de première classe, il est évidemment possible de créer statiquement ou dynamiquement des tableaux de ces structures :

```
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 struct Time {
5     short int milliseconds;
6     char hour;
7     char minutes;
8     char seconds;
9 }
10
11 int main() {
12     int i = ...;
13     struct Time running_times[10];
14     running_times[i].hour = ...;
15     return EXIT_SUCCESS;
16 }
```

Il est également possible de travailler sur des pointeurs de structure :

```

1  #include <stdio.h>
2  #include <stdlib.h>
3
4  struct Something {
5      int a;
6      char b;
7  };
8
9  int function(struct Something* s_ptr) {
10     (*s_ptr).a += (*s_ptr).b;
11     return (*s_ptr).a;
12 }
13
14 int main() {
15     struct Something s;
16     s.a = 0;
17     s.b = 1;
18     for (int i = 0; i < 10; i++) {
19         s.b = function(&s) / s.b;
20     }
21     printf("%d\n", s.a);
22     return EXIT_SUCCESS;
23 }

```

On observe que l'opérateur d'accès à un membre est prioritaire sur l'opérateur d'indirection \*<sup>41</sup>. Oublier de parenthéser amène à une erreur puisque `ptr.member` n'existe pas. En effet, un pointeur n'est qu'une adresse vers la structure. Il n'a donc pas de membres.

La notation apparaît cependant très lourde à écrire, alors une notation purement facilitatrice existe en C, la flèche :

<pre> struct MyStruct {     int a;     int b; };  struct MyStruct *s = ...;  printf("a = %d, b = %d\n", s-&gt;a, s-&gt;b); </pre>	<pre> struct MyStruct {     int a;     int b; };  struct MyStruct *s = ...;  printf("a = %d, b = %d\n", (*s).a, (*s).b); </pre>
---	---

### 4.12.1 Initialisation à la déclaration

L'initialisation de structures contenant une grande quantité de variables peut vite être fastidieuse. Le langage C propose deux facilités d'écriture pour initialiser les variables statiques (c.à.d obtenues sans allocation dynamique) d'une structure dès la déclaration d'une instance :

41. En effet, l'opérateur d'indirection ne fait qu'ajouter un décalage à l'adresse de la variable structurée pour accéder au membre de la structure. Ce calcul est effectué par le compilateur. Pour cette raison, l'opérateur d'accès à un membre fait partie des opérateurs les plus prioritaires du langage.

<pre> struct MyStruct {     int a;     char b;     void *c; // any pointer     long double d; };  // Sequential initializing struct MyStruct s = {     2&lt;&lt;20,     6,     NULL,     3.141592653589 }; </pre>	<pre> struct MyStruct {     int a;     char b;     void *c; // any pointer     long double d; };  // Designated initializing struct MyStruct s = {     .d = 3.141592653589,     .a = 2&lt;&lt;20,     .c = NULL,     .b = 6 }; </pre>
---	---

L'initialisation séquentielle suit exactement l'ordre des membres de la structure.

L'initialisation sélective permet de choisir très exactement quel membre reçoit quelle valeur.

**Remarque :** Les retours à la ligne sont optionnels et ne servent qu'à la lisibilité. Le code suivant est valide :

```

struct Point {
    double x;
    double y;
};

struct Point p = {.y = 4.5, 3.2}; // Point on x = 3.2, y = 4.5

```

Il est parfaitement possible de mélanger les initialisations séquentielles et sélectives. Dans un tel cas, l'initialisation séquentielle reprend au dernier membre désigné par une initialisation sélective. Partant, le code suivant initialise le membre *a* à 1 et le membre *d* à 30.560.

```

struct MyStruct s = {1, .d = 30.560};

```

Alors que le code ci-dessous initialise le membre *c* à `NULL`, le membre *d* à 48, le membre *b* à 0 et le membre *a* à `&s`.

```

struct MyStruct s = {.c = NULL, 48, .b = 0, &s}; // d not initialized

```

## 4.12.2 Exercices

**Exercice 36 (Listes chaînées).** L'objectif de cet exercice est d'introduire une nouvelle manière de structurer les données. La structure de données dite de *liste chaînée* permet de stocker des données non adjacentes en mémoire, au contraire des tableaux. Les *listes chaînées* ont l'avantage inédit par rapport aux tableaux de pouvoir changer de taille au cours de l'exécution du programme. Il devient possible d'ajouter ou de supprimer des éléments *dynamiquement*. On peut ainsi imaginer une liste d'items d'un utilisateur qui pourrait s'accroître indéfiniment<sup>42</sup>, comme un carnet d'adresses par exemple.

42. Dans la limite de la mémoire de l'ordinateur

On commence par construire la structure permettant de stocker un *noeud* de la liste, c'est-à-dire un de ses éléments. La valeur du noeud est la valeur de l'élément de la liste :

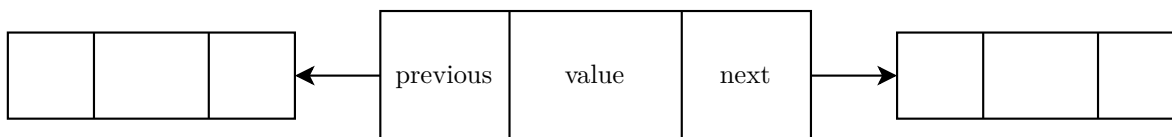


FIGURE 4.2 – Noeud d'une liste

En utilisant un pointeur temporaire vers les noeuds, on peut passer d'un noeud à l'autre grâce aux pointeurs *next* et *previous*.

1. Définir une structure `struct Node` qui contient un entier *value* (la valeur du noeud) et deux pointeurs vers une structure `struct Node` appelés *next* et *previous* (qui permettront de pointer vers les noeuds suivant et précédent de la liste)
2. Écrire une fonction `struct Node* node_new(int value)`; qui alloue dynamiquement une structure `struct Node* nd` initialisée telle que :
  - $nd \rightarrow next$  et  $nd \rightarrow previous$  ne pointent sur rien
  - $nd \rightarrow value = value$
3. Définir une structure `struct LinkedList` qui contient un entier *length* (le nombre de noeuds) et deux pointeurs vers une structure `struct Node` appelés *head* et *queue*. La *tête* de liste est le premier noeud de la liste. La *queue* de liste est le dernier noeud de la liste.
4. Écrire une fonction `struct LinkedList* linkedlist_new()`; qui alloue dynamiquement une structure `struct LinkedList* lst` initialisée telle que :
  - $lst \rightarrow head$  et  $lst \rightarrow previous$  ne pointent sur rien
  - $lst \rightarrow length = 0$
5. Écrire une fonction `char linkedlist_is_empty(struct LinkedList *l)`; qui renvoie 1 si la liste est vide, et renvoie 0 sinon.
6. Écrire une procédure `void linkedlist_push_on_head(struct LinkedList *l, int value)`; qui crée un noeud de valeur *value* et l'insère *en tête de liste*, c'est-à-dire qu'après insertion la valeur de la *tête* sera *value*. On pensera à traiter séparément le cas où la liste est vide (c'est-à-dire  $lst \rightarrow length = 0$ ). En effet, on rappelle que si  $lst \rightarrow head$  ne pointe sur rien, le considérer comme un noeud provoque une erreur du fait de l'accès à une zone mémoire non autorisée. Par ailleurs, si la liste contient un unique élément, la *tête* est égale à la *queue*.
7. De même, écrire une procédure `void linkedlist_push_on_queue(struct LinkedList *l, int value)`; qui crée un noeud de valeur *value* et l'insère *en queue de liste*.
8. Écrire deux fonctions `int linkedlist_pop_from_head(struct LinkedList *l)`; et `int linkedlist_pop_from_queue(struct LinkedList *l)`; qui suppriment respectivement le noeud en tête et en queue de liste et renvoient la valeur de ce noeud. On pensera à libérer la mémoire associée à l'allocation dynamique de ce noeud (c'est-à-dire utiliser `free` sur le pointeur vers le noeud).
9. En utilisant un pointeur temporaire vers les noeuds, écrire une procédure `void linkedlist_display(struct LinkedList *l)`; qui affiche les valeurs des noeuds de la liste.



## MODULATION ET ENTÊTES



Dans le cas du développement d'une application complexe, il arrive très souvent que la division d'un programme informatique en routines dans un unique fichier soit insuffisante. En effet, ces routines



peuvent couvrir des domaines très différents. Dans le cas d'un jeu-vidéo, cela peut aller des routines pour s'occuper de l'inventaire d'un personnage jusqu'aux routines s'occupant de la physique des objets du jeu en passant par les routines pour ouvrir, lire et écrire les fichiers de sauvegarde. Toutes ces routines très différentes ont besoins de structures de données différentes qui peuvent être accompagnés elles-mêmes d'une multitude de routines de manipulation. Bref... Autant dire que mettre tout ça dans un seul et unique fichier de code, ça risque d'être un poil touffu. En particulier, la gestion de l'évolution d'un unique aspect du code source indépendamment du reste devient extrêmement confuse, sans parler de l'organisation en équipes quand les travaux de toute une équipe peuvent interférer.

La solution à cela est simple : écrire dans des fichiers différents les routines qui s'occupent d'aspects différents du programme/de l'application développée(e).

Pour cela, on distingue en langage C deux catégories de fichiers :

- les fichiers d'entêtes (*headers* en anglais) d'extensions *.h*
- les fichiers de code source d'extensions *.c*

#### Définition 24 : Module et programmation modulaire

Un module est la donnée d'un fichier d'entête et d'un fichier de code source. Un programme informatique *modulaire* est un programme constitué de plusieurs modules. On appelle *programmation modulaire* le développement d'un programme par modules. Un module est un composant du programme qui peut lui-même utiliser d'autres modules pour son bon fonctionnement.

On peut penser aux modules informatiques comme aux modules d'un vaisseau spatial. Chacun est indépendant dans son fonctionnement interne mais tous les modules sont interdépendants d'un point de vue abstrait puisqu'ils sont chacun constitutif du programme et ne peuvent pas résoudre le problème individuellement.<sup>43</sup>

```
user@computer ~/working_directory> ls
main.c module1.c module1.h module2.c module2.h etc...
user@computer ~/working_directory>
```

### 4.13.1 Fichier d'entête

L'objectif du fichier d'entête d'un module est de donner au compilateur toutes les informations relatives au module qui ne sont pas le code source des routines proprement dit. Il s'agit donc :

- des instructions préprocesseurs (notamment les `#define` et `#include`)
- des structures
- des prototypes de routines

Le fichier d'entête décrit donc l'interface du module, puisqu'il s'agit de toutes les fonctionnalités qui seront accessibles au programmeur qui inclut le module. En effet, il s'agit du fichier que l'on inclut par la directive de préprocesseur `#include` :

```
| #include "chemin/vers/mon/module.h"
```

Le fichier peut se situer théoriquement n'importe où dans le système de fichiers du système d'exploitation.

43. Si cela est le cas, le programme n'est plus modulaire. On peut alors s'interroger sur la qualité de la conception.

### 4.13.2 Fichier de code source

Le fichier de code source, ou simplement fichier source, fournit le code des fonctions décrites par le fichier d'entête. Il peut également décrire des fonctions internes au module non présentes dans le fichier d'entête, qui ne pourront donc pas être utilisés en externe du module.

L'unique particularité du fichier source est de devoir inclure, par la directive `#include`, le fichier d'entête auquel il est associé :

```
#include "chemin/vers/mon/module.h"

/*
Code source des fonctions internes au module,
non accessibles hors du module
*/

// Code source des fonctions déclarées dans l'entête
```

### 4.13.3 Un exemple simple

On se propose ici de coder un module *vec2* qui contient des outils pour la manipulation de vecteurs en deux dimensions.<sup>44</sup>

On suppose se situer dans un répertoire de travail `<some_path>`. On considère trois fichiers dans ce répertoire :

- “main.c”
- “vec2.c”
- “vec2.h”

On nomme identiquement le fichier d'entête et le fichier source d'un module pour des raisons de lisibilités.<sup>45</sup> Toutefois, la liaison d'un fichier source avec un fichier d'entête passe uniquement par l'inclusion en début de fichier source du fichier d'entête associé.

Commençons par le fichier d'entête, puisque c'est ce que le fichier “main.c” va inclure :

vec2.h

```
1  #include <stdio.h> // preprocessor directives are in the headers
2
3  struct Vec2 {
4      double x;
5      double y;
6  };
7
8  struct Vec2 vec2_new(double x, double y);
9  struct Vec2 vec2_copy(struct Vec2 p);
10
11 void vec2_display(struct Vec2 p);
```

44. Il s'agit d'un exemple, et il n'y aura donc que le strict minimum pour la bonne compréhension de la programmation modulaire en langage C.

45. Cette pratique est d'ailleurs si répandue que certains tutoriels sur Internet vont jusqu'à affirmer qu'il est impossible que le fichier d'entête et le fichier source aient un radical différent.

```

12 struct Vec2 vec2_scalar_multiply(struct Vec2 p, double k);
13 struct Vec2 vec2_add(struct Vec2 p1, struct Vec2 p2);
14 struct Vec2 vec2_sub(struct Vec2 p1, struct Vec2 p2);

```

“vec2.h” décrit bien l’interface du module qui sera implanté réellement dans le fichier source :

vec2.c

```

1  #include "vec2.h" // needed to access to the header interface
2
3  struct Vec2 vec2_new(double x, double y) {
4      struct Vec2 v = {x, y};
5      return v;
6  }
7  struct Vec2 vec2_copy(struct Vec2 p) {
8      struct Vec2 v = {p.x, p.y};
9      return v;
10 }
11
12 void vec2_display(struct Vec2 p) {
13     printf("(%lf, %lf)", p.x, p.y);
14 }
15
16 struct Vec2 vec2_scalar_multiply(struct Vec2 p, double k) {
17     struct Vec2 v = {p.x * k, p.y * k};
18     return v;
19 }
20
21 struct Vec2 vec2_add(struct Vec2 p1, struct Vec2 p2) {
22     struct Vec2 v = {p1.x + p2.x, p1.y + p2.y};
23     return v;
24 }
25
26 struct Vec2 vec2_sub(struct Vec2 p1, struct Vec2 p2) {
27     struct Vec2 v = {p1.x - p2.x, p1.y - p2.y};
28     return v;
29 }

```

Finalement, le fichier “main.c” peut inclure le module *vec2* pour utiliser les structures et routines proposés :

main.c

```

1  #include <stdlib.h>
2
3  #include "vec2.h" // access to the header interface
4  // -> allows to use the routines and structures described in it
5
6  int main() {
7      struct Vec2 v = vec2_new(1.414, 3.14159);
8
9      vec2_display(v);

```

```

10 |
11 |     return EXIT_SUCCESS;
12 | }

```

#### 4.13.4 Compilation modulaire

La compilation modulaire introduit quelques notions plus complexes relatives à la compilation d'un programme. En effet, compiler "naïvement" le programme comme à l'habitude en ne précisant que le fichier "*main.c*" conduit à un dramatique message d'erreur, passablement incompréhensible sans plus d'informations :

```

user@computer ~/working_directory> ls # ou dir sous Windows
main.c vec2.c vec2.h
user@computer ~/working_directory> gcc main.c -o main
# ou chemin/vers/gcc.exe main.c -o main.exe sous Windows
/usr/bin/ld : /tmp/ccAnFoX.o : in function "main" :
main.c:(.text+0x25) : undefined reference to      "vec2_new"
/usr/bin/ld : main.c:(.text+0x4a) : undefined reference to "vec2_display"
collect2: error: ld returned 1 exit status
user@computer ~/working_directory>

```

Pour mieux comprendre cette erreur, et surtout comprendre comment résoudre le problème, il faut d'abord comprendre les couches d'opérations effectuées par le compilateur pour générer le fichier exécutable final. Ces couches sont :

1. l'exécution des directives préprocesseurs
2. la compilation [1]
3. l'assemblage [8]
4. l'édition des liens [8][6]

##### Exécution des directives préprocesseurs

Il s'agit principalement de déterminer quel est véritablement le code source qui sera compilé. Il peut s'agir d'ignorer certaines sections du code dépendamment du système d'exploitation considéré<sup>46</sup>, ou de prendre en compte les interfaces décrites par les modules inclusent par `#include`, de remplacer les constantes définies par `#define`, etc...

##### Compilation

Une fois que le compilateur sait exactement quel est le code source à prendre en compte, il s'adonne à la pratique qui lui donne son nom.

L'opération de compilation consiste à produire les codes *assembleurs* correspondant aux fichiers sources (en relevant les erreurs rencontrés pour aider le programmeur). L'opération elle-même se décompose grossièrement en quatre phases :

1. l'analyse lexicale : identifie dans le code source les différentes unités lexicales<sup>47</sup>
2. l'analyse syntaxique : construit un arbre de la syntaxe du programme et vérifie que celle-ci est correcte.

46. Par exemple grâce à `#ifdef`

47. Aussi appelés *lexèmes*, il s'agit de tous les mots qui font sens pour le langage, comme par exemple : '+', 'if', ')', ou encore n'importe quel nom de variable ou de fonction

3. l'analyse sémantique : après avoir validé la syntaxe, le compilateur construit le “sens” du programme, par exemple en vérifiant la validité des types des arguments vis-à-vis des paramètres d'une fonction par exemple. L'analyse sémantique constitue la part la plus complexe de la compilation.
4. la génération du programme assembleur : grâce aux résultats de l'analyse sémantique, le compilateur a construit le sens du programme et peut le reproduire en langage assembleur. Ce type de langage est le plus proche du langage machine. Le compilateur effectue au passage quelques optimisations de vitesse du code

À ce niveau, est associé à chaque fichier source un fichier d'extension `.s` qui correspond au langage assembleur *gas* utilisé par *gcc*<sup>48</sup>

Il est vitale de comprendre que pour l'instant, les noms de fonctions du programme sont toujours écrits en clair dans le code assembleur et sont appelés via l'instruction *call*.<sup>49</sup>

On peut générer le code assembleur d'un programme grâce à l'argument `-S` :

```
user@computer ~/working_directory> ls
main.c vec2.c vec2.h
user@computer ~/working_directory> gcc main.c -S
user@computer ~/working_directory> ls
main.c main.s vec2.c vec2.h
user@computer ~/working_directory>
```

qui peut être lu par un éditeur de texte classique.<sup>50</sup>

## Assemblage

*gcc* fait appel à un programme tiers nommé *assembleur* qui génère le code machine associé au code assembleur.<sup>51</sup>

À ce niveau, est associé à chaque fichier source un fichier *objet* (généralement d'extension `.o`). Ce fichier contient le code machine qui sera exécuté par l'ordinateur, ainsi que les symboles utiles pour l'édition de liens que sont les noms de fonctions par exemple.

On peut générer le fichier objet d'un programme grâce à l'argument `-c` :

```
user@computer ~/working_directory> ls
main.c vec2.c vec2.h
user@computer ~/working_directory> gcc main.c -c
user@computer ~/working_directory> ls
main.c main.o vec2.c vec2.h
user@computer ~/working_directory>
```

Cette fois-ci, le fichier objet devient assez illisible par un éditeur de texte puisqu'il contient du code machine.

48. Pour plus de détails : [https://en.wikipedia.org/wiki/GNU\\_Assembler](https://en.wikipedia.org/wiki/GNU_Assembler)

49. Une fonction est simplement une adresse en mémoire qui contient des données au même titre qu'une variable. Simplement, ces données sont du code exécutable.

50. Assez évident puisqu'un programme assembleur est un texte... Enfin je dis ça...

51. On distingue donc les *langages assembleurs* qui sont les langages au sens linguistique et les *programmes assembleurs* qui traduisent les programmes écrits en langage assembleur en code machine.

### Édition des liens

`gcc` fait appel ici au programme `ld`. Ce programme va analyser les liens existant entre les différents fichiers objets auquel il est soumis et va fusionner les fichiers objets en un seul fichier exécutable grâce à ces liens. Il peut au passage adapter le code en fonction de la machine vers laquelle est destiné le programme.

C'est ici que l'erreur précédente apparaît :

```
user@computer ~/working_directory> gcc main.c -c
user@computer ~/working_directory> ls
main.c main.o vec2.c vec2.h
user@computer ~/working_directory> ld main.o
# les autres erreurs sont dues à l'absence de certains fichiers complémentaires
main.c:(.text+0x25) : undefined reference to "vec2_new"
ld : main.c:(.text+0x4a) : undefined reference to "vec2_display"
user@computer ~/working_directory>
```

En effet, les appels par l'instruction `call` des symboles `vec2_new` et `vec2_display` présents dans le fichier "`main.o`" ne trouvent pas la définition des routines.

### Solution à l'erreur

Pour permettre à l'éditeur de liens de trouver les définitions des routines, il faut les lui fournir. Pour cela, on compile le programme en fournissant tous les fichiers sources en paramètre :

```
user@computer ~/working_directory> ls
main.c vec2.c vec2.h
user@computer ~/working_directory> gcc main.c vec2.c -o main
user@computer ~/working_directory> ./main
(1.414000, 3.141590)
user@computer ~/working_directory>
```

**Remarque :** Dans le cas de très grands programmes, il peut y avoir une grande quantité de modules. Il devient alors vite fastidieux d'écrire la commande de compilation<sup>52</sup>. Certains outils d'automatisation simplifient ces opérations et permettent de gagner en productivité (voir la section **Makefile**).

De manière générale, on écrit :

```
user@computer ~/working_directory> gcc main.c module1.c module2.c ... moduleN.c -o main
user@computer ~/working_directory>
```

### 4.13.5 Problème des chaînes d'inclusions

Une erreur très courante chez les débutants en programmation modulaire est le problème de la *double inclusion*. Il s'agit d'un bug lors de l'exécution des directives préprocesseurs provoqué par les chaînes d'inclusion. On l'illustre par l'exemple suivant, avec deux modules :

- un module *module1*
- un module *module2*

Le *module1* est dépendant du *module2* et le programme principal est lui-même dépendant des *module1* et *module2* :

52. Qui est en général beaucoup plus complexe, avec l'utilisation de plus de paramètres.

module1.h

```
struct SomeStructure {
    int thing;
};
```

module2.h

```
#include "module1.h" // access to the module1 interface

/*
Some interface
*/
```

main.c

```
#include "module1.h" // access to the module1 interface
#include "module2.h" // access to the module2 interface

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

L'inclusion dans "*main.c*" des deux modules pour accéder aux deux interfaces dont le programme a besoin est tout à fait naturel. Cependant, le *module2* a aussi besoin de l'interface du *module1*. Vient alors le problème : la structure *SomeStructure* apparaîtra comme définie deux fois du point de vue de "*main.c*".

On peut construire un bug d'inclusion infinie en ajoutant au *module1* l'inclusion du *module2* :

module1.h

```
#include "module2.h"
struct SomeStructure {
    int thing;
};
```

module2.h

```
#include "module1.h" // access to the module1 interface
```

Il est possible de construire des garde-fous basés sur des directives préprocesseurs pour forcer l'inclusion à n'être considérée qu'une seule fois par le compilateur.

### 4.13.6 Garde-fous

On introduit de nouvelles directives de préprocesseur ici :

- `#ifdef` et `#ifndef`
- `#else`
- `#endif`

Il s'agit de directives de préprocesseurs qui permettent d'ignorer certaines sections du code lors de la compilation. L'utilisation est la suivante :

<pre><code>#ifdef SOME_CONST /* Code executed if SOME_CONST is defined */ #else // not mandatory /* Code executed if not defined */ #endif</code></pre>	<pre><code>#ifndef SOME_CONST /* Code executed if SOME_CONST is NOT defined */ #else // not mandatory /* Code executed if defined */ #endif</code></pre>
---	--

On peut alors écrire des blocs de code qui ne seront jamais lus plus d'une fois par le compilateur grâce à l'astuce suivante :

```
#ifndef MODULE_INCLUDED // section can't be read more than once
#define MODULE_INCLUDED // because the symbol is defined just below
/*
Module is read only once
*/
#endif
```

On peut réécrire le *module1* et le *module2* de la façon suivante :

module1.h

```
#ifndef MODULE1_H_INCLUDED
#define MODULE1_H_INCLUDED

#include "module2.h"

struct SomeStructure {
    int thing;
};

#endif
```

module2.h

```
#ifndef MODULE2_H_INCLUDED
#define MODULE2_H_INCLUDED
```



```
#include "module1.h" // access to the module1 interface

#endif
```

### 4.13.7 Exercices

**Exercice 37 (Un module de listes chaînées).** Écrire un module qui contiennent les structures et fonctions définies dans l' **Exercice 36.** .



## CHAÎNES DE CARACTÈRES



La manipulation de texte au sens humain du terme n'est pas une fonctionnalité naturelle pour un ordinateur. Celui-ci ne peut que manipuler des chaînes de mots binaires constituées de 0 et 1, c'est-à-dire d'absence ou de présence de tension.

Pour pouvoir manipuler du texte, il faut d'abord pouvoir exprimer ce texte, donc poser une correspondance entre du texte et des mots binaires.

### 4.14.1 Caractères ASCII

Le premier standard international largement utilisé est apparu avec l'ANSI (*American National Standards Institute*). Celle-ci a posé une norme de correspondance entre les caractères de la langue anglaise et les mots binaires codés sur 7 bits. Cela définit donc  $2^7 = 128$  caractères :

Dec	Hex	Char	Dec	Hex	Char	Dec	Hex	Char	Dec	Hex	Char
0	0	[NULL]	32	20	[SPACE]	64	40	@	96	60	`
1	1	[START OF HEADING]	33	21	!	65	41	A	97	61	a
2	2	[START OF TEXT]	34	22	"	66	42	B	98	62	b
3	3	[END OF TEXT]	35	23	#	67	43	C	99	63	c
4	4	[END OF TRANSMISSION]	36	24	\$	68	44	D	100	64	d
5	5	[ENQUIRY]	37	25	%	69	45	E	101	65	e
6	6	[ACKNOWLEDGE]	38	26	&	70	46	F	102	66	f
7	7	[BELL]	39	27	'	71	47	G	103	67	g
8	8	[BACKSPACE]	40	28	(	72	48	H	104	68	h
9	9	[HORIZONTAL TAB]	41	29	)	73	49	I	105	69	i
10	A	[LINE FEED]	42	2A	*	74	4A	J	106	6A	j
11	B	[VERTICAL TAB]	43	2B	+	75	4B	K	107	6B	k
12	C	[FORM FEED]	44	2C	,	76	4C	L	108	6C	l
13	D	[CARRIAGE RETURN]	45	2D	-	77	4D	M	109	6D	m
14	E	[SHIFT OUT]	46	2E	.	78	4E	N	110	6E	n
15	F	[SHIFT IN]	47	2F	/	79	4F	O	111	6F	o
16	10	[DATA LINK ESCAPE]	48	30	0	80	50	P	112	70	p
17	11	[DEVICE CONTROL 1]	49	31	1	81	51	Q	113	71	q
18	12	[DEVICE CONTROL 2]	50	32	2	82	52	R	114	72	r
19	13	[DEVICE CONTROL 3]	51	33	3	83	53	S	115	73	s
20	14	[DEVICE CONTROL 4]	52	34	4	84	54	T	116	74	t
21	15	[NEGATIVE ACKNOWLEDGE]	53	35	5	85	55	U	117	75	u
22	16	[SYNCHRONOUS IDLE]	54	36	6	86	56	V	118	76	v
23	17	[END OF TRANS. BLOCK]	55	37	7	87	57	W	119	77	w
24	18	[CANCEL]	56	38	8	88	58	X	120	78	x
25	19	[END OF MEDIUM]	57	39	9	89	59	Y	121	79	y
26	1A	[SUBSTITUTE]	58	3A	:	90	5A	Z	122	7A	z
27	1B	[ESCAPE]	59	3B	;	91	5B	[	123	7B	{
28	1C	[FILE SEPARATOR]	60	3C	<	92	5C	\	124	7C	
29	1D	[GROUP SEPARATOR]	61	3D	=	93	5D	]	125	7D	}
30	1E	[RECORD SEPARATOR]	62	3E	>	94	5E	^	126	7E	~
31	1F	[UNIT SEPARATOR]	63	3F	?	95	5F	_	127	7F	[DEL]

**Remarque :** On peut trouver cette table en écrivant *man ascii* dans un terminal de commande sous Linux.

La plupart des systèmes manipulent des mots binaires dont le nombre de bits est multiple de 8. Par défaut, certains systèmes posent le bit de poids fort à la valeur 0. Il est toutefois possible d'étendre le codage ASCII en une version étendu (*extended ASCII*). Il existe plusieurs standards d'ASCII étendu et il n'en sera pas discuté ici.

**Exemple :** Selon l'ASCII, le caractère 'a' est décrit par le mot binaire :  $(1100001)_2 = 0 \times 61 = 97$ . On peut également effectuer la correspondance dans l'autre sens, en réécrivant sous forme de caractères des mots binaires de 8 bits. Ainsi,  $0 \times 2B$  décrit le caractère '+'

**Exercice 38 (Traduction ASCII [COURS]).** Trouver la chaîne de caractères décrite par la chaîne de mots binaires suivante :

0x4D696E6974656C203D20474F41540A0D00

Indication : on pourra segmenter en mots de 8 bits la chaîne.

## 4.14.2 Chaîne de caractères

Si un caractère est en informatique un mot binaire écrit sur 8 bits, une chaîne de caractères est basiquement un tableau de caractères. On observe toutefois quelques subtilités :

- un tableau est de taille finie, et donc doit aussi l'être une chaîne de caractères
- une chaîne de caractères ne remplit pas toujours l'entière d'un tableau. Il faut donc pouvoir indiquer la fin de la chaîne n'importe où

- la structure de donnée du tableau ne permet pas aisément certaines manipulations, comme l'insertion d'une chaîne de caractères au milieu d'une autre

**Remarque :** Concernant le dernier point, aucune solution ne sera proposée ici puisque ce n'est pas le coeur du propos. Il faut se rapprocher d'ouvrages d'algorithmie pour trouver une réponse satisfaisante.<sup>53</sup>

### 4.14.3 En langage C

La manipulation de texte sous forme de chaînes de caractères est native au langage C bien que nettement moins développé que dans d'autres langages.<sup>54</sup>

En particulier, on peut remarquer que le type du langage C pour des entiers sur 8 bits est *char*, comme *character* qui signifie *caractère* en français.<sup>55</sup>

La manipulation de caractères et de chaînes de caractères suit une nomenclature précise :

- on représente un caractère entre guillemets simples : 'a', 'b', '=', '!', ' ', etc...
- on représente une chaîne de caractères entre guillemets doubles : "Hello World!", "42 is the answer", etc...

En langage C :

```
char chr = '+';
char text[50] = {'H', 'e', 'l', 'l', 'o', ' ', 'W', 'o', 'r', 'l', 'd', ' ', '!', 0};
```

**Remarque 1 :** Comme *char* est un type entier, '+' est le mot binaire associé au caractère +. De même, le tableau de caractères est un tableau de mots binaires représentant chacun des caractères. Il s'ensuit qu'il est équivalent d'écrire :

```
char chr = 43;
char text[50] = {72, 101, 108, 108, 111, 32, 87, 111, 114, 108, 100, 32, 33, 0};
```

On peut ensuite afficher ces caractères par deux “nouveaux” modificateurs d'affichage :

- *char* interprété comme caractère : %c
- *char[]* ou *char\** interprétés comme chaînes de caractères : %s

```
printf("Caractere %c est represente par %d\n", chr, chr);
printf("Premier caractere du texte : %c\n", text[0]);
printf("%s\n", text); // characters array interpreted as text
```

Pour celles et ceux qui se posent la question : “*Whatz ve feuck ? Fé koi fe véro à la fin d'une faîne ?*”, le langage C pose par convention que la fin d'une chaîne de caractères est indiqué par le caractère spécial `\0`  $\equiv$  `[NULL]` de valeur entière 0. On peut l'écrire directement dans une chaîne de la façon suivante :

```
// 0 after the 'l', and 0 at the end of the string :
char text[50] = {'H', 'e', 'l', 'l', '\0', ' ', 'W', 'o', 'r', 'l', 'd', ' ', '!', 0};
printf("%s\n", text); // displays Hell on the screen
```

53. Cela dépend de plus du besoin. Un *gap buffer* n'aura pas les mêmes propriétés qu'un *rop tree*. C'est bon, j'ai drop quelques noms ça fait genre que je maîtrise ;)

54. Autant dire qu'en C, il n'y a que le strict minimum hein !

55. Coïncidence ? Je ne crois pas.

L’affichage s’arrête au premier 0 rencontré qui indique la fin de la chaîne.

On peut ensuite modifier cette chaîne de caractères comme un tableau classique :

```
char text[50] = {'H', 'e', 'l', 'l', '\0', ' ', 'W', 'o', 'r', 'l', 'd', ' ', '!', '\0'};
text[4] = 'o';
printf("%s\n", text); // displays Hello World ! on the screen
```

#### 4.14.4 Se faciliter la vie avec les chaînes littérales

Les lecteurs/rices attentifs/ives auront remarqué un fait étrange. La fonction `printf` affiche par défaut des chaînes de caractères :

```
printf("Hello World !");
```

et il n’y a pas besoin d’initialiser de p\*tain de tableau de c\*n pour cela.

Cela n’a en fait rien d’étrange<sup>56</sup>, le langage C propose nativement les chaînes de caractères littérales. Il s’agit d’une facilité d’écriture des chaînes de caractères qui consiste simplement à écrire le texte entre guillemets doubles “” et laisser le compilateur créer lui-même le tableau en interne, et le remplir avec les caractères précisés entre guillemets, plus un 0 pour finir la chaîne.

Ainsi, il est possible d’écrire simplement :

```
printf("%s", "%s\n"); // 0 is added automatically
```

Ici, le compilateur construit deux chaînes de caractères : la première qui contient les caractères ‘%’, ‘s’ et ‘\0’ et la seconde qui contient les caractères ‘%’, ‘s’, ‘\n’ et ‘\0’. Les caractères ‘%’ et ‘s’ ne constituent un caractère spécial de formatage que dans la chaîne manipulé par `printf`, c’est-à-dire la première.

#### 4.14.5 Exercices

**Exercice 39 (Calculatrice).** Écrire un programme qui demande à l’utilisateur d’entrer des calculs sous la forme  $x \star y$ , où  $x, y \in \mathbb{R}_{f64}$ ,  $\star \in \{+, -, /, *\}$  et lui donne le résultat. Si une erreur a lieu lors de la lecture, le programme termine. Ainsi, un scénario possible d’exécution est le suivant :

```
user@computer ~/working_directory> gcc calculatrice.c -o main
user@computer ~/working_directory> ./main
> 7 + 5
12.000000
> 8 * 2.2
17.600000
> 0.1 + 0.2
0.300000
> 1. - 0.8
0.200000
> 27.1 / 3
9.033333
```

56. Étonnant n’est-ce pas ? Qui l’aurait crû ?

```
> t * 8
Erreur !
user@computer ~/working_directory>
```

On rappelle que l'appui des touches *Ctrl + C* permet de forcer l'arrêt du programme. On peut donc exécuter une boucle infinie sans risques.

**Remarque :** *nan*  $\equiv$  *Not A Number* et *inf*  $\equiv \infty$ , que l'on obtient par  $0/0$  et  $1/0$ . Par ailleurs, ces deux “nombres” peuvent être utilisés dans la calculatrice.

*Indication :* un caractère est un nombre, il peut donc être utilisé dans un aiguillage 'switch-case'



## LES FLUX DE FICHIERS



*Recommandation :* revoir le vocabulaire sur les fichiers de la partie 1 du cours.

La manipulation de fichiers est vitale pour l'écriture de programmes persistents, c'est-à-dire dont certains états peuvent durer même quand le programme n'est pas en cours d'exécution. On peut par exemple penser à des fichiers de configuration de logiciels. La sauvegarde de certains états particuliers permet aussi au logiciel de traiter des données différentes et de stocker de manière persistente les résultats du/des traitements. On peut penser à tous les fichiers édités par des éditeurs de texte, des sauvegardes de jeux-vidéos, ou de manière beaucoup plus générique à la simple existence de l'entièreté des fichiers sur un disque dur qui permettent de donner une “mémoire longue durée” à l'ordinateur pour qu'il puisse s'exécuter en conservant une part des traitements effectuées.

La bibliothèque standard vous fournit différentes fonctions pour manipuler les fichiers, toutes déclarées dans l'en-tête `<stdio.h>`. Toutefois, celles-ci manipulent non pas des fichiers, mais des flux de données en provenance ou à destination de fichiers.

Cela permet notamment d'effectuer à nouveau des optimisations grâce aux tampons, et de rediriger des flux. On peut par exemple imaginer rediriger un flux standard de sortie vers un fichier pour écrire un *log*<sup>57</sup>, ou rediriger le flux de lecture d'un fichier vers la console (pour en afficher le contenu par exemple).

### 4.15.1 Droits des fichiers

Cette sous-section n'est destinée qu'aux utilisateurs des systèmes Unix comme Linux. En effet, sous Windows, le système est légèrement différent bien que par certains aspects plus proche qu'il n'y paraît. Il ne sera pas traité.<sup>58</sup> Le but est simplement de comprendre l'existence de droits. Pour plus de détails, voir <https://www.linuxtricks.fr/wiki/droits-sous-linux-utilisateurs-groupes-permissions>.

Les fichiers sous Linux ont 3 types de droits :

- lecture (noté *r* comme *read* en anglais) : possibilité de lire le contenu du fichier
- écriture (noté *w* comme *write* en anglais) : possibilité de modifier le contenu du fichier

57. [https://fr.wikipedia.org/wiki/Historique\\_\(informatique\)](https://fr.wikipedia.org/wiki/Historique_(informatique))

58. Pour plus d'informations : <https://learn.microsoft.com/en-us/windows/security/identity-protection/access-control/access-control>

- exécution (noté  $x$  comme *execution* en anglais) : si le fichier contient des instructions exécutables, possibilité de les exécuter

Il est nécessaire de manipuler ces droits avec précaution. En effet, lors de la mise en réseau d'ordinateurs, des droits mal gérés peuvent présenter rapidement des risques de sécurité local comme réseau.

Par ailleurs, chaque fichier sous Linux :

- a un propriétaire<sup>59</sup>
- est affilié à un groupe d'utilisateurs

On distingue parmi les utilisateurs qui peuvent potentiellement accéder au fichier (*i.e.* lire, écrire ou exécuter ce fichier) trois catégories :

1. le propriétaire du fichier (noté  $u$  comme *user* en anglais)<sup>60</sup>
2. les utilisateurs du groupe du fichier (noté  $g$  comme *group* en anglais)
3. les autres utilisateurs (noté  $o$  comme *others* en anglais)

Chaque fichier possède donc en données supplémentaires les droits de  $r$ ,  $w$  et  $x$  pour chacune de ces catégories.

Pour observer les droits des fichiers on peut utiliser l'argument  $-l$  (pour *long list*) de la commande  $ls$  dans un terminal Linux :

```
user@computer ~/working_directory> ls -la
total 20
-rwxrwxr-x 1 user user 15960 sept. 21 18:38 main
-rw-rw-r-- 1 user user      105 sept. 21 18:37 main.c
```

#### Petit aparté :

Le total indiqué est le nombre de ko utilisés (1 *ko* = 1024 *octets*) pour stocker en mémoire les données affichés par  $ls$ . Observe que le total d'octets utilisés est  $20 \times 1024 = 20480 > 15960 + 105$  qui est la taille réelle des fichiers présents. En effet, le système de fichiers utilise des zones mémoires par blocs de 4 ko appelés *pages*. On a :

- “*main*” avec  $\left\lceil \frac{15960}{4096} \right\rceil = 4$  pages
- “*main.c*” avec  $\left\lceil \frac{105}{4096} \right\rceil = 1$  page

Donc au total 5 pages.

#### Fin de l'aparté...

Les droits d'utilisateurs sont indiqués à gauche. Le format est le suivant :

Le type peut indiquer si un fichier est spécial. Par exemple, les répertoires sont considérés comme des fichiers spéciaux de type ‘ $d$ ’. Un fichier classique n’a pas de type précisé (un simple trait horizontal).

Les droits pour chaque catégorie d’accesseurs au fichier sont individuellement sous la forme  $(r, w, x)$ . Indiquer la lettre indique le droit d’accès. Ainsi, un fichier dont les droits d’utilisateurs sont décrits par :

-rwxrw - - - -

donne les informations suivantes :

<sup>59</sup>. Il ne s’agit pas nécessairement du créateur du fichier puisque la propriété peut être transmise

<sup>60</sup>. Appeler “utilisateur” le propriétaire est au mieux vague, au pire dénué de sens. Mais bon, c’est comme ça.

- il s'agit d'un fichier classique
- le propriétaire a les droits de lecture, d'écriture et d'exécution
- les utilisateurs du même groupe ont les droits de lecture et d'écriture
- les autres utilisateurs n'ont aucun droit sur le fichier

Pour la suite, il faut s'assurer que les fichiers manipulés par un programme C ont bien les droits nécessaires. L'utilisateur considéré pour cela est l'utilisateur qui a exécuté le programme. On peut *uniquement pour tester les programmes C dans le cadre de ce cours* ajouter l'intégralité des droits de lecture et d'écriture à un fichier existant par la commande :

```
user@computer ~/working_directory> chmod a+rw somefilename
user@computer ~/working_directory>
```

### 4.15.2 Les flux de fichiers en C

Pour manipuler des fichiers en C, il faut fondamentalement quatre routines :

- une routine d'ouverture/création de flux
- une routine d'écriture dans un flux
- une routine de lecture d'un flux
- une routine de fermeture d'un flux

Heureusement, `<stdio.h>` fournit toutes ces routines :

- `fopen` pour ouvrir/créer un flux de fichier
- `fwrite` pour écrire dans un flux de fichier
- `fread` pour lire dans un flux de fichier
- `fclose` pour fermer un flux de fichier

Les prototypes sont les suivants :

**FILE\* fopen(const char \*pathname, const char \*mode);**

**Entrées :**

*pathname* est une chaîne de caractère constante qui indique le chemin d'accès (relatif ou absolu) vers le fichier.

Le *mode* d'ouverture est aussi une chaîne de caractère constante dont les valeurs peuvent être :

- "r" : ouverture d'un flux *textuel* de *lecture*. Le flux est positionné *au début* du fichier (donc les données sont lues à partir de là).
- "r+" : *idem* que "r" mais flux de *lecture et d'écriture*
- "w" : Crée le fichier si il n'existe pas et le vide de tout contenu. Ouverture d'un flux *textuel* d'*écriture*. Le flux est positionné *au début* du fichier
- "w+" : *idem* que "w" mais flux de *lecture et d'écriture*
- "a" : Crée le fichier si il n'existe pas. Ouverture d'un flux *textuel* d'*écriture*. Le flux est positionné *à la fin* du fichier.
- "a+" : *idem* que "a" mais flux de *lecture et d'écriture*

L'ajout du caractère 'b' après la lettre ouvre le fichier avec un flux *binaire*. Par exemple : "*rb+*" ouvre un flux *binaire* vers le fichier en *lecture et écriture*.

**Sortie :** La fonction renvoie un pointeur vers un flux de fichier.

```
long int fwrite(const void *ptr, long int size, long int nmemb, FILE* stream);
```

**Entrées :**

*nmemb* est le nombre d'objets à copier de *size* octets situés à l'adresse *ptr* vers le flux *stream*.

**Sortie :** La fonction renvoie le nombre d'objets effectivement écrits. En cas d'erreur, ce nombre est strictement inférieur à *nmemb*

```
long int fread(void *ptr, long int size, long int nmemb, FILE* stream);
```

**Entrées :**

*nmemb* est le nombre d'objets à lire de *size* octets depuis le flux *stream*. Ces objets sont stockés consécutivement à l'adresse *ptr*.

**Sortie :** La fonction renvoie le nombre d'objets effectivement lus. En cas d'erreur ou d'arrivée à la fin du fichier (auquel il y a moins à lire que voulu), ce nombre est strictement inférieur à *nmemb*.

```
int fclose(FILE *stream);
```

**Entrée :** *stream* est le flux à fermer

**Sortie :** 0 en cas de succès. EOF en cas d'erreur.<sup>61</sup>

**Un classique de l'ISMIN**

Rien ne vaut un exemple à ce stade.<sup>62</sup> Le code ci-dessous va lire et écrire dans un flux binaire vers un fichier *“annuaire.data”*. Il va y stocker des informations sur des individus :

- leur nom sous forme d'un tableau de 50 caractères
- leur numéro de téléphone sous forme d'un entier sur 64 bits

Le programme suivant ajoute une nouvelle personne à chaque exécution :

```

1  #include <stdio.h>
2  #include <stdlib.h>
3
4  struct Person {
5      char name[50];
6      long int number;
7  };
8
9  struct Person person_make() {
10     struct Person p;
11     printf("Nom : ");
12     scanf("%s", p.name);
13     printf("Numero : ");
14     scanf("%ld", &(p.number));
15     return p;
16 }
17
18 void person_add_to_file(const char *pathname, struct Person p) {
19     FILE *fd = fopen(pathname, "ab");
20     if (!fwrite(&p, sizeof(struct Person), 1, fd)) {
```

61. EOF est une constante définie dans *<stdio.h>* et vaut généralement *-1*

62. Et pas des moindres, il s'agit de la fine fleur des idées de sujet des types qui écrivent les examens de l'ISMIN. Les pauvres... Il ne faudrait pas les laisser seules avec eux-mêmes. Ce serait inconscient.



```

21     fprintf(stderr, "Erreur d'écriture dans annuaire.data\n");
22 }
23 fclose(fd); // necessary : allows the file to be opened again
24 }
25
26 void person_print_numbers_from_file(const char *pathname) {
27     FILE *fd = fopen(pathname, "rb");
28     struct Person p;
29     while (fread(&p, sizeof(struct Person), 1, fd) == 1) {
30         printf("%s : %ld\n", p.name, p.number);
31     }
32     fclose(fd);
33 }
34
35 int main() {
36     struct Person p = person_make();
37     person_add_to_file("annuaire.data", p);
38     person_print_numbers_from_file("annuaire.data");
39     return EXIT_SUCCESS;
40 }

```

### 4.15.3 Rediriger un flux standard vers un fichier

La redirection de flux a été évoquée précédemment et on montre ici comment la réaliser.

**Remarque 1 :** Cette section est purement anecdotique.<sup>63</sup>

**Remarque 2 :** Faire des remarques comme ça, on dirait presque ChatGPT

**Remarque 3 :** À la différence majeure que ChatGPT n'a pas de sens de l'humour

La redirection de flux passe par la fonction `freopen` dont le prototype est le suivant :

```
FILE *freopen(const char *pathname, const char *mode, FILE *stream);
```

L'idée est simple : on ouvre un flux vers un fichier de chemin d'accès *pathname* en mode *mode* et on ajoute en argument le flux *stream* qu'il faut remplacer. *stream* est d'abord fermé puis ouvert et associé au flux de sortie de la fonction.

En un exemple :

```

1  #include <stdio.h>
2  #include <stdlib.h>
3
4  int main() {
5      freopen("log.txt", "w", stderr);
6      fprintf(stderr, "Erreur dans le log !!!");
7      fclose(stderr);
8      return EXIT_SUCCESS;
9  }

```

63. Précisons : inutile dans le cadre de la formation ISMIN, mais absolument pas dans le cadre d'un apprentissage plus complet. Sinon, il n'y aurait pas de chapitre après celui-ci pour traiter des concepts avancés du langage.

#### 4.15.4 Exercices

Voir les annales d'*Algorithmes et Programmation 1*. Toutes les notions nécessaires ont été abordées. Une correction par Minitel (et par les profs (beurk!)) sera proposé aussi vite que possible sur le NextCloud.

## CHAPITRE

# 5

# CONCEPTS AVANCÉS (EN COURS DE RÉDACTION)



## PARAMÈTRES D'UN PROGRAMME



Les programmes informatiques sont des routines, et plus particulièrement des fonctions, puisqu'ils retournent leur valeur de statut :

```
int main() {  
    return STATUT;  
}
```

Cependant, les programmes tels qu'ils ont été vus depuis le début de ce document n'ont pas de paramètres. Cela explique donc mal la possibilité d'écrire de genre de commandes dans un terminal :

```
user@computer ~/working_directory> gcc main.c -o main  
user@computer ~/working_directory>
```

En effet, les chaînes de caractères “main.c”, “-o” et “main” ont tout d'arguments donnés au programme *gcc*. Or cela n'est absolument pas possible avec le prototype de la fonction *main* qui a été donné dans la première partie du cours et tout au long de la seconde partie.

L'objectif de cette section est donc de revenir sur un petit “mensonge” à but simplificateur : le prototype de *main*

### 5.1.1 Le véritable prototype de *main*

Le prototype de *main* est le suivant :

```
| int main(int argc, char **argv);
```

Le compilateur C accepte la version sans paramètres mais c'est bien la version avec paramètres qui est complète. Les deux paramètres fournissent les informations sur les arguments donnés au programme :

- *argc* : nombre d'arguments donnés au programme
- *argv* : tableau des chaînes de caractères des arguments donnés au programme <sup>1</sup>

Plus particulièrement, *argc* est la longueur du tableau *argv*.

Par ailleurs, on a toujours  $argc \geq 1$ . L'explication est la suivante : c'est l'entièreté de la ligne de commande qui effectue l'appel du programme qui est passée à celui-ci. En particulier, le nom du programme est le premier mot de cette ligne et lui est donc passé également.

C'est-à-dire que  $argv[0] = \text{nom du programme}$  :

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  int main(int argc, char **argv) {
5      printf("%s\n", argv[0]);
6      return EXIT_SUCCESS;
7  }
```

```
user@computer ~/working_directory> gcc main.c -o main
user@computer ~/working_directory> ./main
./main
user@computer ~/working_directory> cd ..
user@computer ~> working_directory/main
working_directory/main
user@computer ~>
```

### 5.1.2 Exercices

**Exercice 40 (Liste des arguments).** Écrire un programme qui affiche toute la liste des arguments passés au programme, c'est-à-dire tel que :

```
user@computer ~/working_directory> ./main salut les gens !
./main
salut
les
gens
!
user@computer ~/working_directory>
```

---

1. Le double pointeur est ainsi justifié :

- la première étoile désigne le tableau de caractères
- la seconde étoile désigne le tableau des tableaux de caractères

**Exercice 41 (Un cat minimaliste).** L’objectif est de reproduire le comportement minimal de la commande *cat* du terminal.

Écrire un programme qui prend en argument un chemin vers un fichier texte et en affiche le contenu. On donnera par *stderr* un message d’erreur si aucun nom n’est donné en paramètre, et un autre message d’erreur si l’ouverture du fichier a échoué.

```
user@computer ~/working_directory> ./main fichier.txt
Ceci est le contenu du fichier !!!
user@computer ~/working_directory>
```



### 5.2.1 Motivation par un exemple

Il arrive régulièrement en programmation de manipuler des structures de données dont seules certains champs seront utilisés à un instant donné. Par exemple, on peut vouloir coder une structure stockant les données d’un évènement d’entrée utilisateur. Il peut s’agir :

- de l’appui d’une touche
- d’un mouvement de souris
- de la fermeture d’une fenêtre graphique
- etc...

Il faudrait une structure de donnée capable de stocker toutes les informations relatives à chaque type d’évènement. On pourrait imaginer la structure suivante :

```
struct Event {
    struct {
        unsigned x;
        unsigned y;
    } mouse;
    struct {
        int keycode;
        char is_maj_enabled;
        char is_ctrl_enabled;
    } keyboard;
};

...

struct Event e;
some_update_function(&e);
printf("Position souris : (%u, %u)\n", e.mouse.x, e.mouse.y);
```

On observe alors que la structure *mouse* et la structure *keyboard* ne seront jamais utilisées en même temps. C’est-à-dire qu’une instance de la structure *Event* n’“utilisera” jamais les deux sous-structures dans un même bloc de code. En effet, cette structure n’est destinée qu’à stocker un seul évènement. Analysons l’utilisation mémoire de la structure *Event* :

- $4 + 4 = 8$  octets pour stocker la sous-structure *mouse*

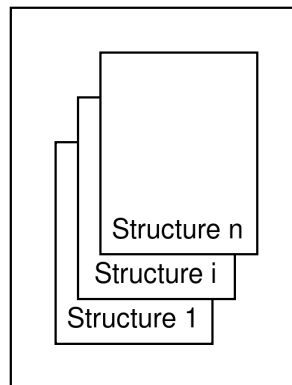
■  $4 + 1 + 1 = 6$  octets pour stocker la sous-structure `keyboard`  
pour un total de  $8 + 6 = 14$  octets.

En soi, il suffirait pourtant de seulement 8 octets pour stocker soit la sous-structure `mouse`, soit la sous-structure `keyboard`. C'est à cela que sert l'union.

### 5.2.2 Principe et syntaxe

L'idée de l'union est assez simple. Alors que la structure agrège les données<sup>2</sup> les unes à la suite des autres en mémoire, l'union les superpose dans le même espace mémoire :

Bloc mémoire de l'union



Ce qui en C s'écrit avec la même syntaxe que pour les structures, en utilisant à la place le mot-clé `union` :

```
union Event {
    struct {
        unsigned x;
        unsigned y;
    } mouse;
    struct {
        int keycode;
        char is_maj_enabled;
        char is_ctrl_enabled;
    } keyboard;
};
```

Et on observe qu'il s'agit bien du même espace mémoire :

```
union Event e;
e.mouse.x = 42;
printf("%u\n", e.keyboard.keycode); // prints 42
```

Le bénéfice est bien visible au niveau de la taille de la zone mémoire :

■ 16 octets pour `struct Event`<sup>3</sup>

2. En respectant ou non l'alignement

3. Et pas 14 du fait de l'alignement mémoire, voir la section dédiée

### ■ 8 octets pour union Event

Dans le cas de la programmation de systèmes embarquées ou pour des structures qui seront alloués un grand nombre de fois en mémoire, le gain de place est non négligeable. Il est ici d'un facteur 2 car il n'y a que deux évènements. La bibliothèque *Xlib* qui sert d'interface au serveur de fenêtrage X11 de Linux utilise une union de 33 structures pour stocker les évènements.<sup>4</sup>

Utiliser les unions à la place de structures quand cela est possible est une bonne pratique de programmation à assimiler.

## 5.2.3 Exercices

**Exercice 42 (Arbres binaires).**



Cette section vient en continuité de la section précédente sur les unions. Les champs de bits constituent en effet un autre moyen d'optimiser le stockage des données en espace. Il s'agit cependant d'une perte en vitesse d'exécution car la plupart des processeurs ne peuvent pas manipuler directement les bits d'un octet mais doivent appliquer des opérations bit-à-bit dessus.

### 5.3.1 Motivation et principe

Toutes les données contenues dans une structure ou une union ne nécessitent pas un nombre entier d'octets pour être stockées. Par exemple, un jour du mois appartient à l'ensemble  $\llbracket 0; 31 \rrbracket$  et ne nécessite donc que 5 bits.

De même un mois de l'année ne nécessite que 4 bits pour être stocké et une année n'en nécessite que 12 pour l'instant (car  $2^{12} = 4096$  semble suffisant). Il n'y a donc au total besoin que de  $4 + 5 + 12 = 21$  bits pour stocker une date.

Une première solution peut être de tout stocker dans un mot binaire de 4 octets, de type unsigned int et d'extraire ensuite les données stockées au format suivant :

$\underbrace{0000000000}_{\text{inutilisées}} \quad \underbrace{aaaaaaaaaa}_{12 \text{ bits pour l'année}} \quad \underbrace{mmmm}_{4 \text{ bits pour le mois}} \quad \underbrace{dddd}_{5 \text{ bits pour le jour}}$

```

1  #include <stdio.h>
2
3  unsigned short read_year(unsigned int date) {
4      return (date >> 9) & 0xFFF;
5  }
6
7  unsigned char read_month(unsigned int date) {
8      return (date >> 5) & 0b1111;
9  }
10
```

4. Voir : <https://tronche.com/gui/x/xlib/events/structures.html>

```

11 unsigned char read_day(unsigned int date) {
12     return date & 0b11111;
13 }
14 unsigned int write_day(unsigned short year, char month, char day) {
15     return (year & 0xFF) << 9 | (month & 0b1111) << 5 | (day & 0b11111);
16 }
17
18 int main(int argc, char *argv[]) {
19     unsigned d = write_day(2003, 12, 22);
20     printf("%u <=> %u/%u/%u\n", d, read_day(d), read_month(d), read_year(d));
21     return 0;
22 }

```

Cette solution est syntaxiquement très lourde, et il est assez facile de se tromper dans la lecture et l'écriture du mot binaire.

Heureusement que le langage C est là et qu'entre en jeu le champ de bits!<sup>5</sup>

### 5.3.2 Syntaxe

Les champs de bits<sup>6</sup> sont une facilité du langage C qui automatise les extractions de bits et les écritures de champs spécifiques d'un mot binaire. Il n'y a plus besoin d'écrire soi-même les fonctions de lecture et d'écriture de ces bits.

L'utilisation de champs de bits induit la création d'un nouveau type. Il est donc intuitif qu'il s'agisse d'une fonctionnalité propre à la définition de structures et d'unions :

```

struct MyBitFields {
    int f1 : width_of_f1;
    ...
    int fn : width_of_fn;
};

```

Ici, chaque champ  $f_i$  utilisera exactement  $w_i$  bits :

$$\text{MyBitFields} \equiv \underbrace{f_1}_{w_1 \text{ bits}} \dots \underbrace{f_n}_{w_n \text{ bits}}$$

et la structure sera donc stockée sur un nombre d'octets plus limité.<sup>7</sup>

Pour revenir à l'exemple des dates :

```

1  #include <stdio.h>
2  #include <stdlib.h>
3
4  struct Date {

```

5. Enfin, qu'entrent en jeu les champs de bits, parce-qu'a priori ils ne devraient pas souvent se balader seuls. Quoiqu'on sait jamais. ...

6. Bon, à partir de maintenant, je les remets ensemble. C'est rigolo parce-que les moutons isolés, on les perd dans des champs, alors que là ce sont les champs qui pourraient se perdre...dans des moutons ? Ça doit être moi qui me perd là :|

7. On a pas toujours  $\text{sizeof}(\text{MyBitFields}) = \left\lceil \frac{\sum_{i=1}^n w_i}{8} \right\rceil$  à cause de l'alignement.



```
5   unsigned year;
6   unsigned month;
7   unsigned day;
8 };
9
10 struct BFDDate {
11     unsigned year : 12;
12     unsigned day : 5;
13     unsigned month : 4;
14 };
15
16 int main(int argc, char *argv[]) {
17     printf("%ld\n", sizeof(struct Date)); // prints 12
18     printf("%ld\n", sizeof(struct BFDDate)); // prints 4
19
20     struct BFDDate d = {.day=22, .month=12, .year=2003};
21     printf("%u/%u/%u\n", d.day, d.month, d.year); // prints 22/12/2003
22     return EXIT_SUCCESS;
23 }
```

La taille de la structure struct BFDDate est bien celle d'un entier sur 4 octets, comme cela avait déjà été déterminé dans la sous-section précédente.

Par ailleurs, l'accès en lecture/écriture des champs est bien plus aisée !

### 5.3.3 Le revers de la médaille

Il a été dit au premier paragraphe de cette section que les champs de bits constituaient une optimisation d'espace mais qu'on observait une perte en vitesse d'exécution. La nécessité d'opérations supplémentaires pour accéder aux données explique ce coût temporel supplémentaire.

La question importante est de savoir à quel point ce ralentissement est notable. Pour cela, on peut simplement exécuter les deux codes suivants et les chronométrer :

<pre> #include &lt;stdlib.h&gt; #include &lt;stdio.h&gt;  struct BFDate {     unsigned year : 12;     unsigned day : 5;     unsigned month : 4; };  int main(int argc, char *argv[]) {     struct BFDate d = {.day=22, .month=12, .year=2003};      for (unsigned int i = 0; i &lt; 4294967295; i++) {         d.day += 1;         d.month += 1;         d.year += 1;     }     printf("%u/%u/%u", d.day, d.month, d.year);     return EXIT_SUCCESS; } </pre>	<pre> #include &lt;stdlib.h&gt; #include &lt;stdio.h&gt;  struct Date {     unsigned short int year;     unsigned char month;     unsigned char day; };  int main(int argc, char *argv[]) {     struct Date d = {.day=22, .month=12, .year=2003};      for (unsigned int i = 0; i &lt; 4294967295; i++) {         d.day += 1;         d.month += 1;         d.year += 1;     }     printf("%u/%u/%u", d.day, d.month, d.year);     return EXIT_SUCCESS; } </pre>
---	--

Il s'agit exactement du même code à l'exception de l'utilisation des champs de bits dans un cas et pas dans l'autre.

Le chronométrage a le défaut de dépendre de la machine utilisée. Celle-ci induit un coefficient de vitesse. Il faut donc mesurer le rapport du temps d'exécution de chaque programme l'un par rapport à l'autre. De plus, il faut avoir conscience que la différence de vitesse entre les deux programmes est dû à la manipulation par le processeur de mots *élémentaires* de 8 bits. Un processeur manipulant des mots élémentaires de 1 bit serait probablement plus rapide avec le code utilisant les champs de bits.

Sur un processeur *11th Gen Intel Core i7-11800H*, en compilant sans optimisations (niveau 0)<sup>8</sup> du compilateur par :

```
gcc main.c -o main -O0
```

on obtient à l'exécution dans les mêmes conditions logiciels (mêmes autres processus en cours d'exécution) les deux temps suivants :

- Sans champs de bits : 7.10 s
- Avec champs de bits : 37.59 s

L'utilisation des champs de bits provoque un ralentissement de 528%. En compilant avec différents niveaux d'optimisations (entre 1 et 3) :

1.  $rapport = \frac{2.42}{0.94332} = 257\%$
2.  $rapport = \frac{2.15}{0.00299} = 71906\%$
3.  $rapport = \frac{2.82}{0.00281} = 100356\%$

Les rapports extraordinaires des optimisations de niveau 2 et 3 proviennent du fait que le compilateur "précalcule" logiquement les résultats des additions du fait d'un schéma dans le code qui a été repéré. En vérité, il n'y a même plus de boucle dans le code après la compilation.

8. Il s'agit du niveau d'optimisations par défaut, raison pour laquelle il n'y pas besoin de le spécifier habituellement. Compiler à un niveau zéro d'optimisation permet de plus grandes facilités de déboguage.

L'utilisation des champs de bits empêche le compilateur d'effectuer une telle optimisation car il ne peut assurer formellement que le résultat en sortie sera le même que celui qui aurait été obtenu dans modifications<sup>9</sup>, ce qui explique un plateau du temps d'exécution autour de 2.5 s.

**Ce qu'il faut retenir :** Les champs de bits ne doivent être utilisés que pour de l'optimisation mémoire dans le cas où la machine exécutant le code n'en possède que peu (par exemple en systèmes embarqués). Il faut toutefois avoir conscience que l'utilisation des champs de bits pour d'autres raisons est particulièrement contreproductive.



## CLASSES DE STOCKAGE



Cette section vient apporter des outils pour préciser la durée de vie d'une variable et sa portée d'utilisation. On appelle :

### Définition 25 : Portée d'une variable et durée de vie

On définit ici deux notions qui ont été entrevues précédemment :

- la portée (*scope* en anglais) d'une variable est l'espace/l'ensemble des sections de code dans lesquels cette variable est déclarée et utilisable.
- la durée de vie (*lifetime* en anglais) d'une variable est la portion de l'exécution du programme durant laquelle l'espace de stockage de la variable est assuré comme réservé pour elle.

Les classes de stockage permettent de définir avec plus de précision la portée et la durée de vie d'une variable au moment de sa déclaration.

### 5.4.1 Quelques détails de la structure d'un programme en mémoire

**Rappel :** Le processeur peut enregistrer les données soit dans la RAM (ou mémoire vive) soit dans les registres du processeur.

On va détailler ici un peu plus l'organisation en mémoire vive d'un programme informatique. Un programme informatique est chargé en mémoire dans quatre différents types de zones mémoires :

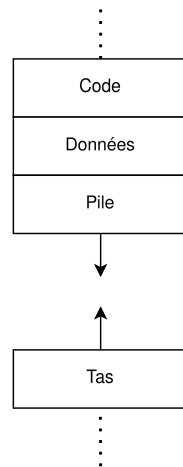
- section de code, aussi appelée section de texte : contient les instructions du programme, qui entre autre manipulent les registres.
- section de données initialisées et non initialisées : contient les données globales ou *statiques*<sup>10</sup>
- la pile : contient les variables temporaires, les arguments de routines, les adresses de retour des routines, etc...
- le tas : contient les variables allouées dynamiquement

Schématiquement, cela ressemble<sup>11</sup> à ça :

9. Les détails de la manière dont les optimisations sont effectués n'ont que peu d'importance ici et sont par ailleurs extrêmement complexes puisque ces optimisations sont très dépendantes du processeur utilisé. Il faut noter que la production des processeurs et des compilateurs est extrêmement lié puisque les processeurs exécuteront des programmes compilés et optimisés par les compilateurs. Il faut donc *designer* des processeurs optimaux vis-à-vis des optimisations des compilateurs et des compilateurs dont les optimisations prennent en compte les dernières fonctionnalités des processeurs.

10. i.e. durables, la notion sera détaillée dans la suite.

11. Les sections ne sont pas toujours dans le même ordre. Les données peuvent être mélangées au code, le tas pourrait évoluer au dessus du code, etc...



Pour éviter une collision potentielle entre le tas et la pile, cette dernière possède une taille maximale dans la plupart des systèmes d'exploitation.

La déclaration d'une variable peut alors avoir lieu dans, au choix :

- une section de données
- la pile
- le tas
- un registre

### Section de données

Les sections de données d'un programme sont présentes dans le fichier binaire du programme compilé. Les sections de données ne peuvent donc ni grandir ni rétrécir puisque le code source du programme (avant compilation) décrit entièrement la taille des données. Elles sont de taille fixe et certaines données peuvent avoir une valeur initiale. On distingue alors deux types de sections de données :

- les sections initialisées : les valeurs initiales sont indiquées dans le fichier binaire du programme compilé
- les sections non initialisées : seule la taille de la donnée est précisée.

Une conséquence très importante de l'impossibilité d'une section de données de changer de taille est que toutes les variables définies dans cette section ne peuvent être supprimées. C'est-à-dire que leur valeur est *persistante*. Une variable de section de données présente dans une routine ne voit pas sa valeur réinitialisée à la sortie de la routine.

### Variables allouées dynamiquement

Les données des variables allouées dynamiquement sont présentes dans le tas. Cependant, le pointeur vers ces données est lui stocké indépendamment de ces données. Il peut donc être stocké dans l'un des quatre lieux de stockage de variable décrits ci-dessus.

### Registres

Les registres sont de très petites unités de mémoire, en général constituées de bascules *RS* ou *JK* en séries. Chaque bascule stocke un bit. Les processeurs ne possèdent pas une grande quantité de registres car ceux-ci sont très coûteux par rapport à la quantité de mémoire qu'ils proposent. Leur accès est


toutefois extrêmement rapide, en écriture comme en lecture. Ils servent donc souvent <sup>12</sup> comme variables temporaires pour les calculs du processeur.

Il est très difficile <sup>13</sup> d'accorder correctement les registres aux variables pendant les calculs. Cette opération est en général effectuée de manière automatisée par le compilateur.

---

12. Cela dépend en vérité du type de registre. Certains registres sont spéciaux et ont un sens spécifique pour le processeur tandis que d'autres sont très généraux et peuvent servir à tout est n'importe quoi.

13. Au sens informatique de complexité, il s'agit d'un problème d'ordonnancement *NP*-difficile.



ESPACES LOCAUX ET ESPACE GLOBAL



CONSTRUCTION DE LITTÉRAUX



POINTEURS DE ROUTINES



TABLEAUX DE ROUTINES



FONCTIONS VARIADIQUES



ALIGNEMENT



DIRECTIVES DE PRÉPROCESSEURS (2)



ASSEMBLEUR EN C



LA VIRGULE

Ceci n'est pas une blague. Il s'agit d'une section traitant de la virgule “,” en C, à la fois comme opérateur entre différentes expressions et comme séparateur de déclarations.

Cette section n'apparaît qu'à la fin du cours de C au vu de l'anecdoticité<sup>14</sup> de cette caractéristique du langage. Il s'agit uniquement d'être exhaustif.

Et très honnêtement, ça ne sert pas à grand-chose... Mais si jamais, c'est là !

### 5.13.1 La virgule comme opérateur

La virgule sert à effectuer des opérations à effets de bord lors de l'écriture d'expressions.

**Rappel :** Une routine à effet de bord est une routine qui agit autrement que par sa valeur de retour, c'est-à-dire qui agit en dehors de son environnement local. Par exemple, une routine qui affiche du texte est une routine à effet de bord. Une routine qui modifie la valeur d'un tableau entré en argument est une routine à effet de bord.

La syntaxe est la suivante :

```
| expression = (expr1, ..., exprn);
```

Elle est équivalente à :

```
| expr1;
| expr2;
| ...;
| expr<n-1>;
| expression = exprn;
```

Un exemple purement technique consistant à compter le nombre d'assignations à des variables :\*

```
1  #include <stdio.h>
2  #include <stdio.h>
3
4  #define ASSIGN(X) (i++, X)
5
6  int main(int argc, char **argv) {
7      unsigned int i = 0;
8      int a = ASSIGN(42);
9      int b = ASSIGN(64);
10     b = ASSIGN(a + b);
11     int c = ASSIGN(i*b + a);
12     printf("Nombre d'assignations : %u\n", i);
13     return EXIT_SUCCESS;
14 }
```

Un second exemple à peine moins inutile pour l'échange de valeur de deux variables :

```
| int a = 42;
| int b = 64;
| int tmp = (tmp = a, a = b, b = tmp);
| print("a = %d et b = %d\n", a, b);
```

---

14. Je ne sais pas si le mot existe...

**Remarque :** Oublier les parenthèses dans ce deuxième exemple est rédhibitoire puisque le compilateur va considérer que les variables `a` et `b` sont redéfinies dans le même bloc, ce qui amène à une erreur de compilation.

En particulier, il faut rester attentif à la priorité des opérateurs. L'opérateur d'assignation est en effet prioritaire sur la virgule.

```
| int x = 1, 2, 3;
```

est incorrect car équivalent à :

```
| (((int x = 1), 2), 3);
```

qui n'a pas de sens.

### 5.13.2 La virgule comme séparateur

La virgule sert également dans un cas déjà observé : la séparation des déclarations de variables.

```
| int a = 3, b = a + 1;
```

### 5.13.3 Conclusion

Ben voilà c'est tout...



## TRICKS RÉCRÉATIFS (FT. BASILE)



L'objectif de cette section récréative est l'écriture de codes C illisibles, et même dirait-on particulièrement immondes. On utilisera pour cela divers outils du langage C plus ou moins peu inutilisés :

- l'indexation inversée
- la définition *K&R* de fonctions (première version du C)
- les digraphes et trigraphes
- la concaténation de symboles préprocesseurs
- les tableaux anonymes (construction de littéraux)
- etc...
- un peu d'imagination

Pour rester pragmatique, observer des codes très mal écrits ou utilisant des fonctionnalités extrêmement peu utilisés et juste là pour être illisibles amène à apprécier par contraste l'importance d'un style d'écriture clair et compréhensible.



### 5.14.1 Indexation inversée et boutisme

Il ne s'agit que d'un rappel :

```
int a = 0x12345678;
char *b = (char *)&a;
printf("%d", 1[b]);
// same as :
printf("%d", b[1]);
```

On observe sur un processeur Intel que le programme ci-dessus affiche 86 = 0x56 au lieu de 52 = 0x34. En effet, les processeurs Intels inversent les octets des données en mémoire, bien qu'ils les interprètent correctement <sup>15</sup>.

Cela ouvre des possibilités pour écrire des codes très moches.

### 5.14.2 Définition *K&R* de fonctions

La syntaxe *K&R* des fonctions n'est plus utilisée par les programmeurs mais elle appartient toujours au standard et est donc toujours utilisable. Les types sont donnés après les noms :

```
TYPE fonction(v1, v2, ..., vN) TYPE v1; TYPE v2; ... TYPE vN; {
    // CODE
}
```

Par exemple, la déclaration suivante est correcte :

```
1 | int addition(a, b) int a; int b; {return a + b}
```

### 5.14.3 Digraphes et trigraphes

Les normes de caractères n'étaient pas aussi bien définies dans les années 80 que maintenant. En particulier, certaines normes régionales ne possédaient pas les caractères {, }, [, ] et #. Cela empêchait les programmeurs de ces pays de programmer en C. Pour cette raison, des suites de caractères spéciales appelées digraphes ont été ajoutés au langage pour remplacer ces caractères :

- < : pour [
- > : pour ]
- % : pour #
- <% pour {
- %> pour }

Écrire un de ces couples de symboles hors d'un identifiant C ou d'une chaîne de caractères l'identifie par sa correspondance. Le code suivant est donc tout à fait valide :

```
1 | %:include <stdlib.h>
2 | %:include <stdio.h>
3 |
```

15. Voir le boutisme en informatique : <https://fr.wikipedia.org/wiki/Boutisme>

```

4 | int main(int argc, char **argv) {
5 |     printf("%c", "Salut"<:0:>);
6 |     return EXIT_SUCCESS;
7 | }

```

Pour la même raison, le standard C inclue également des *trigraphes*, des séquences de trois caractères remplacés par le préprocesseur par le caractère correspondant à la compilation :

Trigraphe	Caractère
??=	#
??/	\
??'	^
??(	[
??)	]
??!	
??<	{
??>	}
??-	~

En particulier le programme suivant est tout à fait valide également :

```

1 | %:include <stdlib.h>
2 | %:include <stdio.h>
3 |
4 | int main(int argc, char **argv) ??<
5 |     // What's the fuck is happening ??????/?/
6 |     ceci fait toujours partie du commentaire ")=â- ç+qrh k poj )â-çtâç)"78320 É"'"/
7 |     et ça aussi LTMY ,<SMFL, SDRM LJRST +98602....'@9ççç$
8 |
9 |     printf("%c", "Salut"<:0:>);
10 |     return EXIT_SUCCESS;
11 |     ??>

```

Toutefois, cette possibilité offerte par le langage pour d'excellentes raisons a été supprimé dans la version C23<sup>16</sup>. Il faut donc forcer le compilateur à utiliser une version plus ancienne du langage :

```

user@computer ~/working_directory> gcc main.c -o main -std=c11 # version de 2011
user@computer ~/working_directory> ./main
S
user@computer ~/working_directory>

```

### 5.14.4 Un peu d'imagination

Ici, un exemple écrit par Basile Tonlorenzi :

```

1 | #include <stdio.h>
2 | #include <stdlib.h>
3 |
4 | #define AB int
5 | #define A ch

```

16. Malheureusement pour de pas si mauvaises raisons, mais certains n'ont visiblement pas le sens de l'humour...

```

6  %:define _ x
7  #define B ar
8  #define C A##B
9  ??=define P C
10
11 int f(x, y)
12     int** _;
13     int** y; <%return *0[y:>*(0<:x)];
14 }
15
16 int main(int argc, char** argv){
17     if(argc < 2){
18         printf("File UN chiffre entre 0 et 9 enculé \n"); // fleuri mais efficace
19         return EXIT_FAILURE;
20     }
21
22     C y = 0[1<:argv]:> - '0';
23     auto* a = &y ; // A changer #include .virgule
24     printf("%d \n", f(&a,&a));
25 }

```

Et un équivalent un peu plus technique :

```

1  #include <stdio.h>
2  #include <stdlib.h>
3
4  #define AB int
5  #define A ch
6  #define $(A) A,A
7  %:define _ x
8  #define B ar
9  ??=define C A##B
10
11 int f(x, y)
12     /* DO NOT DELETE THIS COMMENT LINE */
13     int** _; // What's the fuck is happening ???/?
14     while (y) {goto 5; continue; 5: y = -y;}
15     int** y; <%return *'\0'[y:>*\
16     (*( __LINE__-2*putchar('\b'))<:x)];
17     static int very_useful = 0x000a6425;
18 }
19
20 int main(int _, char** argv){
21     if(_ = !!!!(_-2)){
22         printf("File UN chiffre entre 0 et 9 enculé"); // fleuri mais efficace
23         return !!!!(_-2);
24     }
25     C y = _[~-1+1<:argv]:> - '0';
26     volatile (*__[1])($(C*)) = {f};
27     auto* a = &y;
28     (*__)($(&a));
29     __asm__ inline (
30         "lea 0x2D93(%rip), %rdi\n\t"
31         "mov %eax, %esi\n\t"
32         "xor %eax, %eax\n\t"

```

```
33 |     "call 0x1090\n\t"  
34 | );  
35 | }
```

Partie III

# LE VRAI MONDE DE LA RÉALITÉ RÉELLE



## INTRODUCTION



Cette troisième partie du cours d'informatique continue à utiliser le langage C comme support pédagogique, mais son contenu de fond appartient à un cadre beaucoup plus général de la programmation.

Si le/la lecteur/trice assidu(e) et consciencieux/se des deux parties précédentes doit maintenant maîtriser assez bien le langage C<sup>17</sup> d'un point de vue des connaissances, cela ne suffit pas à en faire un(e) développeur/se compétent(e). Il lui manque encore certains points fondamentaux qui agissent plus au niveau professionnel que personnel. La programmation n'est en général pas un travail absolument individuel puisqu'un programme est souvent destiné à d'autres qu'à soi.

Ont été régulièrement donnés quelques indices vis-à-vis du développement de projets de grandes tailles, et sur la nécessité d'une méthodologie d'approche qui permette de rester efficace/efficient sur le long terme, au fur-et-à-mesure que le projet devient trop gros ou grand pour être facilement pensé dans sa globalité par un unique individu.<sup>18</sup> Cette troisième partie se veut apporter quelques outils pour conserver une stabilité dans le développement d'applications complexes. Elle vise en particulier :

- la minimisation des erreurs incontrôlées *via* la gestion des erreurs et les tests unitaires
- l'internationalisation du code<sup>19</sup>
- l'utilisation efficiente des commentaires
- l'écriture de documentations
- le minimum des outils pour la gestion d'un projet informatique :
  - *Git*
  - *Markdown* et les *READMEs*
  - les licences informatiques

---

17. Ainsi que nombre de notions gravitant autour comme les représentations binaires des nombres ou quelques principes de la programmation système

18. À noter que cela concerne également les projets personnels de grande envergure puisque le développeur à un instant précis n'a pas une vision d'ensemble directement dans sa tête. Il lui a fallu découper cette vision sur des supports écrits et rester méthodique pour ne pas s'embourber dans le déluge de fonctionnalités et de caractéristiques du projet. On peut penser à l'écriture d'une documentation personnelle ou d'un micro-wiki dont la forme peut être quelconque comme des brouillons rassemblés décrivant le fonctionnement de certaines parties du système.

19. déjà évoqué en fin de première partie et début de deuxième

## CHAPITRE

# 6

# OUTILS UTILES À LA PROGRAMMATION



## LA DOCUMENTATION



### 6.1.1 Liste des sites d'information

*Wikipedia.org*

*man7.org*

Microsoft Doc

Zeste de Savoir

*os-dev.org*

*developpez.com*

### 6.1.2 Documentation hors-ligne

La commande *man* sous Linux

La commande *man* du terminal Linux est probablement la commande la plus utile au développeur C.

### 6.1.3 Lire un prototype

Abbreviations classiques : fd, buffer, etc...

### 6.1.4 Liste des sites de forum

[Stack Overflow](#)

[developpez.com](#)

### 6.1.5 À propos des GPTs

**Avis de l'auteur quant à l'utilisation d'intelligences artificielles de type LLM (*Large Language Models* en anglais)<sup>1</sup> :**

La programmation à l'aide de modèles de langage n'est utile que pour gagner du temps lors de l'écriture de code redondant ou pour l'écriture de documentation (processus en général très peu formateur, redondant et ennuyeux). En effet, l'écriture de ces textes ne demande aucune intelligence et n'apprend rien puisque tout est réfléchi par une intelligence tierce. Cela ne peut être remis en cause à mon avis.

Par voie de conséquence, il devient évident que l'utilisation de LLMs pour apprendre à programmer (j'entends par là : écrire à sa place le code<sup>2</sup>) est non seulement inutile mais contre-productif à toute apprentissage, puisque la maîtrise de notions complexes nécessite la maîtrise des fondamentaux, et que rien ne vaut pour véritablement comprendre et assimiler une chose que de la prendre entre quatre yeux et y passer le temps qu'il faut, malgré toute la fatigue que cela peut induire. Dans le cas de projets personnels, il doit être plus épanouissant de réussir à résoudre un problème par soi-même que de copier la solution d'un autre, et c'est de cette façon que non content d'une progression pérenne, le travailleur peut voir en son travail une source de bonheur.<sup>3</sup>

Quant à l'utilité de cet outil pour des problèmes :

1. dont la résolution sera *évaluée* de quelque manière que ce soit par une partie du corps enseignant
2. devant être fait dans le cadre d'un travail professionnel dans un temps très court et dans un pur but de productivité
3. pour toute autre raison pour laquelle l'apprentissage et la réflexion ne sont pas exigés, et sur un sujet pour lequel le développeur n'éprouve strictement aucun intérêt

, l'auteur :

1. jugeant le système construit autour de la notation particulièrement désuet et dépourvu d'attraits pédagogiques<sup>4</sup>
2. se trouvant être marxiste<sup>5</sup>
3. n'appréciant pas de se faire chier pour *aucune* raison

ne voit aucun mal à ChatGPT pour ce qui est de répondre aux exigences d'entités soit incompetentes<sup>6</sup> soit n'ayant aucune considération pour l'étudiant/le travailleur.

---

1. Personne ne me l'a demandé, mais bon... je vais probablement passer pour un moralisateur invétéré mais merde

2. Et pas demander une explication

3. Aucune influence ici d'un certain programme littéraire de classe préparatoire, c'est évident '-D

4. Si tant est que l'on puisse être "pédagogique" à forcer un élève dans un enseignement sans jamais tenter de l'intéresser autrement que par le désir d'un nombre plus ou moins élevé qui ne reflétera que très peu son intelligence ou son investissement. Enfin quoi ! Il doit y avoir d'autres moyens d'accrocher les étudiants aux cours, non ?

5. plutôt marxien pour ceux qui apprécient les nuances

6. Aucun rapport avec certains charlatans dont les existences ont conduit à l'écriture de ce document. Évidemment...





*make* est un logiciel installé par défaut sur les environnements Linux qui permet d'automatiser certaines tâches, en particulier des tâches de compilation de programmes. L'objectif de cette section est d'expliquer le principe général de fonctionnement de ce logiciel et de détailler son utilisation dans le cadre du développement logiciel en C.

### 6.2.1 Généralités

Les fichiers de production *make* ont par convention un des noms suivants :

1. GNUmakefile
2. make-file
3. Makefile

Il est possible de préciser un autre nom au logiciel *make*, mais par défaut, *make* ne prendra en considération qu'un fichier ayant un de ces noms. Il choisira le premier trouvé dans la liste ci-dessus, qu'il cherchera dans le *répertoire de travail*.

### 6.2.2 Principe et premiers exemples

*make* fonctionne sur le principe de règles de production. Ces règles sont constitués :

- d'une cible à produire
- de ressources nécessaires à la production de la cible
- d'actions à effectuer sur les ressources pour produire la cible

Un fichier *make* classique est construit de la manière suivante :

```
1  CIBLE: RESSOURCES # Un commentaire
2  ACTIONS
```

**Remarque :** L'espacement avant les actions est une *tabulation*. Une série d'espaces ne sera pas reconnue par *make*.

Pour utiliser *make*, on se place dans le répertoire de travail et on écrit :

```
user@computer ~/working_directory> make CIBLE
```

*make* va alors :

1. vérifier que les ressources nécessaires à la production de la cible sont bien présentes
2. si ce n'est pas le cas, il va les produire dans l'ordre les unes à la suite des autres
3. puis ensuite, il exécute les actions de productions de la cible

**Remarque :** Si aucune cible n'est précisé, *make* choisi la première du fichier *Makefile*.

Un exemple ultra-simple et inutile :

```

1  main: main.o # commence par chercher à produire 'main.o'
2      gcc main.o -o main # Produit 'main'
3
4  main.o: main.c # 'main.c' est déjà produit, donc inutile de le produire
5      gcc main.c -c # Produit 'main.o'

```

Un deuxième un peu moins inutile<sup>7</sup> :

```

1  main: main.o module1.o module2.o # commence par chercher à produire 'main.o'
2      gcc main.o module1.o module2.o -o main # Produit 'main'
3
4  main.o: main.c # 'main.c' est déjà produit, donc inutile de le produire
5      gcc main.c -c # Produit 'main.o'
6
7  module1.o: module1.c
8      gcc module1.c -c
9
10 module2.o: module2.c
11      gcc module2.c -c

```

qui donne l'exécution par *make* suivante :

```

user@computer ~/working_directory> make
gcc main.c -c
gcc module1.c -c
gcc module2.c -c
gcc main.o module1.o module2.o -o main # Produit 'main'
user@computer ~/working_directory>

```

Il semble inutile à chaque fois de commencer par produire les fichiers *objets* (d'extension *.o*) avant de produire le fichier exécutable alors que l'on pourrait très bien écrire :

```

1  main: main.c module1.c module2.c
2      gcc main.c module1.c module2.c -o main # Produit 'main'

```

Une petite explication s'impose, en trois points :

- *make* ne produit pas une cible si celle-ci est déjà à jour. On entend par à jour que la date de modification de la cible est plus récente que la date de modification de chacune de ses ressources. Ainsi, il est inutile de produire une seconde fois ce qui l'a déjà été.
- L'opération la plus lourde à la compilation est la génération des fichiers objets à partir des fichiers sources. L'édition de liens est très rapide en comparaison
- En précisant des règles de production séparées pour les fichiers objets, on garantit que seules les modules ayant subis des modifications sont recompilés. Les autres n'ont pas besoin de l'être

Dans le cadre de très gros projets, le gain de temps à la compilation est significatif.

### 6.2.3 Personnaliser la production grâce aux variables

Il est possible dans un fichier *make* de poser des *variables* pour faciliter la personnalisation d'une production. Lorsque le *Makefile* devient très complexe, il est plus facile de modifier ces "variables"

<sup>7</sup>. Mais toujours pas fou...

plutôt que de chercher dans le *Makefile* les endroits nécessitant la modification.

On définit une variable comme ceci :

```
| NOM = je suis un texte quelconque
```

et on y accède par la syntaxe :

```
| $(NOM)
```

**Remarque :** Ces variables n'ont de variables que le nom puisque leur valeur n'évolue pas au cours des productions.

Dans la pratique :

```
1  EXE_NAME = main
2  COMPILER = gcc
3  OBJS = main.o module1.o module2.o
4
5  $(EXE_NAME): $(OBJS)
6      $(COMPILER) $(OBJS) -o $(EXE_NAME) # Produit 'main'
7
8  main.o: main.c
9      $(COMPILER) main.c -c
10
11 module1.o: module1.c
12     $(COMPILER) module1.c -c
13
14 module2.o: module2.c
15     $(COMPILER) module2.c -c
```

## 6.2.4 Variables automatiques

Les variables automatiques sont des variables automatiquement créées dans les règles de production, qui évitent de devoir écrire des variables soi-même pour tout et n'importe quoi. On distingue les variables automatiques les plus utiles<sup>8</sup> :

Symbole	Signification
\$@	Nom de la cible
\$<	Nom de la première ressource
\$?	La liste des noms, séparés par des espaces, de toutes les ressources plus récentes que la cible
\$^	La liste des noms de toutes les ressources, séparés par des espaces

En utilisant ces variables, l'exemple ci-dessus se réécrit avec plus de concision :

```
1  EXE_NAME = main
2  COMPILER = gcc
3  OBJS = main.o module1.o module2.o
```

8. Voir [https://www.gnu.org/software/make/manual/html\\_node/Automatic-Variables.html](https://www.gnu.org/software/make/manual/html_node/Automatic-Variables.html) pour les autres.

```

4
5 $(EXE_NAME): $(OBJS)
6     $(COMPILER) $^ -o $@ # Produit 'main'
7
8 main.o: main.c
9     $(COMPILER) main.c -c
10
11 module1.o: module1.c
12     $(COMPILER) module1.c -c
13
14 module2.o: module2.c
15     $(COMPILER) module2.c -c

```

### 6.2.5 Réduire le nombre de règles avec la *stem*

Le symbole % désigne le “radical” *quelconque* d’un mot (en anglais *stem*, littéralement *tige*<sup>9</sup>). L’idée est de pouvoir considérer des cibles et des ressources quelconques pour éviter d’écrire trop de règles de production. En utilisant le symbole de radical, l’exemple ci-dessus devient :

```

1 EXE_NAME = main
2 COMPILER = gcc
3 OBJS = main.o module1.o module2.o
4
5 $(EXE_NAME): $(OBJS)
6     $(COMPILER) $^ -o $@ # Produit 'main'
7
8 %.o: %.c # cible finissant par .o qui utilise une ressource finissant par .c
9     $(COMPILER) $< -c

```

Le fichier *Makefile* donné ci-dessus commence à profiler un certain gain de temps. En effet, il suffit d’ajouter les noms des modules avec “.o” comme extension dans la variable *OBJS* et il n’y a plus ensuite pour compiler qu’à écrire *make*. Il n’y a donc plus à réécrire la liste des fichiers sources de module pour compiler.

### 6.2.6 Les caractères génériques

Il serait encore plus intéressant de ne rien avoir à faire du tout, c’est-à-dire que le *Makefile* s’occupe tout seul de détecter les fichiers sources. C’est à cela que servent les caractères génériques. *Ces caractères viennent du ô très ancien et honorable Bourne shell d’Unix*. Les caractères génériques servent à identifier des fichiers quelconques d’un répertoire de travail. On décrit ici seulement le caractère \*.<sup>10</sup> Celui-ci permet d’identifier n’importe quelle nom de fichier du répertoire de travail. Ainsi, “\*.c” désigne la liste des fichiers du répertoire de travail qui finissent par les deux caractères “.c”

Dans un *Makefile*, cela s’utilise de cette manière :

```

1 EXE_NAME = main
2 COMPILER = gcc
3

```

9. La traduction est de moi. Je ne sais pas comment traduire autrement

10. Voir <https://www.grymoire.com/Unix/Bourne.html#uh-4> pour les autres.

```

4 | $(EXE_NAME): *.c
5 |   $(COMPILER) $^ -o $@ # Produit 'main'

```

**Remarque :** Tel que présenté, il n’est pas possible d’écrire “\*.o” en ressources. En effet, aucun fichier objet n’est initialement présent. Le caractère générique \* ne détecte donc rien. Il n’y a pas de ressources, donc une erreur de production.

Pour faciliter la personnalisation, on peut mettre la liste dans une variable. Une petite difficulté cependant car les textes dans les variables ne sont pas directement interprétés. Ainsi :

```

1 | EXE_NAME = main
2 | COMPILER = gcc
3 | SRC = *.c
4 |
5 | $(EXE_NAME): $(SRC) # la généralisation est effectuée ici
6 |   $(COMPILER) $^ -o $@ # Produit 'main'

```

effectue bien le travail demandé, mais pas de la manière dont on pourrait le penser. En effet, on voudrait :

```
| SRC = main.c module1.c module2.c
```

ce qui n’est pas le cas. En fait, l’interprétation de \* est effectuée lorsque “\*.c” remplace *SRC* dans la ressource.

Pour forcer l’interprétation de \*, on utilise le mot-clé *wildcard* :

```

1 | EXE_NAME = main
2 | COMPILER = gcc
3 | SRC = $(wildcard *.c) # la généralisation est effectuée ici
4 |
5 | $(EXE_NAME): $(SRC)
6 |   $(COMPILER) $^ -o $@

```

## 6.2.7 Substitution de chaînes

Le problème rencontré dans la sous-section précédente est la perte de l’optimisation dû à la conservation des fichiers objets déjà compilés.

On aimerait pouvoir avoir la liste des fichiers objets à produire avant que ceux-ci ne le soit, et ce de manière automatisée. L’idée est alors simple :

1. lister les fichiers sources grâce au caractère générique \*
2. remplacer l’extension des noms de fichier dans la liste précédente par “.o” (au lieu de “.c”)

On utilise la syntaxe suivante :

```
| SRC = *.c
| OBJS = $(SRC:%.c=%.o)
```

qui signifie : “dans la variable *SRC*, chaque mot finissant par *.c* est remplacé par le même mot finit par *.o*”

Ce qui résoud effectivement le problème. On peut maintenant écrire le *Makefile* suivant :

```
1 EXE_NAME = main
2 COMPILER = gcc
3 SRC = $(wildcard *.c)
4 OBJS = $(SRC:%c=%.o)
5
6 $(EXE_NAME): $(OBJS)
7     $(COMPILER) $^ -o $@
8
9 %.o: %.c
10    $(COMPILER) $< -c
```

et ne plus jamais se préoccuper de quelle commande écrire pour compiler. Taper *make* dans le répertoire de travail suffit !

### 6.2.8 Les règles virtuelles

Certaines règles d'un *Makefile* peuvent simplement servir d'utilitaires pour le programmeur. Par exemple, on pourrait penser à une règle “*clean*” qui supprimerait tous les fichiers objets et l'exécutable pour réinitialiser l'espace de travail :

```
clean:
    rm -f *.o $(EXE_NAME)
```

Si cette règle est par mégarde appelée par *make*, celui-ci va croire qu'une erreur aura survenue puisque l'action ne produit pas la cible. Il faut donc préciser que cette règle est fausse à *make* (*phony* en anglais) :

```
clean:
    rm -f *.o $(EXE_NAME)

.PHONY: clean
```

On peut alors appeler cette règle :

```
user@computer ~/working_directory> make clean
rm -f *.o main
user@computer ~/working_directory> make
gcc main.c -c
gcc module1.c -c
gcc module2.c -c
gcc main.o module1.o module2.o -o main
user@computer ~/working_directory>
```

### 6.2.9 Idées pour aller plus loin

Cette sous-section ne présente pas des particularités de *make* mais simplement quelques “idées” pour travailler dans un espace un peu plus propre.

#### Arborescence du répertoire de travail

Il peut être intéressant :

- d’avoir un répertoire *bin* qui contient la sortie binaire de la compilation, c’est-à-dire l’exécutable
- de séparer les fichiers sources et les fichiers d’entêtes tant entre eux que vis-à-vis des autres fichiers du projet
- de séparer les fichiers objets générés pour qu’ils ne gênent pas

On peut pour la séparation des fichiers sources et des fichiers d’entêtes utiliser le paramètre `-I` du compilateur, qui permet d’indiquer un répertoire de fichier d’entête supplémentaire au répertoire courant.

#### Ajout de la gestion de bibliothèques externes

On peut ajouter par défaut le paramètre `-lm` pour utiliser la bibliothèque `math.h`.

On peut ajouter un répertoire *lib* au projet pour y stocker des bibliothèques externes (d’extension “.a” ou “.so”)



## CRÉER SA BIBLIOTHÈQUE EXTERNE



## UTILISER UN DÉBOGUEUR



Le déboguage est une composante fondamentale du développement logiciel. En effet, il est fondamentale de pouvoir identifier les fautes dans un code, en particulier si celui-ci rencontre des erreurs fatales à son fonctionnement (i.e. *crash*). *GDB* (*GNU DeBugger*) est un logiciel qui permet d’opérer le déboguage de programmes complexes. Seront détaillés dans la suite les fonctionnalités basiques qu’il propose.

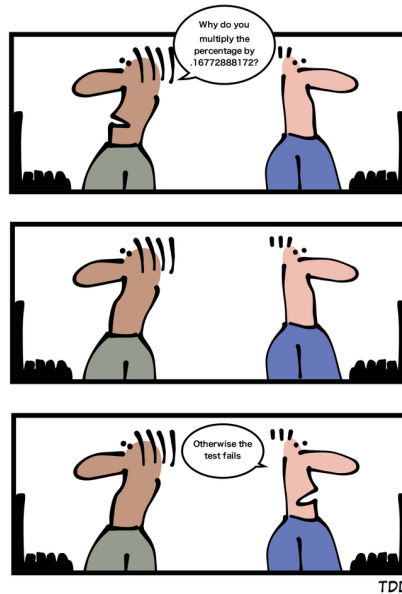
## CHAPITRE

7

# MÉTHODES DE PROGRAMMATION PROPRE



## LA GESTION DES ERREURS





La gestion d'erreurs en programmation est absolument fondamentale. Les raisons sont multiples :

- un ordinateur physique n'est pas théorique, et un programme subit tous les défauts de l'implantation. On peut par exemple penser aux opérations sur les nombres flottants qui induisent une imprécision. Les résultats obtenues sur les flottants sont donc dans la majorité des cas faux pour la modélisation de nombres réels
- les programmes développés sur un ordinateur ont été écrits par des humaines faillibles. Ainsi, il n'est jamais sûr à 100% qu'un programme écrit soit correct, bien qu'il soit démontré correct en théorie.

Bien sûr, dans le cas de “petits” programmes, ne pas gérer les erreurs n'a qu'une incidence minime sur le développement. Dans le cas de très grands projets, des erreurs humaines ont de fortes chances de se glisser dans certaines étapes du développement et les tester et les gérer est extrêmement important pour éviter des bogues incontrôlés et surtout non localisés.

Les manquements à la gestion des erreurs et aux tests sont la source première des failles de sécurité des systèmes informatiques.<sup>1</sup>

### 7.1.1 Un petit exemple pour prendre la température

Lorsque l'on commence à programmer, en C notamment, les erreurs de segmentation sont parmi les plus fréquentes.<sup>2</sup> Pour les réduire, il faut au maximum réduire les probabilités de manipuler des pointeurs non initialisés. Pour cela, il faudrait tester après chaque initialisation de pointeur (par un malloc par exemple) si le pointeur est effectivement non NULL.

Avec un malloc cela ressemblerait à :

```
int *x = NULL;
x = (int *)malloc(10 * sizeof(int));
if (!x) {
    fprintf(stderr, "Erreur d'allocation memoire !\n");
}
```

En toute rigueur, c'est ce qu'il faudrait faire. En effet, si l'ordinateur ne dispose plus d'assez de mémoire vive pour allouer dynamiquement la variable, malloc renverra NULL. Dans les faits, effectuer un test à chaque allocation est assez lent. Si on sait qu'on alloue pas 3 ou 4 *Go* d'un coup, la probabilité d'erreur est furieusement proche de zéro. Et si le système d'exploitation n'arrive pas à allouer une centaine d'octets dynamiquement, il y a fort à parier qu'il ne puisse rien faire du tout de toute manière et qu'une erreur de segmentation n'ait aucune incidence dans la pratique.

**Remarque :** Dans le cas de la programmation de systèmes embarqués, le paragraphe précédent est nul et non avenu puisque les systèmes embarqués doivent fonctionner avec des ressources parfois extrêmement limités. La question est alors plus de trouver une autre manière de faire qui optimise la quantité de mémoire utilisée.

La gestion des erreurs est donc dépendante du contexte.

1. Le test logiciel fait malgré tout partie (de mon point de vue subjectif) des domaines de l'informatique pratique les plus atrocement chiantes dont j'ai pu faire l'expérience... Je veux dire par chiant : un ennui morbide qui fabrique des boules de déprime calées devant des écrans à se demander toutes les cinq secondes l'heure qu'il est ... Et le pire, c'est l'automatisation de tests logiciels en PowerShell sous Windows. Ça c'est particulièrement horrible.

2. Ah bon ?

### 7.1.2 Codes d'erreur

Les routines de la bibliothèque standard du langage C renvoient une valeur indiquant le comportement de l'exécution de ladite routine. Ainsi, il est possible de contrôler le bon fonctionnement d'un grand nombre de fonctions de la bibliothèque standard.

Par exemple, l'ouverture d'un fichier grâce à `fopen` peut échouer car il n'existe pas de fichier accessible par le chemin indiqué ou encore car un autre programme utilise déjà ce fichier. Le test :

```
const char* filename = ...;
FILE *f = fopen(filename, "mode");
if (!f) {
    fprintf(stderr, "Erreur d'ouverture du fichier %s\n", filename);
} else {
    ...
}
```

permet d'effectuer le contrôle de la bon ouverture du fichier. En effet, `fopen` renvoie `NULL` si la fonction échoue.

La gestion d'erreur dépend de chaque fonction. Ainsi, les fonctions `fread` et `fwrite` renvoient le nombre de caractères lus ou écrits. C'est donc en contrôlant ce nombre que l'on sait si la fonction a échoué. La lecture du manuel est ici indispensable.



## TESTS UNITAIRES



C'est pour cela que les tests unitaires existent. L'objectif des programmes de tests unitaires est de vérifier des routines isolées (unitaires) dans des cas particulier d'exécution qui pourrait potentiellement amener à une erreur en comparant la sortie de chaque routine sur une entrée  $e$  prédéterminée avec une sortie attendue  $s(e)$ . Il faut alors tester tous les entrées les plus particulières.

Chaque routine d'un programme doit en théorie être testée par des tests unitaires. Dans la pratique, du fait des ressources humaines nécessaires à la réalisation de ces tests, on ne test que les routines "non triviales" d'un programme, c'est-à-dire celles qui peuvent présenter des fautes potentielles.

L'importance des tests unitaires est multiple :

- Correction efficiente des bogues : une routine bogué est plus facilement corrigable tôt que lorsque la moitié du programme en est dépendant.
- Documentation : les tests unitaires servent d'exemples sur la manière dont les routines doivent être utilisées
- Confiance en la qualité du code



## À PROPOS DU STYLE



Partie IV

# ANNEXES

## CHAPITRE

# 8

## CORRECTION DES 42 EXERCICES

Certains exercices sont vraiment difficiles, et ce n'est qu'en y passant du temps que l'on progresse le plus au niveau du raisonnement et des réflexes. N'hésitez pas !

La solution à chaque exercice n'est pas unique. Il est donc possible souvent de trouver une solution qui diffère. Il peut être intéressant de se poser la question de la raison des différences entre les deux solutions (optimalité, lisibilité, rigueur, etc...). Il n'est pas garanti que les programmes proposés ci-dessous soient optimales, que ce soit par l'optimisation des constantes, la taille du code, ou d'autres paramètres. Il est d'ailleurs rare de pouvoir optimiser un programme selon toutes les contraintes<sup>1</sup>



### PREMIÈRE PARTIE



On pose pour la suite  $\mathcal{B} = \{0, 1\}$  et  $N \in \mathbb{N}^*$

#### 8.1.1 Opérations logiques sur les mots binaires

**Solution 1 (La compréhension pour mieux comprendre).** On peut établir un lien entre les opérations logiques élémentaires et les opérations élémentaires en théorie des ensembles (union, union disjointe, intersection, privation). En effet :

- $A \cup B = \{e \mid e \in A \vee e \in B\}$ , on a :  $\mathbb{P}(e \in A \cup B) = \mathbb{P}(e \in A) + \mathbb{P}(e \in B) - \mathbb{P}(e \in (A \cap B))$
- $A \oplus B = \{e \mid e \in A \oplus e \in B\}$ , on a :  $\mathbb{P}(e \in A \oplus B) = \mathbb{P}(e \in A) + \mathbb{P}(e \in B) - 2 \times \mathbb{P}(e \in (A \cap B))$
- $A \cap B = \{e \mid e \in A \wedge e \in B\}$ , on a :  $\mathbb{P}(e \in A \cap B) = \mathbb{P}(e \in A)\mathbb{P}(e \in B)$
- $\Omega \setminus A = \{e \mid \neg e \in A\}$ , on a  $\mathbb{P}(e \in \Omega \setminus A) = 1 - \mathbb{P}(e \in A)$

Il semble donc intuitif de retrouver des expressions semblables :

---

1. Par exemple, un programme parfois plus gros en mémoire peut-être plus rapide à l'exécution.

- $\vee : (a, b) \mapsto a + b - ab$
- $\oplus : (a, b) \mapsto |a - b|$
- $\wedge : (a, b) \mapsto a \times b$
- $\neg : a \mapsto 1 - a$

**Solution 2 (Universalité des fonctions logiques élémentaires).** On considère un opérateur logique  $*$   $N$ -aire de  $\mathcal{B}^N$  dans  $\mathcal{B}^M$  où  $M \in \mathbb{N}^*$ . Soit  $e \in \mathcal{B}^N$ . On note  $*(e) = *_{M-1}(e) *_{M-2}(e) \cdots *_0(e)$ . On observe que  $*$  est entièrement définie par les fonctions  $*_i, i \in \llbracket 0, M-1 \rrbracket$  de  $\mathcal{B}^N$  dans  $\mathcal{B}$ .

Montrons donc que toute fonction logique de  $\mathcal{B}^N$  dans  $\mathcal{B}$  peut s'écrire comme une combinaison des fonctions  $\vee, \wedge$  et  $\neg$ .

On peut partitionner  $\mathcal{B}^N$  en deux ensembles disjoints  $T$  et  $F$  tels que  $*$  =  $\mathbb{1}_T$ . C'est-à-dire que  $T = \{e \in \mathcal{B}^N \mid *(e) = 1\} = \{t_1, \dots, t_{\text{Card}(T)}\}$ .

D'où, en posant que l'égalité de deux éléments vaut 1 et leur différence vaut 0<sup>2</sup> :

$$*(e) = 1 \Leftrightarrow \bigvee_{i=1}^{\text{Card}(T)} (e = t_i) \quad (8.1)$$

On note  $e = e_{N-1} \dots e_0$  et pour tout  $i \in \llbracket 1, k \rrbracket$ ,  $t_i = t_{i,N-1} \dots t_{i,0}$ .

On remarque que pour avoir  $e = t_i$ , il faut que :

- les 1 de  $e$  soient les 1 de  $t_i$
- les 0 de  $e$  soient les 0 de  $t_i$ .

On note  $K_1(t_i) = \{k \in \llbracket 0, N-1 \rrbracket \mid t_{i,k} = 1\}$  et  $K_0(t_i) = \{k \in \llbracket 0, N-1 \rrbracket \mid t_{i,k} = 0\}$ .

Alors :

$$\begin{aligned} e = t_i \Leftrightarrow & \bigwedge_{k \in K_1(t_i)} (e_k = t_{i,k}) \bigwedge_{k \in K_0(t_i)} (e_k = t_{i,k}) \\ & \bigwedge_{k \in K_1(t_i)} (e_k = 1) \bigwedge_{k \in K_0(t_i)} (e_k = 0) \\ & \bigwedge_{k \in K_1(t_i)} e_k \bigwedge_{k \in K_0(t_i)} \neg e_k \end{aligned}$$

Finalement, on arrive à la définition par compréhension de  $*$  :

$$\begin{aligned} * : \mathcal{B}^N & \rightarrow \mathcal{B} \\ (e_{N-1}, \dots, e_0) & \mapsto \bigvee_{i=1}^{\text{Card}(T)} \left( \bigwedge_{k \in K_1(t_i)} e_k \bigwedge_{k \in K_0(t_i)} \neg e_k \right) \end{aligned}$$

Les informations de la fonction sont contenues dans  $T$ .

**Remarque :**  $\mathcal{B}^N$  est en bijection avec  $\llbracket 0, 2^N - 1 \rrbracket$ . Les opérations  $\vee, \wedge$  et  $\neg$  permettent donc de définir n'importe quelle fonction sur  $\llbracket 0, 2^N - 1 \rrbracket$  d'arité finie, donc par exemple l'addition, la multiplication, etc...<sup>3</sup>

2. On explicite dans la suite l'expression de  $e = t_i$ .

3. On utilise évidemment pas cette formule pour définir les opérateurs arithmétiques. Il s'agit seulement de la preuve que ces opérateurs logiques sont capables de telles définitions.

## À propos des formes normales

Dans l'expression précédente, les  $e_k$  sont appelées des variables propositionnelles en logique propositionnelle (ou d'ordre 0). Elles n'ont que deux interprétations possibles, *Vrai* ou *Faux*, 1 ou 0. La proposition logique associée à l'expression de la fonction logique admet des valeurs de vérités qui sont les couples  $V = (v_{N-1}, \dots, v_0)$  appelés *interprétations* tels que  $*(V) = \text{Vrai}$ . On peut montrer qu'une proposition logique en logique propositionnelle peut s'écrire de manière équivalente sous deux formes :

- la forme normale disjonctive qui est une disjonction de conjonctions de variables :

$$\bigvee \left( \bigwedge x_i \right)$$

- la forme normale conjonctive qui est une conjonction de disjonctions de variables :

$$\bigwedge \left( \bigvee x_i \right)$$

En électronique, ces deux formes sont nommées respectivement *somme de produits* et *produit de sommes* en raison des expressions trouvés à l' **Exercice 1.** .

**Solution 3 (Porte NAND).** La table de vérité de NAND est :

$A$	$B$	$A \uparrow B$
0	0	1
0	1	1
1	0	1
1	1	0

On a ensuite pour tout  $a, b \in \mathcal{B}$  :

- $\neg(a) = a \uparrow a$
- $a \wedge b = \neg a \uparrow b = (a \uparrow b) \uparrow (a \uparrow b)$
- $a \vee b = \neg(\neg a \wedge \neg b) = \neg a \uparrow \neg b = (a \uparrow a) \uparrow (b \uparrow b)$

**Solution 4 (Porte NOR).** La table de vérité de NOR est :

$A$	$B$	$A \downarrow B$
0	0	1
0	1	0
1	0	0
1	1	0

On a ensuite pour tout  $a, b \in \mathcal{B}$  :

- $\neg a = a \downarrow a$
- $a \vee b = \neg(a \downarrow b) = (a \downarrow b) \downarrow (a \downarrow b)$
- $a \wedge b = (\neg a) \downarrow (\neg b) = (a \downarrow a) \downarrow (b \downarrow b)$

**Solution 5 (Petit retour à l'algèbre fondamentale).** Comme  $\oplus$  est une opération *bit à bit*, il suffit de vérifier que  $G = (\mathcal{B}, \oplus)$  est un groupe commutatif.

- $\oplus$  est une fonction de  $\mathcal{B}^2$  dans  $\mathcal{B}$  il s'agit bien d'une loi de composition interne.
- On observe que  $0 \oplus 0 = 0$  et  $0 \oplus 1 = 1$  et  $1 \oplus 0 = 1$ . Donc 0 est un élément neutre par  $\oplus$ . Comme  $1 \oplus 1 = 0$ , 0 est le seul élément neutre.
- $\oplus$  est associative

- Pour tout  $x \in \mathcal{B}$ ,  $x \oplus x = 0$  donc  $x^{-1} = x$
- $\oplus$  est commutative

Donc  $G$  est bien un groupe commutatif.

**Solution 6 (Inversibilité des décalages).** Les décalages ne sont pas inversibles. En effet, quand des bits sont supprimés par un décalage de trop de bits, l'information est perdue. Un décalage dans l'autre sens ne produit que des 0. Par exemple :  $(01000 \ll 2) \gg 2 = 00000 \neq 01000$

### 8.1.2 Opérations arithmétiques

**Solution 7 (Conversion en binaire).**

- |                  |                  |
|------------------|------------------|
| ■ $(01001100)_2$ | ■ $(11010011)_2$ |
| ■ $(10111100)_2$ | ■ $(00000100)_2$ |
| ■ $(00100001)_2$ | ■ $(11101110)_2$ |
| ■ $(01101101)_2$ | ■ $(10100001)_2$ |
| ■ $(01011100)_2$ | ■ $(01111110)_2$ |

### 8.1.3 Opérations sur les nombres à virgules

**Solution 8 (Écriture en base 2 de nombres non entiers).**

- $(1110.1001)_2$
- $(1.01101)_2$
- $(111.0111)_2$
- $(0.0001001001\dots)_2$  : il existe un nombre infini de décimales. On a le même phénomène avec la représentation de  $\frac{1}{3}$  en décimal.

### 8.1.4 Représentation hexadécimale

**Solution 9 (Conversion binaire-hexadécimale).**

- $713705 = (1010\ 1110\ 0011\ 1110\ 1001)_2 = 0xAE3E9$
- $8.8 = (1.00011001001001001001001)_2 \times 2^3 = 0100\ 0001\ 1000\ 1100\ 1001\ 0010\ 0100\ 1001 = 0x418C9249$
- $42 = (0001\ 1010)_2 = 0x1A$
- $-1.1 = -(1.00010010010010010010010)_2 \times 2^1 = 1100\ 0000\ 0000\ 1001\ 0010\ 0100\ 1001\ 0010 = 0xC0092492$
- $101 = (0110\ 0101)_2 = 0x65$



## DEUXIÈME PARTIE



### 8.2.1 Bases du langage

**Variables**

**Solution 10 (Interversion de variables par effet de bord).**

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  int main() {
5      int a = 8;
6      int b = 6;
7
8      // the swap :
9      int tmp = a;
10     a = b;
11     b = tmp;
12
13     return EXIT_SUCCESS;
14 }
```

### Formatage de chaînes de caractères

#### Solution 11 (Taille des types).

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  int main() {
5      printf("sizeof(char) = %ld\n", sizeof(char));
6      printf("sizeof(short int) = %ld\n", sizeof(short int));
7      printf("sizeof(int) = %ld\n", sizeof(int));
8      printf("sizeof(long int) = %ld\n", sizeof(long int));
9      printf("sizeof(long long int) = %ld\n", sizeof(long long int));
10     printf("sizeof(float) = %ld\n", sizeof(float));
11     printf("sizeof(double) = %ld\n", sizeof(double));
12     printf("sizeof(long double) = %ld\n", sizeof(long double));
13     printf("sizeof(pointer) = %ld\n", sizeof(void*));
14
15     return EXIT_SUCCESS;
16 }
```

### Opérateurs sur les variables

#### Solution 12 (Valeur).

Le premier code est strictement équivalent au code suivant :

```
1  int i = 10;
2  i = i - i;
3  i--;
```

D'où  $i = -1$  en fin d'exécution. Le second code est strictement équivalent au code suivant :

```
1  int i = 10;
2  i--;
3  i = i - i;
```



D'où  $i = 0$  en fin d'exécution.

**Solution 13 (Calcul d'expressions).**

- $a = 55$
- $b = 41$
- $c = 93$  (on évalue d'abord la division)
- $d = 65522$  ( $93 - 107 = 0 - 14 \equiv 65535 - 13[65536]$ )

**Solution 14 (Priorité des opérateurs).**

```

1  int a = 6, b = 12, c = 24;
2  a = 25*12 + b;
3  printf("%d", a > 4 && b == 18);
4  (a >= 6&& b < 18) || c != 18;
5  c = a = b + 10;
```

**Solution 15 (Intervention sans effet de bord (1)).**

```

1  #include <stdio.h>
2  #include <stdlib.h>
3
4  int main() {
5      int a = ...; // any value
6      int b = ...; // any value
7
8      // the swap :
9      a = a + b;
10     b = a - b; // b = a + b - b = a
11     a = a - b; // a = a + b - a = b
12
13     return EXIT_SUCCESS;
14 }
```

On note  $a_p$  et  $b_p$  les valeurs de  $a$  et  $b$  précédant l'intervention, et  $a_s$  et  $b_s$  les valeurs de  $a$  et  $b$  succédant à l'intervention. Les opérateurs  $+$  et  $-$  conservent l'écriture binaire des mots. L'échange est donc correct quelques soient les signatures de  $a$  et  $b$ . Avec  $a$  sur  $N_a$  bits et  $b$  sur  $N_b$  bits avec  $N_a > N_b$ , on peut choisir  $a = 2^{N_b}$  et  $b$  quelconque. Alors  $a + b = 2^{N_b} + b$  puis  $b_s = 2^{N_b} + b_p - b_p = 2^{N_b} = -1 \neq a$ . La permutation est donc incorrecte.

Cela peut être justifié sans calcul par l'observation suivante : les bits  $a_{N_a-1} \dots a_{N_b}$  sont perdus par la modulation par  $2^{N_b}$ . La permutation ne peut donc être correcte.

**Solution 16 (Intervention sans effet de bord (2)).**

```

1  #include <stdio.h>
2  #include <stdlib.h>
3
4  int main() {
5      int a = ...;
6      int b = ...;
7
8      a = a ^ b;
9      b = a ^ b;
```

```

10 |     a = a ^ b;
11 |
12 |     return EXIT_SUCCESS;
13 | }

```

En effet, on rappelle que pour tout  $x \in \mathcal{B}^N$ ,  $x \oplus x = 0$  et  $x \oplus 0 = x$ . Ainsi,  $b_s = a \oplus b \oplus b = a$  et  $a_s = a \oplus b \oplus a = b$ .

**Solution 17 (Multiplication par décalage).** On observe que  $14 = (1110)_2 = 2^3 + 2^2 + 2^1 = (1 \ll 3) + (1 \ll 2) + (1 \ll 1)$ . Il est toutefois possible de faire mieux en prêtant plus attention à la représentation binaire de 14. En effet,  $14 + 1 = (1111)_2 = 16 - 1$ , c'est-à-dire  $14 = 16 - 2 = 2^4 - 2^1 = (1 \ll 4) - (1 \ll 1)$

```

1 | #include <stdio.h>
2 | #include <stdlib.h>
3 |
4 | int main() {
5 |     int a = 57;
6 |     int r = (a << 4) - (a << 1);
7 |     printf("57x14 = %d", r);
8 |     return EXIT_SUCCESS;
9 | }

```

Il est ainsi possible, parfois, d'optimiser l'opération de multiplication grâce à cette technique. La méthode classique de la multiplication est en effet plus lente à calculer, bien que plus générale. Cela est particulièrement visible lors de la multiplication de grands nombres :

$$987654321 \times 65534 = (987654321 \ll 16) - (987654321 \ll 1)$$

Seules 2 décalages et une soustraction sont effectuées ( $65534 = 65536 - 2 = 2^{16} - 2^1$ )

**Solution 18 (Valeur absolue).** On utilise la particularité qu'un décalage logique (effectué sur unsigned int) n'effectue pas d'extension du signe :

```

1 | #include <stdio.h>
2 | #include <stdlib.h>
3 |
4 | int main() {
5 |     int x = -5; // any positive or negative value
6 |     unsigned int abs_x = x;
7 |     unsigned int positive = (abs_x >> 31) - 1; // -1 if positive, 0 if negative
8 |     abs_x = ((~positive) & (-x)) | (positive & x);
9 |     printf("|%d| = %u", x, abs_x);
10 |    return EXIT_SUCCESS;
11 | }

```

Par ailleurs,  $-1 = 0xFFFFFFFF$ , donc  $-1 \& x == x$  est toujours vrai et  $0 \& x == 0$  est toujours vrai.

### Projection de type

**Solution 19 (Quelques évaluations).**

1. vraie car  $0xFFFFFFFF + 1 = 0$  (e1 est sur 32 bits)

2. *faux* car  $(\text{unsigned char})(-1) = 255 = 0xFF$ , d'où l'expression est égale à 0.
3. *vraie* car on a  $!(e1 == e2) \equiv e1 != e2$  qui est vraie.
4. *vraie* car  $64 \wedge e3 = 65 \equiv 1[8]$ . D'où  $!(((64 \wedge e3) \% 8) - 1) = 1$ .  
Alors :  $e4 \equiv (\text{unsigned char})(257) - 2 = -1$

### Structures de contrôle du flot d'exécution

Solution 20 ([COURS]).

```

1  #include <stdio.h>
2  #include <stdlib.h>
3
4  #define N ...
5
6  int main() {
7      int u = 3;
8      unsigned int n = 0;
9      while (n < N) {
10         u = 3*u*u - 5*u + 2;
11         n++;
12         printf("u(%u) = %d\n", n, u);
13     }
14     return EXIT_SUCCESS;
15 }
```

À partir de  $u_4$  inclut, les valeurs de la suite sont fausses. On observe ainsi que  $u_3 = 808602$ , d'où  $u_4 = 3 \times 808602^2 - 5 \times 808602 + 2 = 1961507540204 > 2^{31} - 1 = 2147483647$ . Il y a un dépassement de capacité car la variable  $u$  est stocké sur trop peu de bits. Le résultat est donc correct modulo  $2^{32}$

Solution 21 (Question d'âge).

```

1  #include <stdio.h>
2  #include <stdlib.h>
3
4  int main() {
5      signed char age = 18;
6      if (age < 0) {
7          printf("Soyez patient(e), votre tour arrive");
8          return EXIT_SUCCESS;
9      } else if (age < 18) {
10         printf("Mineur");
11         return EXIT_SUCCESS;
12     } else if (age == 18) {
13         printf("Tout juste majeur");
14         return EXIT_SUCCESS;
15     } else {
16         printf("Majeur");
17         return EXIT_SUCCESS;
18     }
19     printf("Et la surpopulation alors ?");
20     return EXIT_SUCCESS;
21 }
```

Solution 22 (Suite de Fibonacci).

```

1  #include <stdio.h>
2  #include <stdlib.h>
3
4  #define N ...
5
6  int main() {
7      unsigned int n = N;
8      unsigned int f_m_1 = 0;
9      unsigned int f_m_2 = 1;
10     unsigned int tmp;
11     for (; n > 0; n--) {
12         tmp = f_m_1;
13         f_m_1 = f_m_1 + f_m_2;
14         f_m_2 = tmp;
15     }
16     printf("f(%u) = %u", N, f_m_1);
17     return EXIT_SUCCESS;
18 }

```

### Solution 23 (Nombres premiers).

Par définition, un nombre  $n$  est premier si il possède exactement deux diviseurs distincts :  $n$  et 1. On peut donc tester la primalité d'un nombre en vérifiant qu'il ne possède pas d'autres diviseurs positifs strictement supérieurs à 1 que lui-même. Un premier algorithme naïf<sup>4</sup> est donc le suivant :

```

1  #include <stdio.h>
2  #include <stdlib.h>
3
4  int main() {
5      unsigned int n;
6      printf("Saisir un nombre entier : ");
7      scanf("%u", &n);
8
9      char is_primal = n == 2 || (n > 2 && n % 2 != 0); // necessary
10     for (int i = 3; i < n-1 && is_primal; i += 2) {
11         is_primal = (n % i == 0) ? 0 : is_primal;
12     }
13
14     if (is_primal) {
15         printf("Premier !\n");
16     } else {
17         printf("Pas premier...\n");
18     }
19
20     return EXIT_SUCCESS;
21 }

```

Considérons maintenant un diviseur  $d$  de  $n$ . Il existe donc  $q$  tel que  $n = dq$ . On observe que si  $d \geq \sqrt{n}$  alors  $q \leq \sqrt{n}$ . Par ailleurs,  $q$  est un diviseur de  $n$  lui aussi. Supposons que l'on ait vérifié que pour tout  $1 < m \leq \sqrt{n}$ ,  $m \nmid n$ , alors il est absurde qu'il existe un diviseur de  $n$  supérieur strictement à  $\sqrt{n}$  car il existerait alors un diviseur  $d$  strictement inférieur à  $\sqrt{n}$ .

---

4. Enfin, pas tout à fait puisqu'en ignorant les nombres pairs, on obtient seulement  $\frac{N}{2}$  tours. Mais cela est équivalent asymptotiquement (les constantes restant des constantes).

Il suffit donc de vérifier la divisibilité de  $n$  par les entiers  $m \leq \sqrt{n}$  :

```

10 | for (int i = 3; i*i <= n && is_primal; i++) {
11 |     is_primal = (n % i == 0) ? 0 : is_primal;
12 | }

```

On a  $\frac{n}{\sqrt{n}} \xrightarrow{n \rightarrow \infty} \infty$ , l'algorithme est donc plus efficace (et il ne s'agit pas d'une constante).

## Routines

### Solution 24 (Encore Fibonacci).

```

1 | #include <stdio.h>
2 | #include <stdlib.h>
3 |
4 | unsigned int fibo(int n) {
5 |     unsigned int f_m_1 = 0;
6 |     unsigned int f_m_2 = 1;
7 |     unsigned int tmp;
8 |     for (; n > 0; n--) {
9 |         tmp = f_m_1;
10 |        f_m_1 = f_m_1 + f_m_2;
11 |        f_m_2 = tmp;
12 |    }
13 |    return f_m_1;
14 | }
15 | int main() {
16 |     for (unsigned int n = 0; n <= 100; n++) {
17 |         printf("fibo(%u) = %u\n", n, fibo(n));
18 |     }
19 |     return EXIT_SUCCESS;
20 | }

```

### Solution 25 (Puissances entières).

```

1 | long int pow(int x, unsigned int n) {
2 |     long int p = 1;
3 |     for (; n > 0; n--) {
4 |         p *= x;
5 |     }
6 |     return p;
7 | }

```

On observe que  $(2^{31} - 1)^{2^{32}-1} \approx 2^{31 \times 2^{32}}$  est très largement supérieur à  $2^{63} - 1$  qui est la valeur maximale d'un `long int`. En fait, pour stocker le résultat maximal de la fonction `pow`, il faudrait un disque dur d'environ  $4 \times 31 = 124$  Go. Par ailleurs, la fonction `pow` reste infiniment loin d'être surjective dans  $\mathbb{Z}$ .

## Pointeurs

### Solution 26 (Quelques procédures inutiles pour devenir un bot efficace).

```

1 void mov(int *x, int val) {
2     *x = val;
3 }
4
5 void add(int *a, int *b) {
6     *a += *b;
7 }
8
9 void mul(int *y, int *z) {
10    *y *= *z;
11 }
12
13 void pow(int *x, int n) {
14     int p = 1;
15     for (; n > 0; n--) {
16         p *= *x;
17     }
18     *x = p;
19 }

```

### Solution 27 (Interversion sans effet de bord (3)).

```

1 void swap(int *x, int *y) {
2     if (x == y) {
3         return;
4     }
5     *x = *x ^ *y;
6     *y = *x ^ *y;
7     *x = *x ^ *y;
8 }

```

En effet, si  $x = y$ , à la première ligne, on a  $*x = *x * x = 0$ , et  $*y = *x = 0$ . Les autres lignes sont inutiles, mais chacune a le même effet que la première.

### Solution 28 (Distance de Manhattan).

```

1 #include <stdio.h>
2 #include <stdlib.h>
3
4 double coords_to_point(float x, float y) {
5     double p;
6     float *ptr = (float *)&p;
7     ptr[0] = x; // *ptr = x;
8     ptr[1] = y; // *(ptr + 1) = y;
9     return p;
10 }
11
12 double subtract(double pA, double pB) {
13     float *ptrA = (float *)&pA;
14     float *ptrB = (float *)&pB;
15     float x = ptrA[0] - ptrB[0]; // *ptrA - *ptrB
16     float y = ptrA[1] - ptrB[1]; // *(ptrA + 1) - *(ptrB + 1)
17     return coords_to_point(x, y);
18 }

```

```

19
20 float absolute(float x) {
21     if (x < 0) {
22         return -x;
23     }
24     return x;
25 }
26
27 float norme1(double p) {
28     float *ptr = (float *)&p;
29     return absolute(ptr[0]) + absolute(ptr[1]);
30 }
31
32 float distance(double pA, double pB) {
33     return norme1(subtract(pA, pB));
34 }
35
36 int main() {
37     double pA = coords_to_point(3.0, 4.0);
38     double pB = coords_to_point(-1.0, -1.0);
39     printf("%f\n", distance(pA, pB)); // prints 9.000000
40     return EXIT_SUCCESS;
41 }

```

### Interagir avec les flux standards

#### Solution 29 (Distance Euclidienne).

```

1  #include <stdio.h>
2  #include <stdlib.h>
3
4  int main() {
5      double xA, xB, yA, yB;
6      printf("Entrer xA : ");
7      scanf("%lf", &xA);
8      printf("Entrer yA : ");
9      scanf("%lf", &yA);
10     printf("Entrer xB : ");
11     scanf("%lf", &xB);
12     printf("Entrer yB : ");
13     scanf("%lf", &yB);
14     printf("||A - B||^2 = %lf", (xA - xB)*(xA - xB) + (yA - yB)*(yA - yB));
15     return EXIT_SUCCESS;
16 }

```

#### Solution 30 (Somme d'entiers).

```

1  #include <stdio.h>
2  #include <stdlib.h>
3
4  #define N 10
5
6  int main() {

```

```
7   int somme = 0;
8   int x;
9   for (int i = 0; i < N; i++) {
10      printf("Entrer l'entier %d : ", i + 1);
11      if (scanf("%d", &x) == 1) {
12         somme += x;
13      } else {
14         fprintf(stderr, "Il fallait entrer un entier !!!");
15         return EXIT_FAILURE;
16      }
17   }
18   printf("Somme : %d\n", somme);
19   return EXIT_SUCCESS;
20 }
```

### Tableaux statiques

Solution 31 (Affichage d'un tableau).

```
1 void afficher(int array[], int taille) {
2     for (int i = 0; i < taille; i++) {
3         printf("|%d|", array[i]);
4     }
5     printf("\n");
6 }
```

Solution 32 (Somme d'un tableau).

```
1 int somme(int array[], int taille) {
2     int sum = 0;
3     for (int i = 0; i < taille; i++) {
4         sum += array[i];
5     }
6     return sum;
7 }
```

Solution 33 (Maximum et minimum d'un tableau).

```
1 int max(int array[], int taille) {
2     int m = array[0];
3     for (int i = 1; i < taille; i++) {
4         m = array[i] > m ? array[i] : m;
5     }
6     return m;
7 }
8
9 int min(int array[], int taille) {
10    int m = array[0];
11    for (int i = 1; i < taille; i++) {
12        m = array[i] < m ? array[i] : m;
13    }
14 }
```



```
14     return m;
15 }
```

### Tableaux dynamiques

**Solution 34** (Afficher un tableau 2d [COURS]).

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  #define N 10
5  #define M 20
6
7  int main() {
8      int array[N][M];
9
10     for (int i = 0; i < N; i++) {
11         for (int j = 0; j < M; j++) {
12             printf("|%d|", array[i][j]);
13         }
14         printf("\n");
15     }
16
17     return EXIT_SUCCESS;
18 }
```

**Solution 35** (Affichage du triangle [COURS]).

Version généralisée pour N\_LINES quelconque :

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  #define N_LINES 5 // we assum it's always odd
5
6  int min(int a, int b) {
7      return (a > b) ? b : a;
8  }
9
10 void print_array(int** triangle, unsigned int n_lines) {
11     for (int i = 0; i < n_lines; i++) {
12         for (int j = 0; j < (1 + min(N_LINES - i - 1, i)); j++) {
13             printf("|%d|", triangle[i][j]);
14         }
15         printf("\n");
16     }
17 }
18
19 int main() {
20     int** triangle = (int**)(malloc(sizeof(int*) * N_LINES));
21     for (int i = 0; i < N_LINES; i++) {
22         triangle[i] = (int*)(malloc(sizeof(int) * (1 + min(N_LINES - i - 1, i))));
23     }
24 }
```

```

25     int cnt = 0;
26     for (int i = 0; i < N_LINES; i++) {
27         for (int j = 0; j < (1 + min(N_LINES - i - 1, i)); j++) {
28             triangle[i][j] = ++cnt;
29         }
30     }
31
32     print_array(triangle, N_LINES);
33     return EXIT_SUCCESS;
34 }

```

## Structures

### Solution 36 (Listes chaînées).

```

1  #include <stdio.h>
2  #include <stdlib.h>
3
4  struct Node {
5      struct Node* next;
6      struct Node* previous;
7      int value;
8  };
9
10 struct Node* node_new(int value) {
11     struct Node* nd = malloc(sizeof(struct Node));
12     nd->next = NULL;
13     nd->previous = NULL;
14     nd->value = value;
15     return nd;
16 }
17
18 struct LinkedList {
19     struct Node *head;
20     struct Node *queue;
21     unsigned int length;
22 };
23
24 struct LinkedList* linkedlist_new() {
25     struct LinkedList* lst = malloc(sizeof(struct LinkedList));
26     lst->length = 0;
27     lst->head = NULL;
28     lst->queue = NULL;
29     return lst;
30 }
31
32 char linkedlist_is_empty(struct LinkedList* l) {
33     return l->length == 0;
34 }
35
36 void linkedlist_push_on_head(struct LinkedList *l, int value) {
37     struct Node *nd = node_new(value);
38     if (linkedlist_is_empty(l)) {
39         l->head = l->queue = nd;

```

```

40     } else {
41         l->head->previous = nd;
42         nd->next = l->head;
43         l->head = nd;
44     }
45     l->length++;
46 }
47
48 void linkedlist_push_on_queue(struct LinkedList *l, int value) {
49     struct Node *nd = node_new(value);
50     if (linkedlist_is_empty(l)) {
51         l->head = l->queue = nd;
52     } else {
53         l->queue->next = nd;
54         nd->previous = l->queue;
55         l->queue = nd;
56     }
57     l->length++;
58 }
59
60 int linkedlist_pop_from_head(struct LinkedList *l) {
61     int to_return = -1;
62     if (!linkedlist_is_empty(l)) {
63         struct Node *to_delete = l->head;
64         l->head = to_delete->next;
65         if (l->head != NULL) { // if l->length == 1
66             l->head->previous = NULL;
67         }
68         to_return = to_delete->value;
69         free(to_delete);
70         l->length--;
71     } else {
72         fprintf(stderr, "Erreur : tentative de suppression d'un element d'une liste vide.\n");
73     }
74     return to_return;
75 }
76
77 int linkedlist_pop_from_queue(struct LinkedList *l) {
78     int to_return = -1;
79     if (!linkedlist_is_empty(l)) {
80         struct Node *to_delete = l->queue;
81         l->queue = to_delete->previous;
82         if (l->queue != NULL) { // if l->length == 1
83             l->queue->next = NULL;
84         }
85         to_return = to_delete->value;
86         free(to_delete);
87         l->length--;
88     } else {
89         fprintf(stderr, "Erreur : tentative de suppression d'un element d'une liste vide.\n");
90     }
91     return to_return;
92 }
93
94 void linkedlist_display(struct LinkedList *l) {
95     struct Node *tmp = l->head;

```

```

96     while (tmp != NULL) {
97         printf("->%d", tmp->value);
98         tmp = tmp->next;
99     }
100    printf("\n");
101 }
102
103 int main() {
104
105     struct LinkedList *l = linkedlist_new();
106
107     linkedlist_push_on_head(l, 5);
108     linkedlist_push_on_head(l, 4);
109     linkedlist_push_on_queue(l, 6);
110
111     linkedlist_display(l);
112
113     return EXIT_SUCCESS;
114 }

```

### Modulation et entêtes

#### Solution 37 (Un module de listes chaînées).

On écrit le code des fonctions de l'exercice précédent dans un fichier *"linkedlist.c"*<sup>5</sup> que l'on fait débiter par la ligne `#include "linkedlist.h"`. On écrit ensuite dans le même répertoire de travail le fichier *"linkedlist.h"* suivant, qui contient les prototypes et les structures :

```

1  #ifndef LINKEDLIST_H_INCLUDED
2  #define LINKEDLIST_H_INCLUDED
3
4  #include <stdlib.h>
5  #include <stdio.h>
6
7  struct Node {
8      struct Node* next;
9      struct Node* previous;
10     int value;
11 };
12
13 struct Node* node_new(int value);
14
15 struct LinkedList {
16     struct Node *head;
17     struct Node *queue;
18     unsigned int length;
19 };
20
21 struct LinkedList* linkedlist_new();
22 char linkedlist_is_empty(struct LinkedList* l);
23 void linkedlist_push_on_head(struct LinkedList *l, int value);
24 void linkedlist_push_on_queue(struct LinkedList *l, int value);
25 int linkedlist_pop_from_head(struct LinkedList *l);

```

5. Dans le répertoire de travail du fichier *"main.c"*

```

26 int linkedlist_pop_from_queue(struct LinkedList *l);
27 void linkedlist_display(struct LinkedList *l);
28
29 #endif

```

Il conserve la même fonction main du fichier “*main.c*” avec une inclusion du module :

```

1 // stdio is not needed here
2 #include <stdlib.h> // just for EXIT_SUCCESS
3
4 #include "linkedlist.h"
5
6 int main() {
7
8     struct LinkedList *l = linkedlist_new();
9
10    linkedlist_push_on_head(l, 5);
11    linkedlist_push_on_head(l, 4);
12    linkedlist_push_on_queue(l, 6);
13
14    linkedlist_display(l);
15
16    return EXIT_SUCCESS;
17 }

```

Et compilation puis exécution :

```

user@computer ~/working_directory> gcc main.c linkedlist.c -o main
user@computer ~/working_directory> ./main
->4->5->6
user@computer ~/working_directory>

```

### Chaînes de caractères

**Solution 38 (Traduction ASCII [COURS]).** La chaîne de caractères décrite est :

“Minitel = GOAT”

**Solution 39 (Calculatrice).**

```

1 #include <stdio.h>
2 #include <stdlib.h>
3
4 int main() {
5     char op;
6     double x;
7     double y;
8     while (1) {
9         printf("> ");
10        if (scanf("%lf %c %lf", &x, &op, &y) == 3) {
11            switch (op) {
12                case '+':
13                    printf("%lf\n", x + y);

```

```

14         break;
15     case '-':
16         printf("%lf\n", x - y);
17         break;
18     case '/':
19         printf("%lf\n", x / y);
20         break;
21     case '*':
22         printf("%lf\n", x * y);
23         break;
24     }
25     } else {
26         fprintf(stderr, "Erreur !\n");
27         return EXIT_FAILURE;
28     }
29 }
30
31 return EXIT_SUCCESS;
32 }

```

Interagir avec les flux de fichiers

## 8.2.2 Concepts avancés

Passage d'arguments au programme

Solution 40 (Liste des arguments).

```

1  #include <stdio.h>
2  #include <stdlib.h>
3
4  int main(int argc, char **argv) {
5      for (int i = 0; i < argc; i++) {
6          printf("%s\n", argv[i]);
7      }
8      return EXIT_SUCCESS;
9  }

```

Solution 41 (Un cat minimaliste).

```

1  #include <stdio.h>
2  #include <stdlib.h>
3
4  int main(int argc, char **argv) {
5      if (argc < 2) {
6          fprintf(stderr, "Il faut donner en argument du programme le nom du fichier à afficher.\n");
7      }
8      FILE *f = fopen(argv[1], "r");
9      if (f == NULL) {
10         fprintf(stderr, "Erreur d'ouverture du fichier %s\n", argv[1]);
11     }
12     char buffer[1024] = {0};

```

```
13  while (fread(buffer, 1, 1023, f) == 1023) {  
14      printf("%s", buffer);  
15  }  
16  printf("%s", buffer);  
17  return EXIT_SUCCESS;  
18 }
```

# BIBLIOGRAPHIE



- [1] A. V. AHO et al. *Compilers : Principles, Techniques, and Tools*. Pearson Education Inc., 2006. ISBN : 0201100886. URL : [https://drive.google.com/file/d/11t2AZMkYnSqhaufvIU1orxafr0mca2X/view?usp=drive\\_link](https://drive.google.com/file/d/11t2AZMkYnSqhaufvIU1orxafr0mca2X/view?usp=drive_link).
- [2] Olivier BOURNEZ. *Fondements de l'informatique : Logique, modèles et calculs*. École Polytechnique, juill. 2024. URL : [https://drive.google.com/file/d/11t2AZMkYnSqhaufvIU1orxafr0mca2X/%20view?usp=drive\\_link](https://drive.google.com/file/d/11t2AZMkYnSqhaufvIU1orxafr0mca2X/%20view?usp=drive_link).
- [3] Randal E. BRYANT et David R. O'HALLARON. *Computer Systems : A Programmer's Perspective*. Addison-Wesley Publishing Company, fév. 2010. ISBN : 9780136108047. URL : [https://drive.google.com/file/d/1bkLb30ByL\\_UaBNz-ksr03WliZ6mnuiy-/view?usp=drive\\_link](https://drive.google.com/file/d/1bkLb30ByL_UaBNz-ksr03WliZ6mnuiy-/view?usp=drive_link).
- [4] B. KERNIGHAN et D. RITCHIE. *The C programming language*. Prentice Hall, 1988. ISBN : 9780131101630. URL : [https://drive.google.com/file/d/1Yyl7VLCzZ\\_Y1la5hTbNi8qN0QaH9ntJZ/view?usp=drive\\_link](https://drive.google.com/file/d/1Yyl7VLCzZ_Y1la5hTbNi8qN0QaH9ntJZ/view?usp=drive_link).
- [5] M. KERRISK. *The Linux Programming Interface*. learning. 2010. ISBN : 9781593272203. URL : [https://drive.google.com/file/d/1CDfK3cz0xKj-E0bJOFTiKaDFwi2f3RgI/view?usp=drive\\_link](https://drive.google.com/file/d/1CDfK3cz0xKj-E0bJOFTiKaDFwi2f3RgI/view?usp=drive_link).
- [6] John R. LEVINE. *Linkers and Loaders*. Morgan-Kaufman, oct. 1999. ISBN : 1558604960. URL : [https://drive.google.com/file/d/1EtAHWMLjpFzL8Y6jlJ5H32-1L2AJ\\_eF9/view?usp=drive\\_link](https://drive.google.com/file/d/1EtAHWMLjpFzL8Y6jlJ5H32-1L2AJ_eF9/view?usp=drive_link).
- [7] Jean-Michel MULLER et al. *Handbook of Floating-Point Arithmetic, 2nd edition*. ACM G.1.0; G.1.2; G.4; B.2.0; B.2.4; F.2.1., ISBN 978-3-319-76525-9. Birkhäuser Boston, 2018, p. 632. URL : [https://drive.google.com/file/d/191hTn7YzczQ\\_B7ftjVLfgRbONEZBcnbM/view?usp=drive\\_link](https://drive.google.com/file/d/191hTn7YzczQ_B7ftjVLfgRbONEZBcnbM/view?usp=drive_link).



- [8] D. SALOMON. *Assemblers and Loaders*. Ellis Horwood Ltd, fév. 1993. ISBN : 0130525642. URL : [https://drive.google.com/file/d/1IAtrVoKw0khZLkjWorFNVQpquWi5zTHR/view?usp=drive\\_link](https://drive.google.com/file/d/1IAtrVoKw0khZLkjWorFNVQpquWi5zTHR/view?usp=drive_link).



## MANUELS ET DOCUMENTATION



- [9] *C11 Specification*. URL : [https://drive.google.com/file/d/1nUx1JETBBm7zyQR0oviL02veJtb5MpLD/view?usp=drive\\_link](https://drive.google.com/file/d/1nUx1JETBBm7zyQR0oviL02veJtb5MpLD/view?usp=drive_link).
- [10] *GNU Make*. URL : [https://drive.google.com/file/d/1admoAxMKteDXBY7uSiRmE60mgag51ia\\_/view?usp=drive\\_link](https://drive.google.com/file/d/1admoAxMKteDXBY7uSiRmE60mgag51ia_/view?usp=drive_link).



## AUTRES LIENS



- [11] Sean Eron ANDERSON. *Bit Twiddling Hacks*. Mai 2005. URL : <http://graphics.stanford.edu/~seander/bithacks.html>.