

22 août 2025

B. Garnier

# Algo/Prog

Version loin d'être finie



TODO (pour indiquer ce qui n'est pas encore fait histoire de pas (trop) décevoir) :

- pas fini la section 1.2 sur l'induction structurelle (mais assez pour comprendre la partie 3)
- pas encore d'exercices pour le chapitre 1
- pas fini le chapitre 3 sur l'architecture des ordinateurs.
- le chapitre 7 n'a que très peu d'exercices
- raisonnements sur les algorithmes (ce qui peut être frustrant puisque certains exercices peuvent nécessiter de savoir faire)
- il manque les solutions des exercices de la partie 3 (sauf **les exercices sur les listes qui sont entièrement corrigés**)
- tables de hachage : load factor et analyse de complexité
- algorithmes sur graphes valués : dijkstra, johnson (et aucune illustration pour faire passer la pilule formelle)
- algo sur les flots
- détail des opérations de tamisage de tas
- Approches algorithmiques
- Partie 4 : GDB, Git, Programmation propre, optimisation des constantes (en approfondissement du chapitre 3 d'un point de vue de programmeur)

À réfléchir pour mettre en annexe toutes les démonstrations longues dont seuls importent les résultats. Dépendra des avis. Pour l'instant, les démos sont à la suite des propositions ou en exercice.



# PRÉFACE



Adresse de contact valable *a minima* jusqu'à la date sur la couverture : [bastiengarnier17@gmail.com](mailto:bastiengarnier17@gmail.com)

La qualité stylistique comme pédagogie varie d'une section à une autre, que ce soit dû à la fatigue, au fait que certaines sections écrites d'abord souffrent d'un manque d'expérience dans la rédaction et que d'autres sections écrites en dernières souffrent de manquement dans la relecture et l'affinage.

Ainsi, il est chaudement demandé que toutes coquilles relevées<sup>1</sup>, toutes suggestions d'amélioration ou toutes autres remarques quelconques soient envoyées soit à la section "Formation" du serveur Discord de l'association Minitel de l'ISMIN ou à l'adresse de contact donnée ci-dessus, pour les prochaines versions.

**À propos de la licence :** Une association n'a pas but à être lucrative. La licence CC-BY-NC-SA spécifie que :

- L'oeuvre peut être reproduite, distribuée et modifiée en accord avec les points ci-dessous.
- Le nom de chaque auteur de ce document doit être cité (mise-à-jour à chaque nouvelle version)
- Ce document ne peut pas être édité ou modifié sous une autre licence<sup>2</sup>.
- Tout usage commercial de ce document est prohibé.

---

1. 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 des erreurs techniques ;)

2. Il existe une copie de l'auteur originel qui atteste de la licence d'origine. *Just to say...*

## Section à sauter sans lire

Merci à Basile Tonlorenzi pour la relecture, les idées et tout et tout! ^^ Content d'avoir un cours relativement complet, après les quelques 10 mois de gestation de ce projet Minitel.

**À propos du neutre :** il n'y a pas de neutre en français malheureusement. Je considère que les genres quand ils désignent un groupe d'individus quelconque ou un individu indéterminé, non identifié comme personne, sont sémantiquement nuls et non avenants et donc purs produits du hasard grammatical (exemple : *table* et *tabouret* respectivement “féminin” et “masculin”). On pourrait tout aussi bien nommer le masculin et le féminin grammaticaux les formes grammaticales *A* et *B* sans que cela ne change rien (c'est-à-dire qu'une table serait à la forme *B* et un tabouret à la forme *A*, ou inversement selon la convention choisie...).

J'utiliserais le masculin grammaticale dans la suite comme forme neutre, par habitude puisqu'il s'agit du standard grammatical pour le neutre depuis quelques centaines d'années. Ainsi, la phrase “Bonjour à tous.” porte la signification : “Bonjour à tous les individus présents.”. Le masculin est ici purement grammaticale et est conservé pour traiter les entités pensées neutres dans le texte. L'idée est aussi d'alléger l'écriture. Si l'utilisation systématique du masculin grammaticale comme neutre pour désigner des entités supposées implicitement humaines dérange, prière d'envoyer un mail avec argumentation. Je réécrirais en conséquence les parties concernées en précisant les deux formes.

Enfin bref... j'ai prévenu qu'il fallait sauter cette section.

## Motivation

Ce livre 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 et complet* 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.

Certains demanderont – en toute légitimité – pourquoi écrire un nouveau livre de programmation quand tant d'autres existent déjà. À cela plusieurs raisons :

1. Ce cours assure la localité des informations. Le programme d'informatique de l'ISMIN concerne plusieurs domaines particuliers de l'informatique qui peuvent être très différents, bien que liés. Pour se renseigner plus avant sur les sujets abordés en cours, ou trouver des ressources de révision, il faut alors parcourir de nombreux ouvrages différents. D'abord un étudiant n'a pas nécessairement connaissance des références littéraires sur le sujet, ce qui peut l'amener à devoir itérer ses recherches un certain nombre de fois avant d'arriver sur une source d'information claire et précise qui couvre exactement l'objet de ses désirs. Deuxièmement, ces ouvrages couvrent leur thème souvent de manière très approfondie et un étudiant n'est pas toujours intéressé par l'intégralité du contenu. Ce livre de Minitel veut couvrir les thèmes du cursus en introduisant juste assez de notions pour garantir la rigueur sans pour autant (trop) déborder en hors-sujet vis-à-vis du programme.
2. Exiger la connaissance de la langue anglaise pour apprendre l'informatique ou la programmation constitue un frein sévère pour une grande quantité d'individus. Au lieu de dépenser son énergie à la compréhension, l'assimilation et la maîtrise de connaissances parfois ardûes, on se fatigue et trébuche sur la langue dans laquelle sont exprimées ces connaissances. Il est évident que la maîtrise de l'anglais est nécessaire dans l'industrie informatique, mais tout un chacun devrait pouvoir apprendre si il le veut sans devoir passer quelques années avant à travailler une langue étrangère qui ne constitue que la forme sous laquelle est exprimé le savoir.

3. Peu de livres abordent avec précision la programmation en C en considérant que l'étudiant lecteur a un niveau de sortie de classes préparatoires. En effet, dans un premier cas les livres vont considérer une absence totale de connaissances même en mathématique, ce qui amène inévitablement à un manque de rigueur dérangeant, car la programmation est fondamentalement liée à des concepts mathématiques qui sont parfois esquivés pour des raisons de facilité. Dans un deuxième cas ces livres peuvent présupposer un savoir préalable de quantité de concepts informatiques dont de nombreux étudiants de classes préparatoires n'ont pas connaissances. Par ailleurs, de nombreux livres qui apportent une réflexion théorique, plus orienté informatique que programmation, partent du postulat que le lecteur est déjà à l'aise avec la programmation. L'idée de ce livre est donc d'apporter un alliage de connaissances théoriques et pratiques orientées vers la programmation (impérative) qui ne réintroduise pas des mathématiques de classes préparatoires, tout en conservant une certaine rigueur.
4. Ce livre est éditable par l'association Minitel et peut donc être étendu aux ajouts de notions résultant de réformes du programme de l'ISMIN. De plus, si certaines explications ne sont pas claires ou qu'un point nécessite des exemples et explications supplémentaires, ce peut être ajouté. La mise en page aussi peut être retravaillée. De manière générale, la possibilité pour les étudiants d'éditer le cours selon leurs besoins le rend particulièrement flexible. Le défaut est évidemment qu'il n'est pas certain que tout contenu ajouté soit parfaitement valide et sans fautes. C'est pour cela qu'il faut s'attacher à la plus grande rigueur, par référencement bibliographique des notions, démonstration des propositions et de manière générale, argumentation de toute assertion. La présente édition n'est pas parfaite. Il est espéré qu'elle soit complétée/améliorée avec le temps.

Toutefois, la légitimité d'un tel cours peut se poser en école d'ingénieurs pour d'autres raisons. Ainsi, les études de la primaire à la classe préparatoire ont habitué les étudiants à ingurgiter de grandes quantités de connaissances pour ainsi dire "à la petite cuillère". Or, un ingénieur, en particulier dans le domaine de l'informatique où les technologies évoluent très rapidement, se doit d'être capable d'apprendre par "soi-même", c'est-à-dire être capable d'aller chercher soi-même les ressources, bibliographiques ou webographiques, nécessaires à son apprentissage. Beaucoup d'enseignants en école d'ingénieurs utilisent ce fait comme argument pour bâcler leur cours<sup>3</sup>. *A contrario*, ce cours a été pensé pour pouvoir être utilisé à la fois comme *catalogue* sur la programmation en langage C, qui décrit rigoureusement les différents points du langage, mais aussi comme une introduction à des connaissances couvrant un spectre plus large de l'informatique et de la programmation. L'idée est de permettre à terme l'autonomie du lecteur pour approfondir de manière technique des sujets spécialisés en fournissant les outils de base nécessaire à leur compréhension.

Ainsi, une bibliographie est disponible en **Annexe** qui propose de manière thématique certains ouvrages spécialisés de référence<sup>4</sup>. Cette bibliographie couvre (entre autres) les thèmes suivants :

- Représentation binaire des nombres [16][2]
- Programmation de manière générale [19]
- Langage C [12][20][7]
- Programmation système sous Linux [13]
- Compilation [10][15][18][1]
- Programmation bas-niveau [6]
- Algorithmique [4][14]
- Informatique fondamentale [17][5]

L'objectif est d'apporter une préparation pour les examens de programmation en C de l'ISMIN, nommément *Algo/Prog 1* et 2. Les chapitres 4, 5 et 5 sont suffisants pour *Algo/Prog 1*. La lecture du

---

3. *Solutions de communications sans fil dans le domaine de l'IOT* en est l'exemple le plus absolu, mais certains professeurs d'autres cours, sans le dire ouvertement, justifient l'incomplétude de leurs cours par le manque de temps pour le donner, sans penser un instant à passer la quatrième.

4. Je n'ai pas encore tout lu en entier... 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 ;(

chapitre 9 est suffisante pour *Algo/Prog 2*. La lecture du chapitre 10 est particulièrement recommandée. Le reste sert à la rigueur, à l'exhaustivité et se veut un complément technique d'intérêt.

La programmation en C/C++ et la maîtrise de domaines de l'informatique sont aussi exigées pour d'autres cours :

- *Programmation Système* en 1A
- *Architecture des processeurs (1 et 2)* en 1A et 2A
- *Systèmes à microcontrôleurs (1 et 2)* en 1A et 2A
- *Programmation Orientée Objets* en 2A
- *Cryptographie* en 2A
- *Optimisation combinatoire* en 2A
- *Intelligence Artificielle* en 2A
- *Sécurité des réseaux informatiques* en 2A

Ce livre ne couvre pas *encore* les points théoriques et techniques abordés durant ces cours. À voir... (priorité pour Architecture des processeurs, y a pas de support de cours en 2A, et SAM le cours est ...)

---

## Difficulté des exercices

Un certain nombre d'exercices sont présents en fin de section, ces sections étant séparés par thème. Ces exercices ont évidemment comme premier objectif de fournir au lecteur des occasions de pratiquer. Un second est de montrer des cas d'application pratique des notions vues dans un chapitre et parfois découvrir de nouvelles notions intéressantes en lien. Ces exercices ont été choisis dans le but d'être intéressants<sup>5</sup>, avec l'objectif ultime qu'en abordant un exercice, le lecteur se dise "Ah ? Tiens donc, rigolo."<sup>6</sup>.

On utilise l'échelle de mesure de difficulté utilisé par Donald Knuth<sup>7</sup>[14] :

### *Interprétation*

- 00 Un exercice extrêmement facile qui peut être résolu immédiatement, de tête, si le cours est compris
- 10 Un problème très simple pour travailler la compréhension du texte. Peut prendre au maximum une minute ou deux et un stylo (ah ! faut du papier aussi !)
- 20 Un problème moyen pour tester sa compréhension du texte. Peut prendre quinze à vingt minutes pour être résolu complètement
- 30 Un problème plus difficile, un peu complexe, qui peut prendre plusieurs heures pour être résolu de manière satisfaisante
- 40 Problème vraiment difficile ou long. Un étudiant doit pouvoir être capable de résoudre le problème en un temps "raisonnable", mais la solution n'est pas trivial
- 50 Un problème de recherche non encore résolu de manière satisfaisante, bien que beaucoup aient essayés.

**Remarque :** La difficulté est interpolée « logarithmiquement », c'est-à-dire qu'un exercice de difficulté 17 est un peu plus simple qu'un exercice de difficulté 20, et passablement plus difficile qu'un exercice de difficulté 10.

---

5. Pas toujours réussi hein !

6. Encore moins réussi...

7. Le G.O.A.T !

**Sigles spécifiques :** On ajoutera certains symboles à côté de la difficulté pour indiquer des détails sur le type d'exercice :

- M : l'exercice implique plus de concepts mathématiques qu'il n'est nécessaire pour un lecteur intéressé uniquement par la programmation
- HM : l'exercice implique l'utilisation d'outils mathématiques assez développés qui ne sont pas détaillés dans ce livre
- • : l'exercice est particulièrement instructif et utile

**Remarque :** un exercice noté *HM* n'induit pas *nécessairement* une difficulté supplémentaire extraordinaire.

Ce livre n'a pas comme objectif d'amener à exposer des théories complexes, et l'auteur<sup>8</sup> n'est pas un ressortissant de l'ENS Ulm, donc il n'y aura pas de problèmes côtés à 45 ou plus.

Par ailleurs, la difficulté d'un exercice est très subjective, puisque ce qui semble simple et intuitif pour l'un peut être un enfer pour l'autre. La difficulté est donc assez approximative, histoire de donner une idée, et que le lecteur ne parte pas bille en tête dans un exercice particulièrement difficile sans se poser de questions ou se complique la vie inutilement sur un exercice *plutôt* simple<sup>9</sup>.

Pour ceux qui voudraient pratiquer plus la programmation<sup>10</sup>, la banque d'exercices de *Leetcode* est assez fournie (<https://leetcode.com/problemset/>). Elle nécessite pour beaucoup d'exercices les notions présentées dans les trois premières parties.

---

8. Les auteurs un jour ?

9. Je n'aimerais pas me chiffonner ou vexer quiconque qui pourrait galérer sur un exercice indiqué "simple"... Ça peut aussi être une erreur d'estimation, OU PIRE nue ftaue ed fappre (genre 13 au lieu de 31), faut me prévenir si tel est le cas :)

10. Pour s'entraîner pour les examens de l'ISMIN par exemple

# Table des matières

Préface . . . . .	1
Paragraphe à sauter sans lire . . . . .	1
Motivation . . . . .	2
Difficulté des exercices . . . . .	4
 <b>I Prolégomènes</b>	 <b>15</b>
Introduction de l'introduction . . . . .	16
 <b>1 Rappels mathématiques</b>	 <b>17</b>
1.1 Relations binaires et ordre . . . . .	17
1.1.1 Relation . . . . .	17
1.1.2 Relations d'équivalence . . . . .	19
1.1.3 Clôture . . . . .	20
1.1.4 Relations d'ordre . . . . .	21
1.2 Induction structurelle . . . . .	22
1.2.1 Principe sur un exemple . . . . .	22
1.2.2 Formalisation de l'induction . . . . .	24
1.2.3 Formalisation du raisonnement par induction . . . . .	25
1.2.4 Caractérisation inductive d'une fonction . . . . .	25
1.2.5 Exercices . . . . .	25
 <b>2 Les données</b>	 <b>26</b>
2.1 Binaire . . . . .	26
2.1.1 Octets . . . . .	26
2.2 Opérations logiques . . . . .	27
2.2.1 Arité d'un opérateur, d'une fonction . . . . .	27
2.2.2 Opérations logiques et tables de vérité . . . . .	28
2.2.3 Opérations logiques élémentaires . . . . .	28
2.2.4 Opérations de décalage . . . . .	30
2.2.5 Exercices . . . . .	31
2.3 Opérations arithmétiques . . . . .	31
2.3.1 Représentation d'un nombre entier en binaire . . . . .	32
2.3.2 Nombres signés . . . . .	33
2.3.3 Addition . . . . .	33
2.3.4 Soustraction . . . . .	34
2.3.5 Décalage arithmétique à droite . . . . .	34
2.3.6 Exercices . . . . .	35
2.4 Opérations flottantes . . . . .	35
2.4.1 Représentation des nombres à virgules . . . . .	35
2.4.2 Écriture en base 2 de nombres décimaux . . . . .	36
2.4.3 Norme <i>IEEE 754</i> . . . . .	37

2.4.4	Notation pour la suite . . . . .	39
2.4.5	Approximation du logarithme par la norme <i>IEEE 754</i> . . . . .	39
2.4.6	Exercices . . . . .	42
2.5	Représentation hexadécimale . . . . .	42
2.6	Données textuelles . . . . .	43
2.6.1	Caractères ASCII . . . . .	44
2.6.2	Caractères imprimables et non imprimables . . . . .	44
<b>3</b>	<b>L'ordinateur</b> . . . . .	<b>46</b>
3.1	Architecture matérielle . . . . .	46
3.1.1	Vue de haut . . . . .	46
3.1.2	À l'intérieur . . . . .	47
3.1.3	Limitation du modèle de Von Neumann . . . . .	48
3.2	Architecture logicielle . . . . .	49
<b>4</b>	<b>Premières notions de programmation</b> . . . . .	<b>53</b>
4.1	Algorithmes et programmation . . . . .	53
4.1.1	Algorithmes et programmes . . . . .	55
4.2	Langages de programmation . . . . .	56
4.2.1	Objectifs des langages de programmation . . . . .	56
4.2.2	Langages compilés et interprétés . . . . .	57
4.3	Le langage C . . . . .	58
4.3.1	Installer un éditeur et un compilateur . . . . .	58
4.3.2	Compiler le premier programme . . . . .	60
4.3.3	Analyse du premier programme . . . . .	61
<b>II</b>	<b>Du langage C</b> . . . . .	<b>64</b>
	Introduction . . . . .	65
<b>5</b>	<b>Fondamentaux du langage</b> . . . . .	<b>67</b>
5.1	Identifiants . . . . .	67
5.2	Commentaires . . . . .	69
5.2.1	Commentaires sur une ligne . . . . .	70
5.2.2	Commentaires par bloc . . . . .	70
5.3	Le point-virgule . . . . .	71
5.4	Point d'entrée du programme . . . . .	72
5.5	Directives du préprocesseur (1) . . . . .	75
<b>6</b>	<b>Bases du langage</b> . . . . .	<b>76</b>
6.1	Variables . . . . .	76
6.1.1	Taille et type d'une variable . . . . .	80
6.1.2	Entiers signés et non signés . . . . .	81
6.1.3	Environnement et blocs . . . . .	81
6.1.4	Exercices . . . . .	83
6.2	Formatage de texte . . . . .	84
6.2.1	Formatage des caractères spéciaux . . . . .	85
6.2.2	Formatage de variables . . . . .	85
6.2.3	Exercices . . . . .	86
6.3	Opérateurs sur les variables . . . . .	86
6.3.1	Opérateurs arithmétiques . . . . .	88
6.3.2	Opérateurs bit-à-bit . . . . .	89



6.3.3	Opérateurs logiques, ou relationnels	90
6.3.4	Opérateurs d'assignation	91
6.3.5	Exercices	93
6.4	Projection de type	94
6.4.1	Exercices	95
6.5	Structures de contrôle	96
6.5.1	Structures élémentaires	97
6.5.2	Structures complexes	99
6.5.3	Détails sur l'instruction <b>break</b> (et l'instruction <b>continue</b> )	105
6.5.4	Exercices	106
6.6	Routines	107
6.6.1	Signature et prototype	108
6.6.2	Note relative à la copie des arguments	109
6.6.3	La pile d'exécution	111
6.6.4	Effet de bord	113
6.6.5	Exercices	114
6.7	Pointeurs	114
6.7.1	Pointeurs et références	115
6.7.2	Détails sur les références : <i>lvalue</i> et <i>rvalue</i>	118
6.7.3	Les pointeurs en paramètres de routines	119
6.7.4	Arithmétique des pointeurs et projection de type	120
6.7.5	Pointeurs itérés	123
6.7.6	Exercices	123
6.8	Interagir avec les flux standards	124
6.8.1	Flux d'entrée standard	125
6.8.2	Flux de sortie standard et flux d'erreur standard	126
6.8.3	Exemple pratique d'utilisation de <i>stderr</i>	127
6.8.4	Exercices	127
6.9	Tableaux statiques	129
6.9.1	Tableaux	129
6.9.2	Définition	130
6.9.3	Les tableaux statiques, kékoï pour de vrai?	132
6.9.4	Exercices	135
6.10	Allocation dynamique	137
6.10.1	Introduction au tas	137
6.10.2	Les routines <b>malloc</b> et <b>free</b>	138
6.10.3	L'importance de réinitialiser le pointeur après libération	139
6.10.4	Exercices	141
6.11	Tableaux multidimensionnels	141
6.11.1	Tableaux multidimensionnels statiques	142
6.11.2	Tableaux multidimensionnels dynamiques	143
6.11.3	Exercices	144
6.12	Structures	144
6.12.1	Initialisation à la déclaration	149
6.12.2	Mécanisme de construction	150
6.12.3	Tableaux de structures	151
6.12.4	Alignement et optimisation des structures	153
6.12.5	Exercices	154
6.13	Modulation et entêtes	156
6.13.1	Fichier d'entête	157
6.13.2	Fichier de code source	157
6.13.3	Un exemple simple	158

6.13.4	Compilation modulaire . . . . .	160
6.13.5	Problème des chaînes d'inclusions . . . . .	163
6.13.6	Garde-fous . . . . .	164
6.13.7	Exercices . . . . .	166
6.14	Chaînes de caractères . . . . .	166
6.14.1	Chaîne de caractères . . . . .	166
6.14.2	En langage C . . . . .	166
6.14.3	Se faciliter la vie avec les chaînes littérales . . . . .	167
6.14.4	Exercices . . . . .	168
6.15	Les flux de fichiers . . . . .	169
6.15.1	Droits des fichiers . . . . .	169
6.15.2	Les flux de fichiers en C . . . . .	171
6.15.3	Rediriger un flux standard vers un fichier . . . . .	173
6.15.4	Exercices . . . . .	174
<b>7</b>	<b>Concepts avancés</b>	<b>175</b>
7.1	Virgule et expressions . . . . .	175
7.1.1	La virgule comme opérateur . . . . .	175
7.1.2	La virgule comme séparateur . . . . .	177
7.1.3	Conclusion . . . . .	177
7.2	Paramètres d'un programme . . . . .	178
7.2.1	Le (véritable?) prototype de <i>main</i> . . . . .	178
7.2.2	Exercices . . . . .	179
7.3	Énumérations . . . . .	179
7.3.1	Motivation . . . . .	179
7.3.2	Syntaxe . . . . .	180
7.4	Unions . . . . .	181
7.4.1	Motivation par un exemple . . . . .	181
7.4.2	Principe et syntaxe . . . . .	181
7.4.3	Le problème de sélection . . . . .	182
7.5	Champs de bits . . . . .	184
7.5.1	Motivation et principe . . . . .	184
7.5.2	Syntaxe . . . . .	185
7.5.3	Le revers de la médaille . . . . .	186
7.6	Classes de stockage . . . . .	187
7.6.1	Quelques détails de la structure d'un programme en mémoire . . . . .	188
7.6.2	Choisir la classe de stockage d'une variable en C . . . . .	189
7.7	const et restrict . . . . .	192
7.7.1	const . . . . .	192
7.7.2	restrict . . . . .	196
7.8	Construction de littéraux . . . . .	197
7.8.1	Motivation . . . . .	197
7.8.2	Syntaxe . . . . .	198
7.8.3	Cas d'utilisation . . . . .	198
7.9	Pointeurs de routines . . . . .	199
7.9.1	Motivation . . . . .	199
7.9.2	Syntaxe . . . . .	200
7.9.3	Tableaux de routines . . . . .	202
7.9.4	Exercices . . . . .	202
7.10	Directives du préprocesseur (2) . . . . .	202
7.10.1	Inclusion de fichiers externes . . . . .	203
7.10.2	Compilation conditionnelle . . . . .	204

7.10.3	Diagnostiques . . . . .	205
7.10.4	Pragma . . . . .	206
7.10.5	Macros . . . . .	206
7.10.6	Opérateurs de macros sur les symboles . . . . .	208
7.10.7	Exercices . . . . .	210
7.11	Routines variadiques . . . . .	211
7.11.1	Motivation . . . . .	211
7.11.2	Appels de routines . . . . .	211
7.11.3	Syntaxe et module <i>stdarg</i> . . . . .	212
7.11.4	Macros variadiques . . . . .	213
7.11.5	Exercices . . . . .	213
7.12	Assembleur et C . . . . .	214
7.13	Tricks récréatifs (ft. Basile) . . . . .	214
7.13.1	Indexation inversée et boutisme . . . . .	215
7.13.2	Définition <i>K&amp;R</i> de fonctions . . . . .	215
7.13.3	Digraphes et trigraphes . . . . .	215
7.13.4	Un peu d'imagination . . . . .	216
<b>III</b>	<b>Petite parenthèse théorique</b>	<b>219</b>
	Introduction . . . . .	220
<b>8</b>	<b>Complexité</b>	<b>222</b>
8.1	Calcul . . . . .	222
8.1.1	Modèles de calcul . . . . .	222
8.2	Mesures du coût . . . . .	224
8.2.1	Notations de Landau . . . . .	224
8.2.2	Analyse de cas pour la complexité . . . . .	226
<b>9</b>	<b>Structures de données</b>	<b>229</b>
9.1	Retour sur les types abstraits . . . . .	229
9.1.1	Spécification d'un type abstrait sur un exemple . . . . .	230
9.1.2	Structure de données . . . . .	232
9.1.3	Exercices . . . . .	233
9.2	Listes . . . . .	234
9.2.1	Piles . . . . .	235
9.2.2	Files . . . . .	237
9.2.3	Implantation par chaînage . . . . .	237
9.2.4	Exercices . . . . .	240
9.3	Arbres binaires . . . . .	241
9.3.1	Motivation . . . . .	241
9.3.2	Arbres quelconques . . . . .	241
9.3.3	Principe . . . . .	242
9.3.4	Signature et implantation . . . . .	243
9.3.5	Parcours d'arbres . . . . .	245
9.3.6	Arbres binaires de recherche . . . . .	247
9.3.7	Arbres équilibrés . . . . .	251
9.3.8	Arbres AVL . . . . .	254
9.3.9	Exercices . . . . .	254
9.4	Files de priorité . . . . .	256
9.4.1	Signature . . . . .	256
9.4.2	Arbres complets . . . . .	257

9.4.3	Tas minimum . . . . .	258
9.4.4	Mutateurs . . . . .	259
9.4.5	Exercices . . . . .	260
9.5	Dictionnaires . . . . .	262
9.5.1	Introduction . . . . .	262
9.5.2	Tables et fonctions de hachage . . . . .	263
9.5.3	Fonction de hachage . . . . .	266
9.5.4	Réduction . . . . .	267
9.5.5	Gestion des collisions . . . . .	268
9.5.6	Exercices . . . . .	271
9.6	Graphes (1) . . . . .	272
9.6.1	Définition et représentations . . . . .	272
9.6.2	Signature du type abstrait . . . . .	274
9.6.3	Représentations . . . . .	275
9.6.4	Propriétés . . . . .	276
9.6.5	Parcours en profondeur et en largeur . . . . .	277
9.6.6	Algorithme de Roy-Warshall . . . . .	282
9.6.7	Exercices . . . . .	284
9.7	Partitions d'ensembles finis . . . . .	287
9.7.1	Signature . . . . .	287
9.7.2	Principe sur un exemple naïf . . . . .	287
9.7.3	La clef vers l'optimisation . . . . .	288
9.7.4	Implantation . . . . .	289
9.7.5	Amélioration de la complexité dans le pire cas . . . . .	291
9.7.6	Exercices . . . . .	293
9.8	Graphes (2) : valuation . . . . .	294
9.8.1	Définition et représentations . . . . .	294
9.8.2	Signature du type abstrait . . . . .	295
9.8.3	Représentations . . . . .	295
9.8.4	Premières propriétés . . . . .	295
9.8.5	Algorithmes de Kruskal et Prim . . . . .	296
9.8.6	Bellman-Ford . . . . .	299
9.8.7	Algorithme de Dijkstra . . . . .	302
9.8.8	Algorithme de Johnson . . . . .	302
9.8.9	Exercices . . . . .	304
9.9	Graphes (3) : flots . . . . .	305
<b>IV</b>	<b>Le vrai monde de la réalité réelle</b>	<b>306</b>
	Introduction . . . . .	307
<b>10</b>	<b>Outils pour la programmation</b>	<b>308</b>
10.1	Makefiles . . . . .	308
10.1.1	Généralités . . . . .	308
10.1.2	Principe et premiers exemples . . . . .	308
10.1.3	Personnaliser la production grâce aux variables . . . . .	310
10.1.4	Variables automatiques . . . . .	311
10.1.5	Réduire le nombre de règles avec la <i>stem</i> . . . . .	311
10.1.6	Les caractères génériques . . . . .	312
10.1.7	Substitution de chaînes . . . . .	313
10.1.8	Les règles virtuelles . . . . .	313
10.1.9	Idées pour aller plus loin . . . . .	314

10.2	Utiliser un débogueur . . . . .	314
10.3	Git . . . . .	315
10.4	Markdown . . . . .	315
<b>11</b>	<b>Programmer proprement</b>	<b>317</b>
11.1	À propos du style . . . . .	317
11.2	La gestion des erreurs . . . . .	317
11.2.1	Un petit exemple pour prendre la température . . . . .	318
11.2.2	Codes d'erreur . . . . .	319
11.3	Tests unitaires . . . . .	320
11.4	Benchmarks . . . . .	320
<b>V</b>	<b>Annexes</b>	<b>321</b>
<b>12</b>	<b>Correction des 95 exercices</b>	<b>322</b>
12.1	Première partie . . . . .	322
12.1.1	Opérations logiques sur les mots binaires . . . . .	322
12.1.2	Opérations arithmétiques . . . . .	325
12.1.3	Opérations sur les nombres à virgules . . . . .	326
12.1.4	Représentation hexadécimale . . . . .	327
12.1.5	Caractères ASCII . . . . .	327
12.2	Deuxième partie . . . . .	327
12.2.1	Bases du langage . . . . .	327
12.2.2	Concepts avancés . . . . .	342
12.3	Troisième partie . . . . .	343
12.3.1	Types abstraits . . . . .	343
12.3.2	Listes . . . . .	343
	<b>Bibliographie</b>	<b>349</b>

# Liste des définitions

1.1.1	Relation . . . . .	18
1.1.2	Domaine et image d'une relation . . . . .	18
1.1.3	Propriétés standards des relations homogènes . . . . .	19
1.1.4	Relation d'équivalence . . . . .	19
1.1.5	Classes d'équivalences et quotient . . . . .	19
1.1.6	Clôture . . . . .	20
1.1.7	Relation d'ordre . . . . .	21
1.1.8	Bornes par un ordre . . . . .	21
1.1.9	Ensembles bien ordonnés . . . . .	21
1.2.10	Constructeurs et termes . . . . .	24
3.1.11	Instruction . . . . .	48
3.2.12	BIOS . . . . .	50
3.2.13	Système d'exploitation . . . . .	50

3.2.14	Fichier . . . . .	51
3.2.15	Extension de nom fichier . . . . .	51
3.2.16	Répertoire et système de fichiers . . . . .	51
3.2.17	Répertoire de travail . . . . .	52
3.2.18	Chemin d'accès . . . . .	52
4.1.19	Problème algorithmique (intuitivement) . . . . .	54
4.1.20	Algorithme (intuitivement) . . . . .	54
4.1.21	Agent algorithmique . . . . .	55
4.1.22	Programme . . . . .	55
4.2.23	Niveau d'abstraction . . . . .	57
5.1.24	Identifiant . . . . .	67
5.4.25	Exécutable . . . . .	72
6.1.26	Type . . . . .	77
6.1.27	Environnement global . . . . .	81
6.1.28	Environnement local . . . . .	81
6.3.29	Opérateurs et opérandes . . . . .	86
6.3.30	Type booléen et valeur de vérité d'une expression . . . . .	90
6.5.31	Flot d'exécution/de contrôle . . . . .	96
6.6.32	Signature . . . . .	108
6.6.33	Prototype . . . . .	109
6.6.34	Argument . . . . .	110
6.8.35	Flux . . . . .	124
6.9.36	Structure de donnée . . . . .	129
6.12.37	Alignement . . . . .	153
7.6.38	Portée d'une variable et durée de vie . . . . .	188
7.7.39	Aliasing . . . . .	196
7.9.40	Fonction d'ordre supérieur . . . . .	200
8.1.41	Étoile de Kleene usuelle . . . . .	222
8.1.42	Modèle de calcul . . . . .	222
8.1.43	Problème . . . . .	223
8.1.44	Problème de décision . . . . .	223
8.1.45	Algorithme . . . . .	223
8.2.46	Modèle de coût . . . . .	224
8.2.47	Fonction de coût . . . . .	224
8.2.48	Domination linéaire . . . . .	225
8.2.49	Notation de Landau . . . . .	226
8.2.50	Cas . . . . .	226
8.2.51	Pire cas . . . . .	227
8.2.52	Meilleur cas . . . . .	227
8.2.53	Analyse de complexité du pire cas . . . . .	227
8.2.54	Analyse de complexité du meilleur cas . . . . .	227
9.1.55	Structure de données . . . . .	232
9.3.56	Arbres binaires . . . . .	243
9.3.57	Hauteur d'un arbre . . . . .	243
9.3.58	Arbre binaire de recherche . . . . .	247
9.3.59	Équilibre d'un arbre . . . . .	251
9.3.60	Rotations gauche et droite . . . . .	252
9.3.61	Arbre AVL . . . . .	254
9.4.62	Arbres presque complets . . . . .	257
9.4.63	Arbre tournoi . . . . .	258
9.4.64	Tas . . . . .	259
9.5.65	Clefs et valeurs . . . . .	262

9.5.66	Taux d'occupation . . . . .	268
9.6.67	Graphe orienté . . . . .	273
9.6.68	Graphe non orienté . . . . .	273
9.6.69	Degré entrant d'un sommet . . . . .	274
9.6.70	Degré sortant d'un sommet . . . . .	274
9.6.71	Degré d'un sommet . . . . .	274
9.6.72	Chemin élémentaire . . . . .	276
9.6.73	Parcours . . . . .	277
9.6.74	Parcours partiel . . . . .	277
9.6.75	Sommet ouvert . . . . .	277
9.6.76	Sommet fermé . . . . .	277
9.6.77	Sommet découvert . . . . .	277
9.6.78	Arbre non orienté . . . . .	285
9.6.79	Arbre couvrant . . . . .	286
9.8.80	Graphe valué . . . . .	294
9.8.81	Arbre couvrant minimal . . . . .	296
9.8.82	Coupe . . . . .	299
9.8.83	Sommets incidents . . . . .	299

## Liste des tableaux

2.1	Table de vérité de l'opérateur $\vee$ . . . . .	29
2.2	Table de vérité de l'opérateur $\oplus$ . . . . .	29
2.3	Table de vérité de l'opérateur $\wedge$ . . . . .	29
2.4	Table de vérité de l'opérateur $\neg$ . . . . .	30
2.5	Table des caractères ASCII . . . . .	44
3.1	Caractéristiques des différents supports mémoires . . . . .	48
6.1	Priorités des opérateurs en C . . . . .	88
6.2	Opérateurs arithmétiques . . . . .	89
6.3	Opérateurs bit-à-bit . . . . .	89
6.4	Opérateurs logiques . . . . .	90

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 culturelles 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 :

- mathématiques très utilisées en informatique :
  - ◆ relations binaires et ensembles ordonnés
  - ◆ induction structurelle
- la représentation des données par un alphabet binaire :
  - ◆ les opérations logiques élémentaires sur les mots binaires
  - ◆ les interprétations des mots binaires pour représenter des nombres entiers ou à virgules
  - ◆ les opérations arithmétiques qui en découlent
  - ◆ l'interprétation des mots binaires pour représenter des caractères textuels
- l'architecture fondamentale d'un ordinateur commun
  - ◆ architecture matérielle naïve de Von Neumann
  - ◆ architecture matérielle plus moderne
  - ◆ architecture logicielle commune
- une première approche de la programmation
  - ◆ introduction (très) élémentaire à l'algorithmique
  - ◆ installation des outils de base nécessaires à la programmation en langage C (Linux/Windows)
  - ◆ compilation et analyse d'un premier programme en langage C

Les mathématiques communes à toutes les filières de classes préparatoires sont supposées acquises.

À la fin de cette partie, le lecteur possédera la culture minimale pour apprendre à programmer en comprenant ce qui est manipulé matériellement et logiquement.

## NOTES :

- La lecture du chapitre 1 est recommandée pour la lecture de la partie 3 (et peut être esquivée sinon)
  - La lecture des chapitres 2, 3 et 4 est hautement recommandée pour la lecture de la partie 2.
-

# RAPPELS MATHÉMATIQUES



## RELATIONS BINAIRES ET ORDRE



Les relations binaires sont *extrêmement* utilisées en informatique, raison du rappel. En effet, elles sont équivalentes (strictement le même objet) à la structure de *graphe non enrichi*.

Une relation binaire entre les éléments de deux ensembles  $A$  et  $B$  est un ensemble d'associations entre  $A$  et  $B$ , caractérisant les éléments qui sont “en relation” les uns avec les autres. La nature de la relation en question peut être très variée :

- similarité entre les objets (comme  $=$ )
- hiérarchie entre un tout et ses parties (comme  $\in$  ou son extension  $\subseteq$ )
- comparaison de grandeurs (comme  $\leq$ )
- antécédents et images d'une fonction
- dépendance entre deux événements
- accessibilité d'un point d'arrivée depuis un point de départ ( $\rightarrow$  en théorie des graphes)

Ces exemples parcourent plusieurs grands types de relations en mathématiques, à savoir les relations fonctionnelles, les relations d'équivalence et les relations d'ordre.

### 1.1.1 Relation

#### Définition 1 : Relation

Une *relation* d'un ensemble  $A$  dans un ensemble  $B$  est définie par un ensemble de paires  $\mathcal{R} \subseteq A \times B$ . Si  $(a, b) \in \mathcal{R}$ , on dit que  $a$  et  $b$  sont “mis en relation” par  $\mathcal{R}$  et on note  $a\mathcal{R}b$ . Si  $A = B$ , on dit que  $\mathcal{R}$  est une relation *homogène*.

Ce sous-ensemble de  $A \times B$  définit le *graphe* de  $\mathcal{R}$  qui est noté  $G_{\mathcal{R}}$ <sup>1</sup>

On peut également voir une relation  $\mathcal{R}$  de  $A$  dans  $B$  comme une application

$$\begin{aligned}\mathcal{R} &: A \rightarrow \mathcal{P}(B) \\ a &\mapsto \{b \in B \mid (a, b) \in \mathcal{R}\}\end{aligned}$$

**Exemple :** si  $A = \{0, 1, 2, 3\}$ , on peut représenter la relation homogène

$$\begin{aligned}\mathcal{R} &: A \rightarrow \mathcal{P}(A) \\ a &\mapsto \{a' \in A \mid a' \leq a\}\end{aligned}$$

qui à chaque élément de  $A$  associe les éléments qui lui sont inférieures par le graphe  $G_{\mathcal{R}}$  :

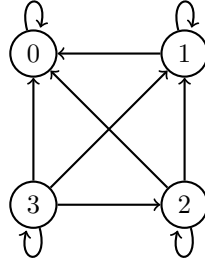


FIGURE 1.1 – Graphe de  $G_{\mathcal{R}}$

Voir une relation de  $A$  dans  $B$  comme une application de  $A$  dans  $\mathcal{P}(B)$  permet de définir le *domaine* et l'*image* de la relation :

#### Définition 2 : Domaine et image d'une relation

Soit  $\mathcal{R} : A \rightarrow \mathcal{P}(B)$  une relation de  $A$  dans  $B$ .

L'image de  $\mathcal{R}$  est définie par :

$$\begin{aligned}\text{Im}(\mathcal{R}) &= \{b \in B \mid a\mathcal{R}b\} \\ &= \{b \in B \mid \exists a \in A / b \in \mathcal{R}(a)\}\end{aligned}$$

Le domaine de  $\mathcal{R}$  est défini par :

$$\begin{aligned}\text{Dom}(\mathcal{R}) &= \{a \in A \mid \exists b \in B, a\mathcal{R}b\} \\ &= \{a \in A \mid \mathcal{R}(a) \neq \emptyset\}\end{aligned}$$

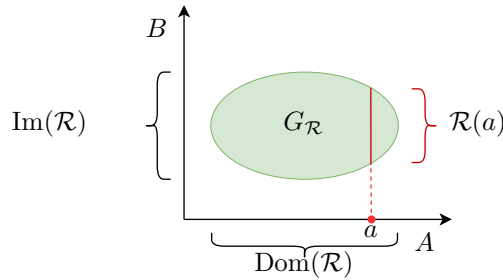


FIGURE 1.2 – Graphe d'une relation

1. On peut aussi définir le graphe comme la relation, ce qui est équivalent. Il s'agit de la même chose

On en déduit la relation de  $B$  dans  $A$  inverse de  $\mathcal{R}$ , noté  $\mathcal{R}^{-1}$ , telle que :

$$\begin{aligned} \mathcal{R} &: B \rightarrow \mathcal{P}(A) \\ b &\mapsto \{a \in A \mid (a, b) \in \mathcal{R}\} \end{aligned}$$

c'est-à-dire telle que  $b\mathcal{R}^{-1}a \Leftrightarrow a\mathcal{R}b$ .

On a immédiatement :

- $G_{\mathcal{R}^{-1}} = G_{\mathcal{R}}^t$
- $\text{Im}(\mathcal{R}^{-1}) = \text{Dom}(\mathcal{R})$
- $\text{Dom}(\mathcal{R}^{-1}) = \text{Im}(\mathcal{R})$

### Définition 3 : Propriétés standards des relations homogènes

Soit  $E$  un ensemble et  $\mathcal{R}$  une relation homogène sur  $E$  (i.e.  $\mathcal{R} \subseteq E \times E$ ).

On dit que  $\mathcal{R}$  est :

- *réflexive* si et seulement si  $\forall e \in E, e\mathcal{R}e$
- *transitive* si et seulement si  $\forall e, f, g \in E, e\mathcal{R}f \text{ et } f\mathcal{R}g \Rightarrow e\mathcal{R}g$
- *symétrique* si et seulement si  $\forall e, f \in E, e\mathcal{R}f \Rightarrow f\mathcal{R}e$  (i.e.  $\mathcal{R} = \mathcal{R}^{-1}$ )
- *antisymétrique* si et seulement si  $\forall e, f \in E, e\mathcal{R}f \text{ et } f\mathcal{R}e \Rightarrow e = f$

## 1.1.2 Relations d'équivalence

### Définition 4 : Relation d'équivalence

Une relation *d'équivalence*  $\mathcal{R}$  de  $E$  dans  $E$  est une relation réflexive, transitive et symétrique.

**Exemple :** les relations homogènes sur  $\mathbb{Z}$  définies pour  $n \in \mathbb{N} \setminus \{0\}$  par :

$$\mathcal{R}_n = \{(a, b) \in \mathbb{Z}^2 \mid \exists k \in \mathbb{Z}, a - b = nk\}$$

sont des relations d'équivalences totales sur  $\mathbb{N}$ .

### Définition 5 : Classes d'équivalences et quotient

Soit  $E$  un ensemble et  $\mathcal{R}$  une relation d'équivalence sur  $E$ . La *classe d'équivalence* d'un élément  $e \in E$  est :

$$[e]_{\mathcal{R}} = \{e' \in E \mid e'\mathcal{R}e\}$$

L'ensemble des classes d'équivalences de  $E$  par  $\mathcal{R}$  appelé *quotient* de  $E$  par  $\mathcal{R}$  :

$$E/\mathcal{R} = \{[e]_{\mathcal{R}} \subseteq E \mid e \in E\}$$

**Proposition 1 (Propriétés des classes d'équivalences).** Si  $\mathcal{R}$  est une relation d'équivalence sur un ensemble  $E$ , alors :

1. tout élément  $e \in E$  appartient à au moins une classe d'équivalence
2.  $a\mathcal{R}b \Leftrightarrow [a]_{\mathcal{R}} = [b]_{\mathcal{R}}$

On en déduit que les classes d'équivalences de  $E$  par  $\mathcal{R}$  forment une partition de  $E$  :

$$E = \bigsqcup_{C \in E/\mathcal{R}} C$$

**Exemple (suite) :** Soit  $n \in \mathbb{N}$ . Les classes d'équivalences d'une relation  $\mathcal{R}_n$  telle que définie plus haut sont les restes possibles par la division euclidienne par  $n$ , c'est-à-dire :

$$\mathbb{Z}/\mathcal{R}_n = \mathbb{Z}/n\mathbb{Z} = \{\dot{1}, \dots, \dot{n}\} \text{ (où on note } \dot{k} = [k]_{\mathcal{R}_n} \text{)}$$

Ce quotient sera utilisé au chapitre 2 pour l'interprétation entière signée des mots binaires.

### 1.1.3 Clôture

Il est parfois pratique de décrire une relation homogène en ne donnant qu'une partie de son graphe, mais en précisant que la relation doit être complétée pour avoir une ou plusieurs des propriétés standards de ce type de relation. Cette complétion de la relation est appelée sa *clôture*.

#### Définition 6 : Clôture

Soit  $\mathcal{R}$  une relation homogène sur un ensemble  $E$ . La *clôture transitive* (resp. *réflexive*, *symétrique*) est la plus petite relation homogène sur  $E$  qui :

- contient  $\mathcal{R}$
- est transitive (resp. réflexive, symétrique).

On note souvent  $\mathcal{R}^*$  la clôture transitive de  $\mathcal{R}$

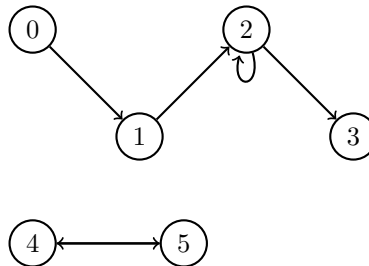
**Proposition 2 (La clôture transitive est l'accessibilité).** Soit  $\rightarrow$  une relation homogène sur  $E$  et  $\rightarrow^*$  sa clôture transitive. Alors :

$$\forall (e, f) \in E^2, e \rightarrow^* f \Leftrightarrow \exists e_0, \dots, e_n \in E, e = e_0 \text{ et } f = e_n \text{ et } \forall i < n, e_i \rightarrow e_{i+1}$$

**Démonstration :** l'énoncé à droite de  $\Leftrightarrow$  définit une relation  $\beta$  transitive qui contient  $\rightarrow$  (puisque  $\rightarrow^*$  est la plus petite qui contient  $\rightarrow$ ) et contenue dans toutes les relations transitives qui contiennent  $\rightarrow$ . Donc  $\beta = \rightarrow^*$

**Interprétation :** si la relation  $\mathcal{R}$  (noté  $\rightarrow$  dans la proposition) permet de se déplacer d'un élément  $e$  vers un autre, la clôture transitive  $\mathcal{R}^*$  représente les éléments accessibles depuis  $e$  par déplacements successifs dans  $\mathcal{R}$ .

**Exemple :** considérons une relation représentée par le graphe :



Les clôtures réflexive et transitive de cette relation sont représentées par les deux graphes suivant :

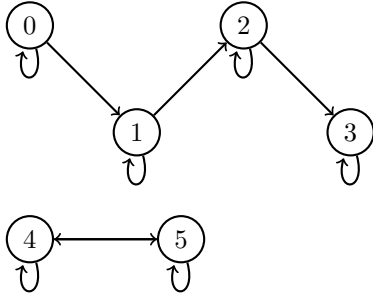


FIGURE 1.3 – Graphe de la clôture réflexive

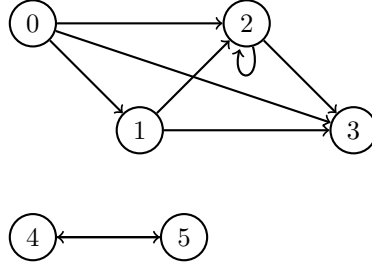


FIGURE 1.4 – Graphe de la clôture transitive

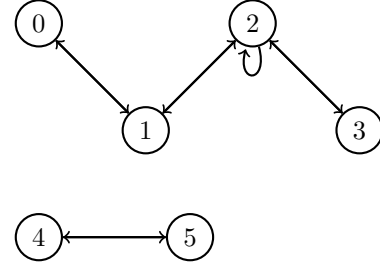


FIGURE 1.5 – Graphe de la clôture symétrique

L'union des arêtes de ces trois graphes donne la clôture réflexive-transitive-symétrique du graphe, qui est donc une relation d'équivalence dont on voit bien les deux classes :

$$\{0, 1, 2, 3, 4, 5\}/\mathcal{R}^= = \{[0]_{\mathcal{R}^=}, [4]_{\mathcal{R}^=}\}$$

### 1.1.4 Relations d'ordre

#### Définition 7 : Relation d'ordre

Une relation d'ordre  $\mathcal{R}$  de  $E$  dans  $E$  est une relation réflexive, transitive et anti-symétrique. On appelle plus simplement une relation d'ordre un *ordre*.

Un ordre  $\mathcal{R}$  est dit *total* si  $\forall e, f \in E, e\mathcal{R}f$  ou  $f\mathcal{R}e$ . C'est-à-dire que tout couple d'éléments peut être comparé. L'ordre est dit *partiel* sinon.

Soit  $\leq$  est un ordre sur  $E$ . On appelle *ordre strict associé* à  $\leq$  la relation  $<$  telle que :

$$\forall e, f \in E, e < f \Leftrightarrow e \leq f \text{ et } e \neq f$$

#### Définition 8 : Bornes par un ordre

Soit  $\leq$  un ordre sur un ensemble  $E$ . Soit  $P \subseteq E$  une partie de  $E$ .

$e \in E$  est un *minorant* (resp. *majorant*) de  $P$  si pour tout  $p \in P$ ,  $e \leq p$  (resp.  $p \leq e$ ). Un minorant  $e$  de  $P$  est un *minimum* (resp. *maximum*) de  $P$  si  $e \in P$ .

#### Définition 9 : Ensembles bien ordonnés

Un ensemble  $E$  muni d'un ordre total  $\leq$  est dit *bien ordonné* si toute partie  $P \subseteq E$  admet un minimum. On dit aussi que  $\leq$  est un *bon ordre* sur  $E$ .

**Proposition 3 (Descente infinie).** Soit un ordre total  $\leq$  sur un ensemble  $E$ . Les phrases suivantes sont équivalentes :

1.  $\leq$  est un bon ordre
2. toute suite décroissante d'éléments de  $E$  est constante à partir d'un certain rang

**Démonstration :**

- ( $\Rightarrow$ ) Supposons que  $\leq$  est un bon ordre. Soit  $s : \mathbb{N} \rightarrow E$  une suite décroissante d'éléments de  $E$ . Comme  $\leq$  est un bon ordre,  $\text{Im}(s)$  admet un minimum. Il existe donc  $N \in \mathbb{N}$  tel que pour tout

$n \in \mathbb{N}, s(N) \leq s(n)$ . Par décroissance de  $s$ , pour tout  $n \geq N$ , on a  $s(n) \leq s(N)$  donc  $s(n) = s(N)$  par antisymétrie.

- ( $\Leftarrow$ ) Supposons que  $\leq$  n'est pas un bon ordre. Il existe donc une partie  $P \subseteq E$  qui n'admet pas de minimum. Pour tout élément  $p \in P$ , il existe donc  $p' \in P$  tel que  $p' < p$  (sinon,  $p$  est un minimum de  $P$  car  $\leq$  est total). On construit alors une suite  $s : \mathbb{N} \rightarrow P$  telle que  $s_0 \in P$  et pour tout  $n \geq 0$ ,  $s_{n+1} = s'_n$  est tel que  $s_{n+1} < s_n$ . Cette suite est décroissante et jamais constante.



## INDUCTION STRUCTURELLE



### 1.2.1 Principe sur un exemple

La récurrence permet de construire des ensembles par répétition, itération, de règles simples. Par exemple, la suite d'éléments suivante  $f : \mathbb{N} \rightarrow \mathbb{N}$  peut être définie par la récurrence suivante :

$$\forall n \in \mathbb{N}, f(n) = \begin{cases} 1 & \text{si } n = 0 \\ 1 & \text{si } n = 1 \\ f(n-1) + f(n-2) & \text{si } n \geq 2 \end{cases}$$

L'induction structurelle est une généralisation de la récurrence et est un moyen pratique de définir des objets complexes. On observe qu'une récurrence est construite en deux phases :

- un ou plusieurs cas de *bases* qui initialisent les objets décrits par la récurrence (pour  $n \leq C$  une constante fixée)
- une ou plusieurs règles de combinaison qui permettent de construire un  $n^e$  élément étant donnés des éléments de rang  $(n-k)^e$

Les règles de combinaison sont appelées règles d'*induction*.

On généralise la récurrence pour construire des ensembles suivant ce schéma. On considère :

- ( $\mathcal{B}$ ) un ou plusieurs cas de *base* qui décrivent directement un ensemble de base  $E_0$
- ( $\mathcal{I}$ ) une ou plusieurs règles dites d'*induction* qui décrivent, étant donné un ensemble  $E_i$  d'éléments déjà décrits, un nouvel ensemble  $\mathcal{I}(E_i)$  d'éléments. On a alors  $E_{i+1} = E_i \cup \mathcal{I}(E_i)$

L'ensemble (potentiellement infini) des éléments décrits par ces règles ( $\mathcal{B}$ ) et ( $\mathcal{I}$ ) est donc :

$$E = \lim_{i \rightarrow +\infty} E_i$$

### Construction des mobiles de Calder

#### Exemple (mobiles de Calder)<sup>2</sup> :

Alexandre Calder est un sculpteur qui doit sa réputation à ses mobiles, qui sont un ensemble d'objets suspendus à des barres elles-mêmes suspendues en équilibre à d'autres barres, etc... L'induction apparaît naturellement dans la construction de mobiles. La contrainte étant que deux mobiles ne peuvent être combinés ensemble par une barre que si ces mobiles font le même poids<sup>3</sup>.

2. L'exemple n'est pas de moi, mais "emprunté" à un excellent livre [3] co-écrit par mon professeur d'informatique de CPGE L. Sartre (apport minoritaire il faut l'avouer puisque son nom apparaît en dernier dans la liste, mais sur 1114 pages, *minoritaire* ne signifie pas peu).

3. Sinon, tout se casse la gueule...

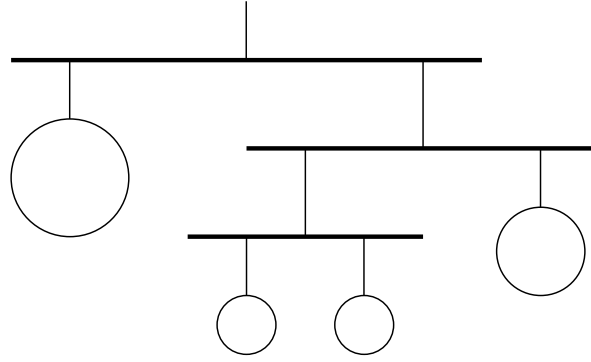


FIGURE 1.6 – Un modèle de mobile de Calder

On peut aisément décrire l'ensemble de base et les règles d'induction permettant de construire des mobiles *quelconques* :

- un objet suspendu seul à un fil est un mobile
- étant donnés deux mobiles, on peut relier ces deux mobiles par une barre pour former un nouveau mobile

**Remarque :** pour l'instant, les mobiles définis comme ci-dessus ne sont pas équilibrés. La définition de fonctions sur l'ensemble des mobiles va permettre d'extraire de  $M$  l'ensemble des mobiles équilibrés, dits de Calder.

Si on note pour tout  $m \in \mathbb{N}$ ,  $O_m$  et  $B_m$  un objet et une barre de masse  $m$ , on peut définir par induction l'ensemble  $M$  des mobiles :

$$\begin{cases} \mathcal{B} & : M_0 = \{O_m \mid m \in \mathbb{N}\} \\ \mathcal{I}(M_i) & : \forall M_1, M_2 \in M_i, \forall m \in \mathbb{N}, B_m(M_1, M_2) \in \mathcal{I}(M_i) \end{cases}$$

ainsi que la fonction *masse* sur l'ensemble  $M$  des mobiles :

$$\begin{cases} \mathcal{B} & : \forall m \in \mathbb{N}, \text{masse}(O_m) = m \\ \mathcal{I}(M_i) & : \forall M_1, M_2 \in M_i, \text{masse}(B_m(M_1, M_2)) = m + \text{masse}(M_1) + \text{masse}(M_2) \end{cases}$$

### Raisonnement sur les mobiles

Le raisonnement par induction est une généralisation du raisonnement par récurrence. Là où le raisonnement par récurrence permet de montrer des propositions sur des objets construits par récurrence, le raisonnement par induction permet de montrer des propositions sur des objets construits par induction.

Si on note pour tout  $m \in M$ ,  $|m|_O$  (resp.  $|m|_B$ ) le nombre d'objets (resp. de barres) dans le mobile  $m$ , on peut chercher à démontrer la proposition élémentaire suivante :

$$\forall m \in M, |m|_O = |m|_B + 1$$

On observe qu'il suffit, comme pour la récurrence, de démontrer la proposition

1. pour l'ensemble décrit par  $\mathcal{B}$
2. pour tout nouvel objet  $m$  construit par  $\mathcal{I}$ .

Puisque tous les objets de  $M$  sont construits par  $\mathcal{B}$  ou par  $\mathcal{I}$ , on aura bien démontré la proposition pour tout  $m \in M$ .

**Démonstration de  $\mathcal{P}$  :** on pose pour tout  $E \subseteq M$  la proposition  $\mathcal{P}_E : \forall m \in E, |m|_O = |m|_B + 1$ .



- ( $\mathcal{B}$ ) : soit  $O_m \in M_0$ , on a bien  $|O_m|_O = 1 + |O_m|_B$ . On a donc montré  $\mathcal{P}_{M_0}$
- ( $\mathcal{I}$ ) : Soit  $i \in \mathbb{N}$ . Supposons  $\mathcal{P}_{M_i}$  vraie. Soient  $M_1, M_2 \in M_i$  tels que  $masse(M_1) = masse(M_2)$ .

Pour tout  $m \in \mathbb{N}$ , on a  $|B_m(M_1, M_2)|_B = 1 + |M_1|_B + |M_2|_B$  et :

$$\begin{aligned} |B_m(M_1, M_2)|_O &= |M_1|_O + |M_2|_O \\ &= 2 + |M_1|_B + |M_2|_B \quad \text{par hypothèse d'induction} \\ &= 1 + |B_m(M_1, M_2)|_B \end{aligned}$$

donc  $\mathcal{P}_{M_{i+1}}$  est démontré.

Par principe de raisonnement par induction,  $\mathcal{P}_M$  est vraie, c'est-à-dire  $\forall m \in M, |m|_O = |m|_B + 1$

**Remarque (C'est SALE!) :**

On peut formaliser un peu plus précisément la construction par induction décrite précédemment. En particulier, la forme des règles ( $\mathcal{B}$ ) et ( $\mathcal{I}$ ) n'est absolument pas précisée. En vérité, le mot "règle" ne décrit rien en lui-même, et la définition de l'induction donné ci-dessus ne veut strictement rien dire du tout !

De plus, on *sous-entend* que l'ensemble des équations décrites par  $\mathcal{B}$  et  $\mathcal{I}$  peut permettre de décrire sans ambiguïtés une *fonction* (dans l'exemple, la fonction *masse*) de manière unique, mais ce n'est pas évident à première vue.

Bref, "*c'est pô propre*" comme dirait Titeuf et malheureusement "*Bis repetita ne placent pas toujours*" mais c'est la vie!<sup>4</sup>.

On cherche donc à nettoyer tout ça en donnant une forme explicite aux (*i.e.* formaliser les) constructions inductives.

### 1.2.2 Formalisation de l'induction

#### Définition 10 : Constructeurs et termes

Un *constructeur* est un symbole attendant un nombre fixe (son arité) d'arguments.  
Un ensemble d'objets inductifs est construit en fournissant un ensemble de constructeurs appelé *signature*. Les éléments de  $E$  sont appelés des *termes*. Les termes sont construits exclusivement par l'application de constructeurs à des termes déjà construits.

C'est-à-dire qu'en notant  $S$  une signature et  $S_n$  l'ensemble des constructeurs de  $S$  d'arité  $n \in \mathbb{N}$  alors on peut redéfinir la construction d'un ensemble  $E$  par induction par :

$$\forall n \in \mathbb{N}, \forall c \in S_n, (t_1, \dots, t_n) \in E^n \Rightarrow c(t_1, \dots, t_n) \in E$$

**Lien avec la première définition :**

- ( $\mathcal{B}$ ) : Les constructeurs d'arité 0 sont appelés des *constantes*. Ils n'ont pas besoin d'arguments. De fait, on a  $E_0 = S_0$ .
- ( $\mathcal{I}$ ) représente simplement l'application *une fois* des constructeurs sur les éléments déjà construits.

**Exemple (reprise des mobiles) :** on peut à nouveau définir les mobiles en se donnant un ensemble  $S$  de constructeurs qui est l'*union* de

4. "Bis repetita placent", paraît que c'est adapté d'Horace, mais je suis à peu près certain que c'est une citation d'Uderzo kekpert dans un Astérix, grande littérature historique qui vaut amplement ses inspirations latines, qu'on se le dise sans sarcasme !

- $S_0 = \{O_m \mid m \in \mathbb{N}\}$  un ensemble de constantes
- $S_2 = \{B_m \mid m \in \mathbb{N}\}$  un ensemble de constructeurs d'arité 2

Par exemple,  $B_2(O_5, B_1(O_2, O_2))$  est un mobile (équilibré (par hasard))<sup>5</sup>.

### 1.2.3 Formalisation du raisonnement par induction

**Remarque (expressivité de l'induction structurelle) :** La première définition de l'induction et son lien avec la formalisation donné ensuite montre que toutes les propriétés que l'on peut démontrer à l'aide du principe d'induction peuvent l'être par récurrence forte sur les tailles des termes. L'emploi de l'induction structurelle est en revanche généralement plus agréable et élégant, puisque ce principe suit directement la structure des termes auxquels on l'applique.

### 1.2.4 Caractérisation inductive d'une fonction

### 1.2.5 Exercices

---

5. Ce serait dommage d'écrire un cours avec des trucs qui se cassent la gueule dedans, ça fait pas très sérieux. . .

# LES DONNÉES



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 ( $U \approx 0\text{ V}$ ), 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 est justifiée par les liens efficaces que l'on peut établir entre ces séquences d'états et  $\mathbb{N}$ , l'ensemble des entiers naturels.

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. 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 (voir la sous-section 2.6.1) et les mots formés de ces lettres comme des textes.

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

2. *idem*

Nom	Symbole	Valeur
kilooctet	1 Ko	$10^3$ octets
megaoctet	1 Mo	$10^6$ octets
gigaoctet	1 Go	$10^9$ octets
teraoctet	1 To	$10^{12}$ octets
petaoctet	1 Po	$10^{15}$ octets

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

Symbole	Valeur
1 Kio	$2^{10}$ octets
1 Mio	$2^{20}$ octets
1 Gio	$2^{30}$ octets
1 Tio	$2^{40}$ octets
1 Pio	$2^{50}$ octets

#### Petit aparté : le bit comme unité d'information

Le bit est l'unité de l'information. Il permet de mesurer la quantité d'information enregistrée<sup>a</sup> puisqu'il correspond à un choix binaire et que le nombre de choix nécessaires pour construire une manifestation d'un type d'objet constitue la quantité d'information contenue par l'objet. 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.

<sup>a</sup>. voir théorie de l'information de Shannon

Les données sur un ordinateur sont mesurés en octets. Originellement, les processeurs n'étaient toutefois pas standardisés et pouvaient diviser les bits par paquets dont la taille différait selon le modèle de l'appareil (par exemple, en paquets de 4 bits).



## OPÉRATIONS LOGIQUES



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

### 2.2.1 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.

### 2.2.2 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>3</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>4</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  $C$  :

$A$	$B$	$C$	$f(A, B, C)$
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 :

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

### 2.2.3 Opérations logiques élémentaires

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

- le OU logique exclusif
- le OU logique inclusif

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

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

- 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 2.1 – 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 2.2 – 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 2.3 – 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 2.4 – 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  $\neg$ ) :

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

## 2.2.4 Opérations de décalage

### Décalage à gauche

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

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

Par exemple :  $(11010110)_2 \ll 3 = (10110000)_2$

### Décalage logique à droite

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

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

Par exemple :  $(11010110)_2 \gg_l 3 = (00011010)_2$

### 2.2.5 Exercices

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

**Exercice 1 (La compréhension pour mieux comprendre) [13].** Décrire les fonctions  $\vee$ ,  $\oplus$ ,  $\wedge$  et  $\neg$  par compréhension grâce à des fonctions élémentaires ( $+$ ,  $-$ ,  $*$ ,  $/$ ,  $|\cdot|$ , etc...)

**Exercice 2 (Universalité des fonctions logiques élémentaires) [23].** Montrer que toute fonction logique  $N$ -aire peut s'exprimer comme une combinaison des fonctions  $\vee$ ,  $\wedge$  et  $\neg$ .

**Exercice 3 (Porte NAND) [19].** 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) [19].** 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) [09].** 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) [06].** 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  $\frac{\mathbb{Z}}{2^N \mathbb{Z}}$ .



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

On considère la fonction suivante :

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

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  $\llbracket 0; 2^N \llbracket$ .

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

$$btoi_u(b) = b_0 + 2(b_1 + 2(b_2 + 2(\dots) \dots)) \quad (2.1)$$

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 (2.1). Pour tout  $i \in \llbracket 0; N-1 \rrbracket$ , on a  $b_i \equiv \left\lfloor \frac{n}{2^i} \right\rfloor [2]$

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 définie 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  $0 \leq a \% b < b$ . Plus précisément,  $a \% b$  donne la classe d'équivalence de  $a$  dans  $\frac{\mathbb{Z}}{b\mathbb{Z}}$

Par exemple, trouvons la représentation binaire de 53 :

- $\left\lfloor \frac{53}{2^0} \right\rfloor = 53 \equiv 1[2]$
- $\left\lfloor \frac{53}{2^1} \right\rfloor = 26 \equiv 0[2]$
- $\left\lfloor \frac{26}{2^2} \right\rfloor = 13 \equiv 1[2]$
- $\left\lfloor \frac{13}{2^3} \right\rfloor = 6 \equiv 0[2]$
- $\left\lfloor \frac{6}{2^4} \right\rfloor = 3 \equiv 1[2]$
- $\left\lfloor \frac{3}{2^5} \right\rfloor = 1 \equiv 1[2]$

• On s'arrête car  $2^6 > 53$ . Du fait de la partie entière, toutes les prochaines divisions donneront 0. D'où  $53 = 1 \times 2^5 + 1 \times 2^4 + 0 \times 2^3 + 1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0 = (110101)_2$

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

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

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

### 2.3.2 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  $v \in \mathcal{B}^N$  quelconque.

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

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

**Proposition 4.**  $(btoi_s(v)) = (btoi_u(w)) \Leftrightarrow v = w$

**Interprétation :**

Quelque soit l'interprétation du mot binaire (signé ou non signé), la représentation binaire est unique. Ainsi, la proposition précédente est équivalente à  $\forall x, y \in \mathbb{Z}, \hat{x} = \hat{y} \Leftrightarrow btoi^{-1}(x) = btoi^{-1}(y)$  (ce qui peut d'ailleurs aussi servir de preuve).

**Proposition 5.**  $b_{N-1} = 1 \Leftrightarrow btoi_s(v) < 0$ .

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

**Proposition 6.**  $btoi_s(v) = -(btoi_s(\neg v) + 1)$

### 2.3.3 Addition

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

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

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

**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.

Autrement dit,  $+$  définit un isomorphisme de  $\frac{\mathbb{Z}}{2^N \mathbb{Z}}$  dans  $\mathcal{B}^N$ .

**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

### 2.3.4 Soustraction

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

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

### 2.3.5 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 définie 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 8.**  $\forall v \in \mathcal{B}^N, \text{btoi}_u(v \gg_l 1) = \left\lfloor \frac{\text{btoi}_u(v)}{2} \right\rfloor$

**Proposition 9.**  $\forall v \in \mathcal{B}^N, \text{btoi}_s(v \gg_a 1) = \left\lfloor \frac{\text{btoi}_s(v)}{2} \right\rfloor$

**Remarque :**  $\text{btoi}_s(v \gg_a 1) = \left\lfloor \frac{\text{btoi}_u(v)}{2} \right\rfloor$  et  $\text{btoi}_u(v \gg_a 1) = \left\lfloor \frac{\text{btoi}_s(v)}{2} \right\rfloor$  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 tandis qu'il doit l'être en représentation signée.

### 2.3.6 Exercices

**Exercice 7 (Du décimal au binaire) [15].**

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

- |       |       |
|-------|-------|
| • 76  | • 211 |
| • 188 | • 4   |
| • 33  | • 238 |
| • 109 | • 161 |
| • 92  | • 126 |

**Exercice 8 (P'tites démos) [18].** Vérifier les propositions 1, 2, 3, 4, 5 et 6.



## OPÉRATIONS FLOTTANTES



### 2.4.1 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, et de manière générale aux applications scientifique de l'informatique.

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>5</sup>
- $\frac{1}{3}$  possède un nombre infini de chiffres après la virgule mais l'information reste finie puisqu'il suffit de stocker la partie entière 0 et la partie répétée des décimales, c'est-à-dire 3 (ainsi l'information d'une 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 contenus par ce nombre diffère (et est parfois impossible comme dans le cas de  $\pi$ ).

**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>6</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.

5. En informatique, une application  $f : E \rightarrow F$  est dite *algorithmiquement calculable* ou simplement *calculable* si il existe un algorithme (*i.e.* procédé effectif de calcul) qui pour tout  $x \in E$  détermine  $f(x)$  en temps fini.

6. Comme les processeurs *Intel x86*

**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 “norme *IEEE 754*”. Il s'agit de la norme la plus largement utilisée en informatique, qui se base sur la notation scientifique, qui permet d'éviter l'écueil soulevé par les remarques 3 et 4.

## 2.4.2 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 en déduit une propriété qui permet de calculer la représentation en base  $b$  d'un nombre réel  $x$  :

**Proposition 10.**  $\forall n \in \mathbb{N}^*, \lfloor b^n \text{fract}(x) \rfloor \equiv c_{(-n)}[b]$ , où  $c_{(-i)}$  est le  $i^{\text{e}}$  chiffre après la virgule dans l'écriture de  $x$  en base  $b$ .

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

- $c_{-1} = \lfloor 2^1 \times 0.125 \rfloor = \lfloor 0.25 \rfloor = 0$
- $c_{-2} = \lfloor 2^2 \times 0.125 \rfloor = \lfloor 0.5 \rfloor = 0$
- $c_{-3} = \lfloor 2^3 \times 0.125 \rfloor = \lfloor 1 \rfloor = 1$
- $\forall i < -3, c_{(-i)} = 0$

d'où :  $x = (10101.001)_2$

On peut ainsi calculer la notation scientifique en base  $b = 2$  de tout réel  $x$  (voir section 2.4.3.1)

**Exercice 9 (Écriture en base 2 de nombres non entiers) [15].** 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.

### 2.4.3 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 :

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

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

Par exemple :

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

Ici :

- $+$  est le *signe* du nombre
- 5.56 est la *mantissee* 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 :

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

La précision de la mantisse est implicitement de 24 bits en précision simple (23 bits explicites). Elle est implicitement de 53 bits en précision double (52 bits explicites). On en déduit :

**Proposition 11 (Limite de précision de la représentation des entiers).** Soit  $x \in \mathbb{Z}$  :

- si  $|x| \leq 2^{24}$ , sa représentation en précision simple (32 bits) selon la norme *IEEE 754* est exacte.
- si  $|x| \leq 2^{53}$ , sa représentation en précision double (64 bits) selon la norme *IEEE 754* est exacte.

#### 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>7</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 (puisque l'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  $21.125 \geq 0$
- 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.

<sup>7</sup> 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.

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,  $bi\text{ais} = 127$ , donc  $N + bi\text{ais} = 0$ . Le nombre est donc dénormalisé<sup>8</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

### 2.4.4 Notation pour la suite

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

On note également  $ftob$  la fonction qui à un réel  $x \in \mathbb{R}_{fN}$  associe le mot binaire  $v \in \mathcal{B}^N$  construit selon la norme *IEEE 754* et  $btof$  la fonction qui à un vecteur  $v \in \mathcal{B}^N$  associe le réel correspondant selon la norme *IEEE 754*.

### 2.4.5 Approximation du logarithme par la norme *IEEE 754*

La norme *IEEE 754* présentée précédemment semble avoir été sortie du chapeau par un coup de baguette magique, comme si une telle norme était “évidente” pour résoudre les problèmes donnés en partie 1.4.1. Cela n'est absolument pas évident, et sera essayé ici de présenter certaines conséquences du choix de cette norme, qui “justifie” son existence *a posteriori*. On pourrait supposer que cette justification *a posteriori* a mené à sa normalisation *a priori*.

Commençons par une observation naturelle, qui consiste à tracer la courbe de la fonction :

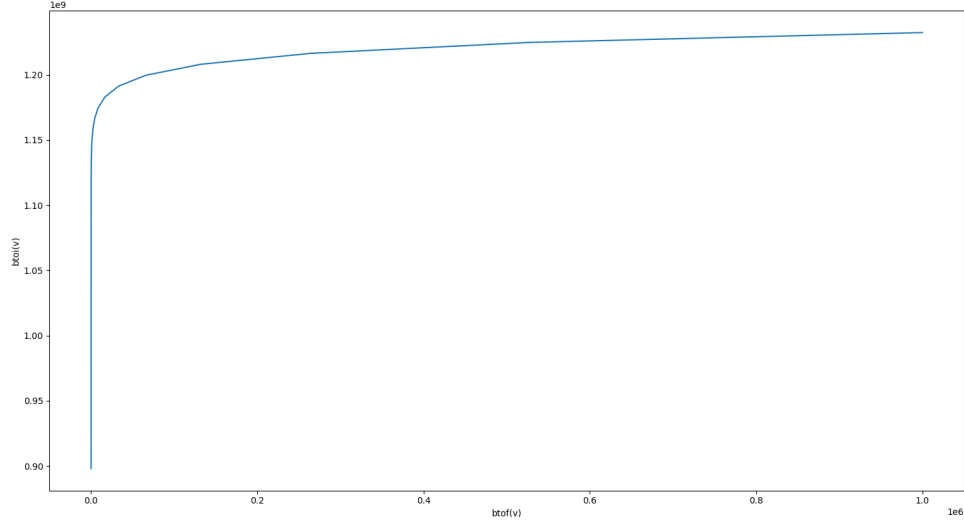
$$\begin{array}{rcl} f & : & \mathbb{R}_{f32} \rightarrow \mathbb{R} \\ & & x \mapsto btoi(ftob(x)) \end{array}$$

On trace cette courbe pour  $10^6$  valeurs uniformément réparties dans l'intervalle  $[10^{-6}; 10^6]$  :

---

8. Comme par hasard...





qui a exactement la même allure que la fonction logarithme. Seule l'échelle semble différer. Il semblerait donc que pour tout  $v \in \mathcal{B}^N$ ,  $btoi(v) \approx \ln(btof(v))$ , à une transformation linéaire près.

Tentons d'y voir plus clair par le calcul. Soit  $v \in \mathcal{B}_{f32}$ . On note  $s$ ,  $m$  et  $e$  respectivement le bit de signe, la mantisse et l'exposant dans  $v$  selon la norme *IEEE 754*. On a :

- $btof(v) = (-1)^s (m \times 2^{-23} + 1) \times 2^{e-127}$
- $btoi(v) = s \times 2^{31} + e \times 2^{23} + m$

De l'observation précédente et des deux expressions données, il semble naturel d'appliquer un logarithme base 2 sur  $btof(x)$  :

$$\begin{aligned} \log_2(btof(v)) &= e - 127 + \log_2(1 + 2^{-23}m) \\ &\simeq (e - 127) + \frac{1}{\ln(2)} (2^{-23}m) \quad \text{car } \ln(1+x) \sim_0 x \text{ avec } x = 2^{-23}m \approx 0 \end{aligned}$$

C'est-à-dire :

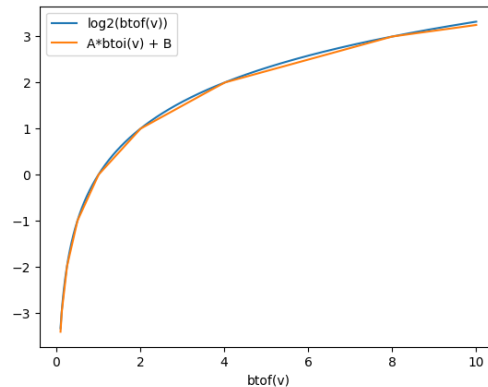
$$2^{23} \log_2(btof(v)) + 2^{23} 127 \simeq \frac{1}{\ln(2)} m + 2^{23} e \approx btoi(v)$$

**Remarque :** l'approximation  $\frac{1}{\ln(2)} \approx 1$  devant  $m$  est de moindre importance que  $\ln(2) \approx 1$  devant  $e$  car  $m$  représente les bits de poids faible de  $btoi(v)$ . Approximer le logarithme népérien serait donc beaucoup moins précis.

On obtient l'approximation du logarithme base 2 sur les flottants :

$$\log_2(btof(v)) = 2^{-23} btoi(v) - 127$$

Comparons avec un graphe sur l'intervalle  $[0.1, 10.0]$  :



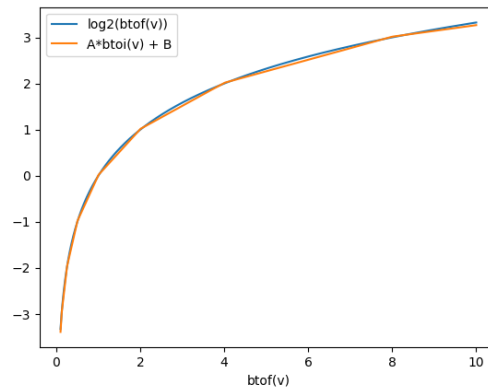
les courbes sont presque confondues mais une imprécision subsiste du fait des approximations effectués durant les calculs.

On peut obtenir une transformation linéaire beaucoup plus précise en utilisant les valeurs normalisés positives minimale et maximale pour calculer la régression linéaire<sup>9</sup>.

On pose  $m$  la valeur minimale et  $M$  la valeur maximale. On a alors :

- la pente de la régression linéaire  $A = \frac{M - m}{btoi(ftob(M)) - btoi(ftob(m))}$  de représentation hexadécimale<sup>10</sup> `0x340003ce`
- la constante à l'origine  $B = \log_2(m) - Am$  de représentation hexadécimale<sup>11</sup> `0xc2fe000f`

On obtient alors le graphe suivant sur l'intervalle  $[0.1; 10.0]$  :



## Conséquences

Deux conséquences à cette approximation relativement précise du logarithme base 2 par la norme *IEEE 754* :

1. la relation d'ordre sur les mots binaires interprétés comme des entiers est la même que celle sur les nombres flottants. Il suffit donc pour comparer deux flottants de comparer l'interprétation

9. On a prouvé son existence comme approximation.

10. Voir 2.5 pour la représentation hexadécimale.

11. *idem*

entière du mot binaire.

2. on obtient une fonction relativement précise du calcul du logarithme base 2, qui si elle n'est pas utilisable pour des cas d'application scientifique pointus suffit amplement pour le reste.

Les deux points soulèvent deux optimisations calculatoire de taille dans la manipulation des nombres flottants.

Ils relèvent aussi la pertinence du choix de cette norme et pas d'une autre.

### 2.4.6 Exercices

**Exercice 10 (Racine carrée inverse rapide (1)) [20M].** Une grande part du temps de calcul dans un jeu-vidéo est destiné à l'affichage 3D, et en particulier au calcul de la direction d'un rayon. Cela sert en particulier au rendu de la lumière. Il faut pour cela normaliser le vecteur de direction. Cela peut servir pour calculer un produit scalaire ou un produit vectoriel. On s'intéresse ici à une optimisation connue sous le nom de *Racine carrée inverse rapide* qui apparaît dans le jeu *Quake III*.

Soit  $v = (x, y, z) \in \mathbb{R}^3$ . On veut calculer le vecteur unitaire  $\frac{v}{\|v\|_2} = \frac{1}{\sqrt{x^2 + y^2 + z^2}}v$ .

Le problème est que les fonctions  $x \mapsto \frac{1}{x}$  et  $x \mapsto \sqrt{x}$  sont très coûteuses en temps de calcul.

Sachant que  $x^2 + y^2 + z^2$  est stocké comme un nombre flottant sur 32 bits, donner une approximation de  $\frac{1}{\|v\|_2}$  qui n'utilise ni  $x \mapsto \frac{1}{x}$  ni  $x \mapsto \sqrt{x}$ .



## 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>12</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 :

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

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

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

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

<sup>12</sup>. En particulier la base octale (c'est-à-dire la base 8) a été utilisé pendant longtemps bien que maintenant assez obsolète.

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>13</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 11 (Conversion binaire-hexadécimale) [15].

Trouver les représentations binaires puis hexadécimale des nombres suivants. On utilisera l'écriture en base 2 (indépendante du signe d'après la **Proposition 1**) 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



## DONNÉES TEXTUELLES



La manipulation de texte au sens humain du terme n'est pas une fonctionnalité naturelle pour un ordinateur.

Pour pouvoir manipuler du texte, il faut représenter ce texte, donc poser une correspondance entre des caractères et des mots binaires.

---

13. Paquet

### 2.6.1 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]

TABLE 2.5 – Table des caractères ASCII

**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 '+'

### 2.6.2 Caractères imprimables et non imprimables

Certains caractères du tableau 2.5, dont la signification est donné entre crochets, sont appelés des *caractères de contrôle* aussi dits *non imprimables*. Ces caractères, qui appartiennent à la première colonne du tableau (plus [SPACE] et [DEL]), ne représentent pas de symboles. Ils sont seulement utilisés pour la mise en page, ou pour fournir des informations dans les télécommunications.

On retiendra tout particulièrement les caractères de contrôle suivant, qui sont souvent utilisés en programmation pour le traitement de texte :

- 0x0 = [NULL] : utilisé de nos jours pour indiquer la fin d'une chaîne de caractères
- 0x8 = [BACKSPACE] : utilisé pour revenir en arrière d'un caractère dans une chaîne. Ainsi, 0x410842 représente non pas 'A' mais 'B'.
- 0x9 = [HORIZONTAL TAB] : tabulation horizontale
- 0xB = [VERTICAL TAB] : tabulation verticale
- 0xA = [LINE FEED] : saut de ligne
- 0xC = [FORM FEED] : saut de page
- 0xD = [CARRIAGE RETURN] : retour chariot : signifie un retour en début de ligne<sup>14</sup>

Le caractère [ESPACE] est en général représenté par un blanc.

**Exercice 12 (Traduction ASCII 1) [10].** 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.*

**Exercice 13 (Traduction ASCII 2) [10].** Écrire en hexadécimale le mot binaire qui décrit le texte suivant :

Oh![LINE FEED]Qui es-tu![BACKSPACE] ?

---

14. Pour les détails : [https://fr.wikipedia.org/wiki/Retour\\_chariot](https://fr.wikipedia.org/wiki/Retour_chariot). On observera que la plupart des caractères de contrôle sont des références aux machines à écrire et à de vieilles normes de télécommunications (parfois encore usitées).

# L'ORDINATEUR

**Notes :** On présente ici les bases fondamentales. L'idée n'est pas (pour l'instant) de faire un cours d'architecture des processeurs (à voir pour un autre cours de Minitel). C'est d'ailleurs très incomplet. Certains détails sont ajoutés durant la partie sur le langage C. Ces détails vont probablement être déplacés ici dans un futur plus ou moins proche. À voir.



## ARCHITECTURE MATÉRIELLE



### 3.1.1 Vue de haut

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



Ces entrées et sorties sont captées et produites via des appareils appelés périphériques d'entrée/sortie. On garde généralement l'acronyme *E/S* (*Entrée/Sortie*)<sup>1</sup>.

---

1. *Input/Output* en anglais

**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".

Le calcul est le plus généralement associé à une série d'actions pouvant être automatisés par un système physique. Certains modèles théoriques représentent cette automatisation et sont d'ailleurs constructibles (comme le modèle RAM qui se rapproche beaucoup des ordinateurs classiques). Il existe en fait des modèles mathématiques parfaitement abstraits dont on a montré qu'ils sont absolument équivalents aux modèles plus "intuitifs" d'ordinateurs.

Ce qui résulte principalement des analyses de ces modèles de calcul est que certains problèmes ne peuvent pas être résolus par des ordinateurs, quelles que soient la forme de ceux-ci, tant qu'ils fonctionnent par le calcul <sup>a</sup>.

<sup>a</sup>. C'est-à-dire : même en supposant qu'ils aient une puissance et une vitesse de calcul arbitrairement grande et une mémoire infinie.

L'acronyme *E/S* est extrêmement récurrent puisqu'il apparaît dans tous les cas où existe un système d'entrée et de sortie d'informations.

### 3.1.2 À l'intérieur

Les périphériques E/S 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 entraît 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 E/S fondamentaux pour une interaction "facile" avec un ordinateur.

L'ordinateur lui-même est uniquement un calculateur possédant une mémoire. Il est représenté de manière par le schéma ci-dessous qui représente la modélisation d'un ordinateur selon l'*architecture de Von Neumann* <sup>2</sup> :

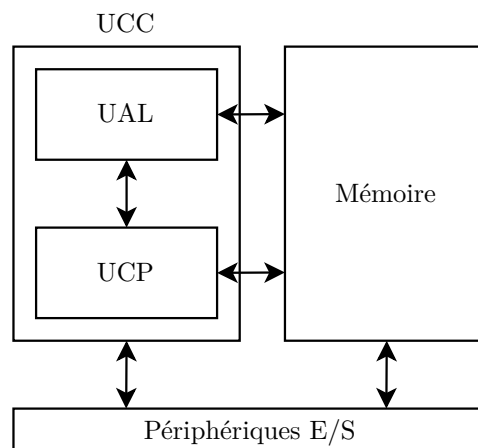


FIGURE 3.1 – Architecture de Von Neumann

2. Mathématicien et physicien, John von Neumann a travaillé sur ce modèle et a publié en 1945 un rapport sur la conception de l'ordinateur EDVAC. Ce modèle est donc vieux et présente des limitations comme cela va être souligné.



**Remarque :** En considérant seulement le système du processeur, 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 processeur, et l'écriture d'une information dans la mémoire constitue une sortie du processeur.

## Unité Centrale de Calcul

L'**UAL** (Unité Arithmétique et Logique) est constituée de circuits électroniques qui permettent d'effectuer toutes les opérations fondamentales de calcul, comme l'addition, la multiplication, la soustraction, la ou la division.

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

L'UCP et l'UAL forment l'**UCC** (Unité Centrale de Calcul, **CPU** en anglais), ou plus simplement *processeur*. Si l'UCP et l'UAL sont assemblés en un seul circuit, on parle alors de *microprocesseur*.

### Définition 11 : Instruction

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

## Mémoire

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), forment un système de stockage de capacité très (très (très)) faible, destiné à stocker des valeurs temporaires. Ils servent en particulier de tampons pour les calculs du processeur.

Type	Temps d'accès	Débit	Capacité
Disques durs	3 – 20 ms	10 – 320 Mio/s	de l'ordre du Tio
Mémoire RAM	5 – 60 ns	1 – 20 Gio/s	de l'ordre du Gio
Mémoire SRAM	2 – 3 ns		de l'ordre du Mio
Registres	≤ 1 ns		de l'ordre du Kio

TABLE 3.1 – Caractéristiques des différents supports mémoires

### 3.1.3 Limitation du modèle de Von Neumann

Le modèle de calcul présenté ci-dessus impose des échanges extrêmement réguliers entre le processeur et la mémoire. Cependant, l'accès à la mémoire est beaucoup plus lent que la vitesse à laquelle les calculs sont effectués par le processeur. En effet, la plupart des processeurs modernes effectuent une

opération élémentaire de calcul en un quart de nanoseconde, ce qui est d'au moins un ordre de grandeur plus rapide que l'accès à la mémoire RAM (voir table 3.1).

Pour cette raison, les constructeurs de processeurs ont situé les registres directement dans le (micro-)processeur pour minimiser les temps d'accès et ont ajouté une mémoire supplémentaire au sein du processeur appelée *mémoire cache*, plus rapide que la mémoire RAM, qui est chargée avec les données de la RAM souvent utilisées. L'idée est que le temps d'accès ultérieur à ces données souvent utilisées sera plus court. Un processeur peut posséder plusieurs niveaux de mémoire cache, notés<sup>3</sup>  $L_1, L_2, \dots$ . Chaque niveau est plus lent que le précédent mais a une capacité plus élevée. Plus une donnée est utilisée souvent et localement, plus bas sera le niveau de cache utilisé et plus rapide sera donc l'accès.

La structure de donnée utilisée pour la mémoire cache est une table d'association, ou dictionnaire (voir la section 9.5).

Finalement, on peut donner un schéma (un peu) plus précis d'un ordinateur moderne :

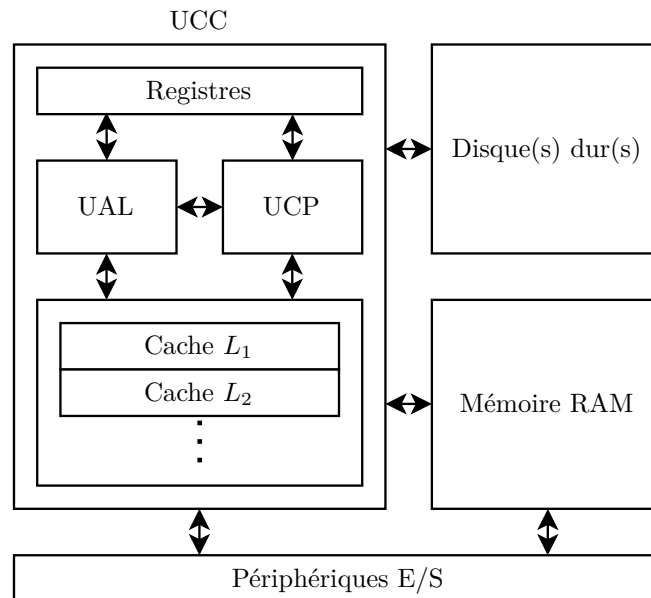


FIGURE 3.2 – Architecture moderne



## ARCHITECTURE LOGICIELLE

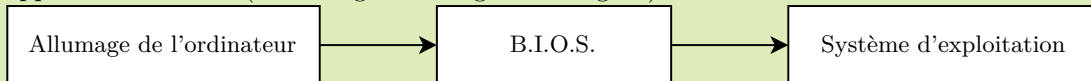


Pas encore écrit... juste quelques définitions.

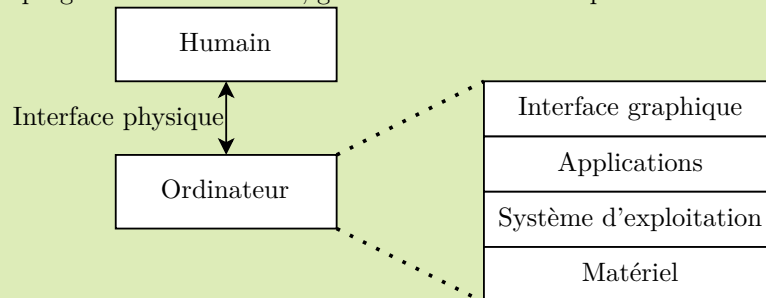
3. "L" pour *Level*

**Définition 12 : BIOS**

Le BIOS (*Basic Input Output System*) 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 13 : 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 14 : 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. Les métadonnées d'un fichier fournissent des informations externes complémentaires.

**Métadonnées du fichier :**

- nom
- taille
- etc...

**Données du fichier :**

- image
- texte
- instructions machine
- vidéo
- etc...

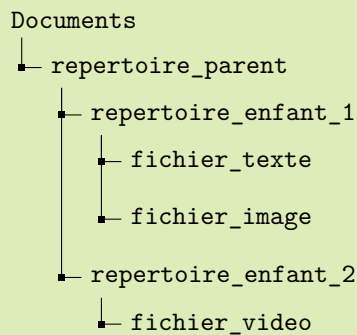
**Définition 15 : Extension de nom fichier**

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 16 : Répertoire et système de fichiers**

Un répertoire est le nom technique donné au dossier. Il s'agit simplement d'un conteneur d'autres répertoires ou fichiers. Il permet de *répertorier* des fichiers ou d'autres répertoires. Sa fonction principale est donc la classification. 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.

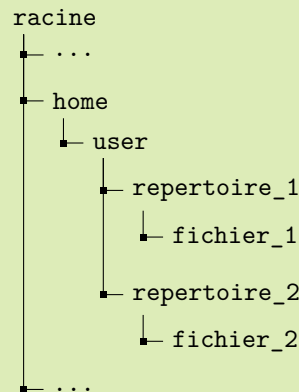
**Définition 17 : Répertoire de travail**

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 18 : 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



Ci-dessus, le chemin absolu du fichier 1 est *racine/home/user/repertoire\_1/fichier\_1*. Si on suppose que le répertoire de travail est *racine/home/user/repertoire\_1*, alors le chemin relatif du fichier 2 est *../repertoire\_2/fichier\_2*.

En effet, “.” désigne le nom du répertoire de travail, et “..” désigne le répertoire parent du répertoire de travail. Ainsi, *../repertoire\_2* désigne le chemin relatif depuis le répertoire 1 équivalent au chemin absolu *racine/home/user/*

# PREMIÈRES NOTIONS DE PROGRAMMATION



## 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 *formellement* 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 *informelle*, c'est-à-dire « avec les mains », qui manque peut-être de rigueur mais qui a le bénéfice d'être simple et suffisante dans un cadre purement pratique.

**Définition 19 : Problème algorithmique (intuitivement)**

Si un problème doit être résolu par un algorithme, c'est qu'étant données certaines données d'entrée du problème, on veut *produire effectivement* certaines données (en sortie donc) correspondant aux solutions du problème.

Un problème est donc décrit en deux parties :

- la description des entrées possibles
- la description du résultat/des effets attendus selon l'entrée

À la section 8.1, on assimilera ainsi un problème à une fonction de l'ensemble des entrées à l'ensemble des sorties.

**Définition 20 : Algorithme (intuitivement)**

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).

À la section 8.1, on définira plus précisément un algorithme comme la réalisation effective selon un modèle de calcul donné d'une fonction décrivant un problème.

**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**

---

**Entrées :** *Entier* :  $x$ , *Entier* :  $y$

```

1 si  $x > y$  alors
2   | retourner  $x$ ;
3 sinon
4   | retourner  $y$ ;

```

---

Cet algorithme est la réalisation effective de la fonction mathématique *max* sur des entiers, qui peut être vue comme un problème à résoudre. Il n'est toutefois valide que si les opérations  $>$  et **retourner** ont été décrites précisément en amont, *pour éviter l'interprétation*. La définition d'un *Entier* doit être également connue.

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

**Définition 21 : 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).

On dit que l'agent *exécute* l'algorithme lorsqu'il effectue les actions décrites par l'algorithme.

Si il est exécuté par un ordinateur avec comme entrées les entiers 3 et 4, l'**Algorithme 1** produira donc le résultat suivant :

4

L'intérêt d'algorithmes est notamment d'automatiser des tâches. On peut notamment imaginer l'algorithme suivant, dont une entrée est un paquet de pâte et dont la sortie associée est une casserole de pâtes cuites :

**Algorithme 2 : Cuisson des pâtes**

- 1 Prendre une casserole;
- 2 Mettre de l'eau dans la casserole;
- 3 Mettre la casserole sur le feu;
- 4 Allumer le feu sous la casserole;
- 5 **tant que** *l'eau ne bout pas* **faire**
- 6   | Attendre que l'eau bout;
- 7 Mettre les pâtes dans la casserole;
- 8 Attendre 9 minutes;

On peut imaginer apprendre cet algorithme à notre singe de compagnie<sup>1</sup>, et simplement lui ordonner d'exécuter cet algorithme par l'instruction "Va faire cuire les pâtes!"<sup>2</sup>.

**Remarque :** En petit malin, le lecteur attentif peut observer<sup>3</sup> que l'*agent* qui exécute cet algorithme doit déjà savoir de quelle casserole, de quel feu, pâtes et eau 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, ce qui n'est pas toujours évident pour un singe).

### 4.1.1 Algorithmes et programmes

**Définition 22 : 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.

1. C'est pas si con les singes... enfin... des pâtes ça doit pas être trop dur, non ?  
 2. Avec un siouplait, c'est mieux, juste histoire de ne pas mettre sa bêtise sur le dos d'une vexation *évidemment* injustifiée.  
 3. C'est la technique classique de "Nan mais t'inquiètes, tu vois l'erreur là ? C'est juste pour voir si tu suis, et pas parce-que j'ai oublié"... *a priori* là ça va :)



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 ou pour faire cuire des pâtes).

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*.

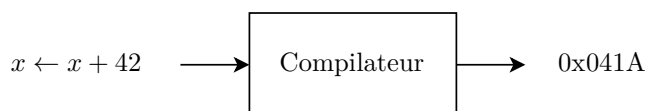


FIGURE 4.1 – Compilation d'un programme  
(damn!<sup>4</sup>)

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 »<sup>5</sup>



## LANGAGES DE PROGRAMMATION



### 4.2.1 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

4. C'est vraiment le mot binaire correspondant à l'ajout de 42 dans le registre AL d'un processeur Intel !

5. Au sujet de la «représentation cérébrale» que nous avons des nombres, je recommande la lecture de Stanislas Dehaene[8].

directement les données des *langages de description*. Les langages décrivant des instructions exécutables par un ordinateur sont appelés des *langages impératifs*.

#### Définition 23 : 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.

Haut niveau : Python, Java, Lisp, etc...
...
Bas niveau : C, C++
Assembleur
Langage Machine

### 4.2.2 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.

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 efficacement que lorsque les ordinateurs ont été assez puissants

pour le permettre.<sup>6</sup> Cela explique par exemple l'explosion de Python dans le monde du développement ces dix dernières années.<sup>7</sup>



## LE LANGAGE C

Le langage C est un langage de bas niveau, c'est-à-dire qu'il est *proche de la machine*. Cela signifie qu'il est nécessaire de bien comprendre l'organisation de la mémoire, la représentation des nombres et de manière générale le fonctionnement interne de l'ordinateur pour en tirer le meilleur parti et éviter certains les écueils inhérents à ce fonctionnement. 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 une meilleure compréhension et un apprentissage facilité des autres langages de programmation.

Le C a été inventé dans les années 70 aux Bell Labs par Denis Ritchie pour lequel il a reçu la *IEEE Richard W. Hamming Medal*<sup>8</sup>, 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<sup>9</sup> 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, le développement de systèmes d'exploitation et de nombreux domaines scientifiques et technologiques. 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>10</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>11</sup>, tiennent plus la main au programmeur en terme de sécurité<sup>12</sup> et/ou de stabilité de développement<sup>13</sup>, proposent plus de fonctionnalités pré-écrites<sup>14</sup>, etc. . .

### 4.3.1 Installer un éditeur et un compilateur

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

6. *Fun fact* : le langage interprété Lisp était très utilisé en intelligence artificielle au XX<sup>e</sup> siècle. Comme les ordinateurs généralistes de l'époque étaient trop lents, des ordinateurs spécialisés pour interpréter ce langage ont été développés : [les machines Lisp](#).

7. 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. . .

8. La médaille Richard-Hamming est décernée chaque année depuis 1988 par l'[IEEE](#), pour honorer les contributions exceptionnelles à l'informatique et aux technologies de l'information.

9. Là j'avoue j'abuse. . . y a des trucs en C qui ont été *designed* avec les pieds. Genre les tableaux statiques par exemple. . . Mais chuut !

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

11. comme la programmation orientée objets

12. On pourra penser au *garbage collector* introduit dans quantité de langages

13. Il faut entendre : cette fois-ci le *design* de base du langage est correct.

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

## 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

```
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 et même nécessaire dans de nombreux cas*)

Il existe plusieurs EDIs/éditeurs de code connus qui permettent d'éditer des programmes dans la plupart des langages de programmation. On se restreindra ici à Sublime Text, un éditeur de texte simple mais complet : <https://www.sublimetext.com/download>. Visual Studio Code est aussi une possibilité, probablement plus connue : <https://code.visualstudio.com/>.

## Interface en ligne de commande

La plupart des ordinateurs proposent aujourd'hui aux utilisateurs *lambda*<sup>15</sup> une interface graphique pensée pour l'utilisation d'une souris, avec des boutons sur lesquelles cliquer. Pourtant, cela n'est que relativement récent (début des années 1980). L'interface graphique n'a été pensée initialement que pour la mise sur marché d'ordinateurs au grand public. En particulier, de nombreuses possibilités d'interactions, de natures techniques, avec l'ordinateur ne peuvent être effectuées grâce à l'interface graphique des systèmes d'exploitations comme Linux ou Windows.

Il faut pour pouvoir utiliser pleinement son ordinateur revenir aux outils accessibles par l'interface en ligne de commande (*command line interface* en anglais). Cela est particulièrement nécessaire pour des systèmes Unix (comme Linux par exemple) qui ont d'abord été pensés à travers ce prisme (contrairement à Windows qui est pensé pour l'interface graphique).

**Sous Linux :** on peut ouvrir généralement une interface en ligne de commande grâce au raccourci clavier CTRL + ALT + T (c'est-à-dire l'appui simultané des touches CTRL, ALT et de la lettre T)

**Sous Windows :** on peut ouvrir une interface en ligne de commande grâce au raccourci clavier Windows + R et en tapant dans la fenêtre qui apparaît soit `cmd`, soit `powershell`.

---

15. Sans rien de péjoratif, précisons.

## Compilateur

### Sous Linux :

Un compilateur du langage C peut-être installé *via* l'interface en ligne de commande :

```

1 user@computer ~> sudo apt-get update && sudo apt-get upgrade
2 user@computer ~> sudo apt-get install gcc
3 user@computer ~> gcc -v
4 NUMERO DE VERSION AFFICHÉE SI INSTALLATION CORRECTE

```

### Sous Windows :

Pour installer un compilateur indépendant sous Windows, il suffit de 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 d'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++.

## 4.3.2 Compiler le premier programme

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>16</sup> :

```

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

```

Le répertoire “src”<sup>17</sup> contiendra les fichiers de code C. Cette arborescence sera développée et étendue dans par la suite.

### Sous Linux :

Il suffit ensuite d'écrire dans l'interface en ligne de commande :

Compiler sous Linux

```

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

```

### Sous Windows :

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

17. Pour “source”

Il suffit d'écrire dans l'interface en ligne de commande :

Compiler sous Windows

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

Dans les deux cas :

La ligne 1 modifie le répertoire de travail du terminal.

La ligne 2 compile notre programme sous la forme d'un exécutable appelé "main.exe" ou "main"<sup>18</sup>

La ligne 3 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.

### 4.3.3 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>19</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.

18. "-o" signifie *output* et "pedantic" spécifie au compilateur d'être intransigeant avec la spécification standard du langage.

19. 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>20</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.

---

20. 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 afin d'amener le lecteur à une maîtrise du langage suffisante pour l'écriture de programmes quelconques sur un ordinateur personnel. Le cours ne cherche donc pas à se spécialiser dans un domaine particulier et aborde de manière générale la programmation en C<sup>21</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<sup>22</sup> qui abordent certaines subtilités et particularités de première nécessité douteuse mais d'utilité parfois avérée

À la fin de cette partie, le lecteur connaîtra tous les aspects du C et saura réaliser n'importe quel algorithme en C. Dans le cadre de l'ISMIN, il ne sera plus en difficulté pour aucune matière nécessitant la maîtrise de ce langage. Pour être plus précis, les deux premiers chapitres sont suffisant dans ce but.

## Limitations

Un langage de programmation est désigné avant tout par sa grammaire, sa syntaxe. Toutefois, celle-ci est profondément liée à la sémantique, c'est-à-dire au sens du programme et plus particulièrement à la manière dont on pense un programme informatique. Un langage est en effet plus qu'un moyen de donner des ordres à un ordinateur. Il sert de cadre pour organiser ses idées à propos de processus.

De ce fait, le simple fait de se limiter au langage C limite fortement la vision de la programmation qui va être développée. De même, il est dommage de n'apprendre que les règles de grammaires du langage et penser savoir programmer de ce fait. D'ailleurs, il suffirait pour cela de lire les 200 pages de spécification syntaxique du langage [7], bourrage de crâne tout à fait inutile, puisque rien ne pourra en être fait<sup>23</sup>.

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

Il peut venir à l'esprit à la lecture de ce livre deux idées opposées. D'abord que les concepts présentés à propos du langage sont présents dans d'autres langages comme le Python, et semblent présenter un

---

21. Rien ne sera donc dit vis-à-vis des spécificités de la programmation de systèmes embarqués bien qu'il s'agisse d'une spécialité de la formation ISMIN, ou d'autres domaines importants aux règles spécifiques comme la programmation hautes performances.

22. Le terme « avancé » ne sous-entend aucune difficulté supplémentaire, il s'agit simplement de la présentation de notions non directement nécessaires.

23. Cette spécification n'est destinée qu'à la lecture de certains points particuliers lors de l'écriture d'un programme pour la précision technique, ou dans le cadre très spécifique de l'écriture d'un compilateur du langage

caractère universel à la programmation (on pensera par exemple au concept de *variable*). Ensuite que certaines particularités pourraient n'être que spécifiques au C et généralisable à aucun autre langage, comme cela peut être pensé des *classes de stockage* – ce qui est évidemment faux puisque des langages comme le Rust le permettent.

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 : si il existe un langage manipulant un concept, on peut construire un deuxième langage utilisant le même concept plus un autre, ce qui rend absurde l'unicité d'un concept au sein d'un langage.<sup>24</sup> 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* peut en fait être ignoré 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 d'accès aux données par adressage mémoire presque universel<sup>25</sup>. Si l'utilisateur ne l'utilise pas consciemment, il apparaît ainsi systématiquement dans le fonctionnement interne des langages<sup>26</sup>.

---

---

24. Hors intuition, cela est démontré mathématiquement par l'équivalence stricte des langages impératifs découlant des modèles de la machine de Turing et du  $\lambda$ -calcul.

25. Ce qui ne signifie pas que le concept de pointeur soit nécessaire. Son apparition dans le langage C et sa pérennité au sein des langages de programmation est d'ailleurs très controversé.

26. 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

# FONDAMENTAUX DU LANGUAGE



## IDENTIFIANTS



### Définition 24 : 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 :

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

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 `#`  $\notin \mathcal{A}$ )
- 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 → clairement interdits
- les identifiants débutant par un underscore → déconseillées, car potentiellement utilisées par la bibliothèque standard

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 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 modules ajoutés 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 bugs 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>

**Remarque :** Ce livre est en français pour être accessible et lisible par des francophones. Cependant, les identifiants sont très souvent standards dans les codes. Il est utile d'avoir l'habitude de lire et d'écrire des identifiants en anglais.



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 destinés 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.
- lorsqu'une bibliothèque est fournie à un client ou publiée sur Internet, les commentaires permettent de fournir une *spécification* à chacun des modules, où des détails d'implantations utiles pour permettre aux codes d'être modifiés par des tiers.

## 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>

**Remarque :** Ce livre est en français pour être accessible et lisible par des francophones. En conséquence de quoi les commentaires seront écrits en français dans tous les codes écrits dans ce livre <sup>3</sup>. Les commentaires ont une vertu explicative, il n'y a aucune utilité à s'entraîner à lire des commentaires écrits en anglais (il suffit d'apprendre l'anglais).

En C, on distingue deux types de commentaires :

- les commentaires sur une ligne
- les commentaires par bloc

---

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

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

3. Qui est déjà en français, donc ça ne change rien à l'absence d'internationalité.

### 5.2.1 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 représente le symbole de retour à la ligne
```

**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.

### 5.2.2 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  Association Minitel
3  Un exemple pour illustrer les commentaires
4
5  #include <stdio.h> // Cette ligne ne sera jamais lue
6  */
7
8  #include <stdio.h> // Inclue le module 'stdio' de la bibliothèque standard
9
10 int main() { // Définit la fonction principale du programme
11     return 0; // Signifie une exécution réussie du programme
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 5.2 et 5.5).

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> // Les instructions de préprocesseurs ne finissent pas par un point-virgule
2
3  int main() {
4      // Les instructions finissent par des point-virgules mais pas les commentaires
5      printf("Salut !\n"); printf("Deuxieme Salut !\n");
6      return 0;
7  }
```

### Petit aparté : de la frustration des erreurs de compilation

Il arrive souvent que les débutants en programmation se sentent frustrés par des erreurs d'un programme aussi simples que l'oubli d'un point-virgule. On a humainement la sensation qu'il ne s'agit pas du coeur du problème et que personne ne devrait avoir à passer du temps là-dessus. Pourtant on fait l'observation que le compilateur agit formellement sur le langage. Il n'y a donc pas d'autres choix que d'écrire du code sans fautes syntaxiques. Ces erreurs "simples" font parties des oublis humains et leur correction fait partie intégrante du travail d'un programmeur. Un ordinateur ne peut rien inventer.

La frustration peut être évitée en comprenant parfaitement les raisons d'être de chaque message d'erreur. Ainsi, le programmeur peut y réagir de manière appropriée. Un des objectifs de ce livre est aussi d'apporter un ensemble de connaissances pour cela.



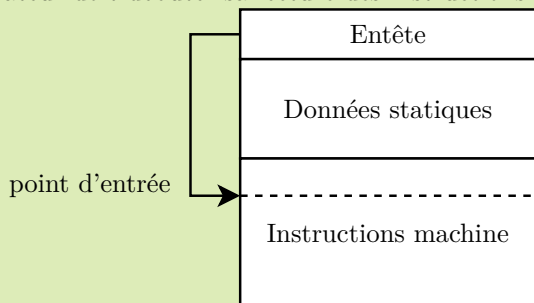


## POINT D'ENTRÉE DU PROGRAMME



### Définition 25 : 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.



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.

À l'exécution du programme, ce sont les  $l$  octets du programme *sans l'entête* qui sont chargés en mémoire.

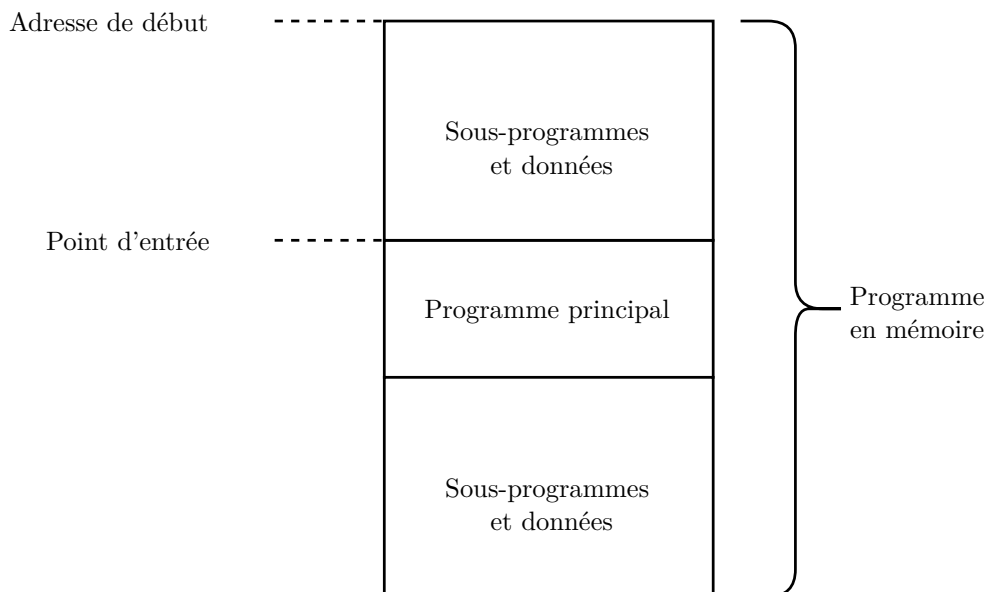


FIGURE 5.1 – Schéma simplifié d'un programme en mémoire

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 le module de la bibliothèque standard `stdlib` qu'il faut donc inclure par la ligne :

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

On peut se demander pourquoi utiliser ces constantes plutôt qu'écrire directement la valeur de sortie soi-même. En fait, la valeur signifiant l'exécution réussie d'un programme diffère selon le système d'exploitation. Sous Windows et Linux, ces valeurs sont :

- `EXIT_SUCCESS = 0`
- `EXIT_FAILURE = -1`

On peut donc écrire :

Exemple

```
1 // L'ordre d'inclusion des modules est (presque) sans importance
2 #include <stdlib.h> // EXIT_SUCCESS et EXIT_FAILURE (entre autre)
3 #include <stdio.h> // printf (entre autre)
4
5 int main() {
6     return EXIT_SUCCESS; // Signifie une exécution réussie du programme
7 }
```

Ces constantes sont définies au sein du module `stdlib` par la directive de préprocesseur `#define`.



## DIRECTIVES DU PRÉPROCESSEUR (1)



Les directives préprocesseurs sont des instructions qui seront exécutés avant le compilateur par un programme nommé le *préprocesseur*. Elles ne font donc pas partie du programme lui-même mais permettent de “préparer” 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<sup>4</sup>.

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 du module 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.

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>5</sup>. Ces directives ne font pas partie *en soi* du langage.

Un approfondissement technique sera effectué ultérieurement (voir la section 7.10).

---

4. Et en fait beaucoup plus, détaillé dans une partie ultérieure.

5. 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

# BASES DU LANGUAGE



Un aspect critique d'un langage de programmation est la possibilité de donner des noms aux objets informatiques construits et manipulés. En langage C, c'est le concept de *variable* qui fournit une telle possibilité.

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.

## 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 :

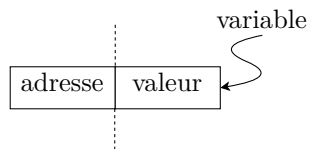


FIGURE 6.1 – Schéma du stockage d'une variable

## Type d'une valeur de variable

### Définition 26 : Type

Une donnée est une séquence finie de bits (0 ou 1). Il faut interpréter cette donnée pour qu'elle soit utilisable dans un contexte de programmation. L'interprétation de données établit une correspondance entre un ensemble de données dit de *représentation*<sup>a</sup> et un ensemble  $E$  quelconque d'éléments. On peut de plus décrire des opérations sur  $E$  qui permettent de manipuler la donnée abstraitement. Ces opérations sont calculées concrètement sur la base de la représentation.

La donnée de la correspondance et des opérations de manipulation des éléments est appelée un *type abstrait*<sup>b</sup>. Il s'agit mathématiquement d'une structure algébrique complexe possédant potentiellement de nombreuses opérations.<sup>c</sup>

On a vu un certain nombre d'interprétations dans le premier chapitre de la première partie (*btoi<sub>u</sub>*, *btoi<sub>s</sub>*, norme *IEEE 754* et caractères ASCII).

a. Donc un ensemble de séquences de 0 et de 1

b. Cette notion sera détaillée un peu dans la section 9.1

c. Si les mathématiques pures permettent de travailler sur les propriétés complexes de structures simples, l'informatique se concentre sur l'étude de propriétés "simples" de structures complexes.

Le langage C propose un certain nombre de types élémentaires directement intégrés au langage. Définir une variable selon un type du langage C consiste à délimiter une séquence de bits en mémoire qui seront mis en correspondance vers un ensemble donné.

Les ensembles mis en correspondances avec ces données en langage C sont des sous-ensembles de  $\mathbb{Z}$  ou de  $\mathbb{R}$ <sup>1</sup>. Une correspondance supplémentaire entre  $\llbracket 0; 127 \rrbracket$  et les caractères de la langue anglaise vient en surcroupe.

Le type permet donc de délimiter les  $N$  octets d'un mot binaire et de l'interpréter. Il n'y a plus besoin de savoir comment est manipulée la représentation interne de l'objet pour le manipuler correctement. Il suffit de connaître précisément l'ensemble abstrait représenté et les opérations définies sur cet ensemble.

Le langage C propose les types de base suivants :

- Anneaux  $\frac{\mathbb{Z}}{2^N\mathbb{Z}}$ , où  $N$  varie selon la taille de la représentation :
  - ◆ **char** : caractères du tableau ASCII, représentée par une donnée sur 1 octet (8 bits) qui peut aussi être interprétée comme une valeur de  $v \in \llbracket -128; 127 \rrbracket$
  - ◆ **short int** : valeur  $v \in \llbracket -2^{15}; 2^{15} - 1 \rrbracket$  représentée par une donnée sur 2 octets (16 bits)
  - ◆ **int** : valeur  $v \in \llbracket -2^{31}; 2^{31} - 1 \rrbracket$  représentée par une donnée sur 4 octets (32 bits)
  - ◆ **long int** : valeur  $v \in \llbracket -2^{63}; 2^{63} - 1 \rrbracket$  représentée par une donnée sur 8 octets (64 bits)
- Certains sous-ensembles de  $\mathbb{R}$  :

1. Ce qui indique soit une interprétation entière soit une interprétation flottante par la norme *IEEE 754*

- ◆ **float** : valeur  $v \in \mathbb{R}_{f32}$  représentée par une donnée sur 4 octets (32 bits)
- ◆ **double** : valeur  $v \in \mathbb{R}_{f64}$  représentée par une donnée sur 8 octets (64 bits)
- ◆ **long double** : valeur  $v \in \mathbb{R}_{f128}$  représentée par une donnée sur 16 octets (128 bits)
- Pour représenter une adresse mémoire : **TYPE\*** : donnée  $d$  sur 8 octets contenant l'adresse d'une autre variable de type **TYPE**

#### Petit aparté : Rapport entre variable informatique et variable mathématique :

Une variable en mathématique désigne un objet indéterminé et sans représentation. 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 monadique quelconque

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
- “ $\forall x \text{ indéterminé}, x \in E \implies \mathcal{P}(x)$ ”, où  $E$  désigne un ensemble quelconque et  $\mathcal{P}$  un prédicat monadique quelconque

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

Par ailleurs, la représentation de cet objet ne lui est pas intrinsèque mais dû à des conventions d'écriture et de notation.

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. À cette adresse mémoire est stockée une donnée sous forme d'un mot binaire. Les mots binaires dans la mémoire d'un ordinateur fournissent une représentation intrinsèque des variables informatique (et en fait de tous les objets manipulés sur un ordinateur).

Les variables peuvent alors potentiellement évoluer et changer de valeur – par la modification des données qui leurs sont associées – au cours du temps. Cette notion de temporalité logique est tout à fait absente du concept de variable mathématique.

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

```
TYPE any_variable; // variable non initialisée : valeur indéterminée
TYPE any_variable = ANY_VALUE; // initialisée à la valeur 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 :

```
int age = 20; // déclare age comme un entier de valeur 20
short int age = 19; // ERREUR : age déjà déclaré
```

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; // ERREUR : 'test' n'est pas encore déclaré
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'*assignation*. Contrairement aux conventions mathématiques, il ne définit pas un prédicat 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
```

### Petit aparté : le type char

Les caractères ASCII ont été abordés dans la première partie. `char` signifie *character* en anglais, c'est-à-dire *caractère* en français.

Le langage C propose le même type pour faire correspondre le sous-ensemble des entiers  $\llbracket -128; 127 \rrbracket$  et l'ensemble des caractères ASCII, bien que l'interprétation soit différente, car la représentation binaire est identique. On peut écrire un caractère ASCII grâce aux guillemets simples :

```
1 | char carac = 'z';
```

ce qui revient strictement à écrire :

```
1 | char carac = 122;
```

### Petit aparté : module pour abstractions entières

L'apparté précédent pointe du doigt l'ambiguïté entre le type des caractères et celui des entiers de  $\llbracket -128; 127 \rrbracket$ .

Le module `stdint.h` de la bibliothèque standard pallie ce problème en proposant les types suivants :

- `int8_t` et `uint8_t` pour les entiers de  $\llbracket -128; 127 \rrbracket$  et  $\llbracket 0; 255 \rrbracket$  respectivement
- `int16_t` et `uint16_t` pour les entiers de  $\llbracket -2^{15}; 2^{15} - 1 \rrbracket$  et  $\llbracket 0; 2^{16} - 1 \rrbracket$  respectivement
- `int32_t` et `uint32_t` pour les entiers de  $\llbracket -2^{31}; 2^{31} - 1 \rrbracket$  et  $\llbracket 0; 2^{32} - 1 \rrbracket$  respectivement
- `int64_t` et `uint64_t` pour les entiers de  $\llbracket -2^{63}; 2^{63} - 1 \rrbracket$  et  $\llbracket 0; 2^{64} - 1 \rrbracket$  respectivement

En particulier, il est préférable d'utiliser (u)`int8_t` pour désigner de petits entiers, puisque le type est celui d'un entier, tandis que `char` devrait être réservé pour stocker des caractères.

Le module `stdint.h` propose également d'autres définitions, notamment les valeurs maximales et minimales pouvant être prises par chaque type. Les détails sont données sur le site <https://cplusplus.com/reference/cstdint/>

On observe une équivalence stricte entre les expressions suivantes :



```
// Première expression :  
TYPE variable = valeur;
```

```
// Deuxième expression :  
TYPE variable; // valeur indéterminée  
variable = valeur;
```

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

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

```
| int a = a; // La valeur de 'a' est toujours indéterminée
```

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

### 6.1.1 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`.

#### 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, idem si non signé  
sizeof(short int); // 2, idem si non signé  
sizeof(int); // 4, idem si non signé  
sizeof(long int); // 8, idem si non signé  
sizeof(float); // 4  
sizeof(double); // 8  
sizeof(long double); // 16  
  
TYPE a;  
sizeof(a); // Nombre d'octets pour stocker 'a' en mémoire  
sizeof(TYPE); // idem
```

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

#### Opérateur `typeof` (pour la culture uniquement)

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 :

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

<pre>typedef(int) a; int b; typedef(b) c;</pre>	<pre>int a; int b; int c;</pre>
---	---------------------------------

On préférera le code de droite pour des raisons d'élégance et de simplicité. Plus un code est lisible, mieux cela est. Par ailleurs, le programme de droite sera compilé un peu plus vite que celui de gauche. *Il est inutile d'être trop intelligent.*<sup>3</sup>

### 6.1.2 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 :

```
// Le signe n'a aucune importance pour la représentation des caractères,
// seule la représentation binaire importe :
char v = 'X';
unsigned short int v2 = -1; // interpretation non signée, la vérité est que v2 = 65535
signed int v3 = -4; // 'signed' est inutile car implicite
// 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.

### 6.1.3 Environnement et blocs

#### Définition 27 : Environnement global

Il semble clair que la possibilité d'associer à des symboles à des valeurs et y accéder plus tard signifie que le compilateur doit garder quelque part en mémoire les paires (*objet, nom*). Cette mémoire est appelée l'*environnement du programme*, et plus précisément l'*environnement globale*.

Cet environnement global désigne un espace de noms. La fonction **main** est un de ces noms, et à ce nom est associé des instructions, un code.

#### Définition 28 : Environnement local

Un environnement local est un sous-espace de noms, relatif à environnement plus global que lui. Les paires (*nom, valeur*) d'un environnement local ne sont pas accessibles hors de cet environnement.

Toutefois, une paire (*nom, valeur*) définie dans un environnement *A* est également définie dans tous les sous-espaces de noms de *A*.

Un environnement local est défini en langage C par un bloc de portée. Les blocs de portée sont indiqués par les accolades gauche et droite { et }. Ils délimitent un environnement local dans lequel des variables pourront être définies. En particulier, il est possible d'imbriquer les blocs entre eux, c'est-à-dire de créer des sous-environnements locaux d'environnements locaux :

3. Enfin, la bêtise c'est quand même très rigolo. C'est une question d'humour au fond :)

```
1  #include <stdlib.h>
2
3  // 'main' est déclarée dans l'environnement global :
4
5  int main()
6  { // Premier bloc : environnement local de 'main'
7
8      int a = 5; // déclarée dans l'environnement local de 'main'
9
10     { // Définit un sous-environnement local dans 'main'
11         short int b = -45.3; // déclarée dans le sous-environnement local
12     } // sortie du sous-environnement local
13
14     // 'b' n'est plus définie, mais 'a' l'est encore
15
16     return EXIT_SUCCESS;
17 } // sortie de l'environnement local de 'main'
18
```

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 appartiennent alors comme `main` à l'environnement global du programme. Ces variables sont alors dites *globales*. Elles sont accessibles dans tous les blocs du programmes même ceux qui ne sont pas le bloc de la fonction `main` :

```
1  #include <stdlib.h>
2
3  int a = VALUE; // variable globale
4  int main() {
5      char b = OTHER_VALUE;
6  }
7
8  /*
9  Un autre code peut aussi accéder à 'a' et à 'main'
10 mais pas à 'b'
11 */
```

**Remarque :** il est assez rare d'avoir à utiliser la fonction `main` hors de celle-ci durant l'exécution du programme.

### Masquage de variables

Le code ci-dessous montre que la déclaration d'une variable dans un sous-environnement local est prioritaire sur la définition précédente d'une variable de même nom dans un environnement hiérarchiquement supérieur. On appelle ce phénomène le *masquage*<sup>a</sup> de variable.

```

1  #include <stdlib.h>
2
3  int main()
4  {
5      int a = 5;
6      {
7          // ici, 'a' vaut toujours 5
8
9          double a = 3.14; // Déclaration d'une nouvelle paire nom-valeur pour 'a'
10
11         // ici, 'a' vaut 3.14
12     }
13
14     // 'a' vaut à nouveau 5
15
16     return EXIT_SUCCESS;
17 }
18
```

Les variables globales peuvent donc aussi être remplacées :

```

1  #include <stdlib.h>
2
3  int a = VALUE; // variable globale
4  int main() {
5      // variable locale qui remplace 'a' dans l'environnement de 'main' :
6      char a = OTHER_VALUE;
7  }
8  // ici, 'a' est égale à VALUE

```

---

a. *shadowing* en anglais

**Cas d'application réel :** On essaie en général d'alléger, dans la mesure où cela est utile<sup>4</sup>, les environnements les plus globaux. Cela permet principalement de réduire la taille de l'espace des noms utilisés à un endroit donné du programme et donc de réduire les bugs d'origine humaine qui consistent à réutiliser deux fois le même nom d'un même environnement dans des contextes différents.

De plus, cela simplifie les dépendances dans le programme et permet au compilateur d'optimiser plus efficacement le code. En effet, le compilateur ne va pas optimiser un code si il n'est pas certain que son optimisation ne va pas modifier le comportement du programme ailleurs.

## 6.1.4 Exercices

**Exercice 14 (Intervention de variables par effet de bord) [10].** Écrire un programme en C qui initialise deux variables *a* et *b* de valeurs différentes. En déclarant au maximum une seule variable

---

4. Je veux dire... Créer un sous-environnement local pour *chaque* contexte de variable, c'est un peut-être un peu abusé. Quoi que...

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 avoir à être modifié si les valeurs de  $a$  et  $b$  sont changées.

## FORMATAGE DE TEXTE

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 dans la bibliothèque standard du C permet d'inclure dans une chaîne de caractères les valeurs de variables quelconques.

Ce chapitre est introduit pour des raisons purement pratique, pour permettre à tout lecteur de tester ses programmes ou d'afficher les résultats d'exécution d'un programme.<sup>5</sup>

Le formatage d'une chaîne dans la bibliothèque standard du C est effectué à l'aide de deux caractères spéciaux :

- le caractère d'insertion de caractères spéciaux \<sup>6</sup>
- le caractère de formatage de variables %

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

- \ est lui-même un caractère spécial en C
- % n'est pas un caractère spécial mais les fonctions de la bibliothèque standard du C le considère comme spécial puisqu'utilisé pour le formatage.

```

1  #include <stdio.h>
2  #include <stdlib.h>
3
4  int main() {
5  // produit un avertissement à la compilation et affiche "'est pas affichable." :
6  printf("% n'est pas affichable.");
7  // erreur de compilation car '\ ' n'est pas un caractère
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 "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");
```

5. Par ailleurs, le formatage de chaîne de caractère tel que présenté n'est pas une caractéristique du langage C à proprement parler mais bien plutôt une caractéristique de la bibliothèque standard du langage C.

6. La description de caractères spéciaux fait partie du langage C et n'est pas spécifique à une bibliothèque. Il ne s'agit pas de formatage

```

6 | printf("\\ non plus.\n");
7 | return EXIT_SUCCESS;
8 | }

```

**Remarque :** Comme % n'est pas un caractère spécial spécifique au langage C mais seulement à la bibliothèque standard, le caractère spécial '%' n'existe pas, et l'échappement de % est lui aussi spécifique à la bibliothèque standard.

## 6.2.1 Formatage des caractères spéciaux

Les caractères considérés comme spéciaux sont :

- caractère de passage à la ligne suivante : \n<sup>7</sup>
- caractère retour au début de la ligne actuelle : \r<sup>8</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 | }

```

## 6.2.2 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

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 :

7. Ce caractère est désigné en anglais par le symbole LF pour "Line Feed"

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

```

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 :

```

// Erreur : aucun entier n'est donné ici :
printf("%u est un entier non signé");
// Erreur : 42 est en trop :
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  }

```

### 6.2.3 Exercices

**Exercice 15 (Taille des types) [06].** Écrire un programme en C qui affiche la taille en octets des différents types de base du langage en utilisant `sizeof`.



## OPÉRATEURS SUR LES VARIABLES



### Définition 29 : 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 en C, qui permettent d'agir soit sur la donnée de représentation d'une variable (le mot binaire) soit sur la valeur de la variable (dans l'ensemble déterminé par son type).

- les opérateurs arithmétiques agissent sur les valeurs
- les opérateurs bit-à-bit agissent sur la représentation
- les opérateurs logiques/relationnels agissent sur les valeurs
- les opérateurs d'assignation agissent sur les valeurs

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

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

// opérateur unaire :
<OPERATION> v1; // le résultat n'est pas stocké

// opérateur binaire :
v1 <OPERATION> v2; // le résultat n'est pas stocké
```

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)); // l'opérateur OP1 est calculé avant OP2
```

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 ci-dessus. Dans le cas où <OP2> est prioritaire sur <OP1>, cela diffère et <OP2> est évalué avant <OP1>

Pour tous les opérateurs présentés dans la suite, on pourra se référer à la table de priorités suivante <sup>9</sup> :

9. 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 6.1 – 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...

### 6.3.1 Opérateurs arithmétiques

Les opérateurs arithmétiques agissent sur la valeur de la variable selon son type. On peut donc se référer aux interprétations décrites dans l'introduction du cours et aux opérations relatives à ces interprétations ;

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

TABLE 6.2 – 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", a + b);
9      printf("a - b = %d\n", a - b);
10     printf("a * b = %d\n", a * b);
11     printf("a / b = %d\n", a / b);
12     printf("b / a = %d\n", b / a);
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  $67 = 18 \times 3 + 13$

Pour modifier la valeur déjà existante d'une variable, il faut ajouter un opérateur d'assignation de valeur (voir sous-section 6.3.4.1) qui va modifier en mémoire la valeur (c'est-à-dire la représentation binaire selon le type) de la variable.

### 6.3.2 Opérateurs bit-à-bit

Les opérateurs bit-à-bit agissent directement sur la représentation binaire d'une variable et ne tiennent pas compte de sa valeur. La modification de la représentation entraîne la modification de la valeur.

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

TABLE 6.3 – Opérateurs bit-à-bit

Ces opérateurs sont décrits en détails dans la partie I du cours, dans la section 2.2

**Attention :** Les symboles utilisés pour le OU exclusif en C et le ET mathématique sont ressemblant.

#### À propos des flottants :

Les nombres à virgules ne sont pas traités physiquement par la même unité de calcul que les nombres entiers. En particulier, l'unité de calcul des nombres flottants ne contient pas d'opérateurs bit-à-bit.

Le code suivant provoque donc une erreur à la compilation :

```
1 float pi = 3.14; // sur 32 bits
2 float abs_pi = (pi & 0x8fffffff); // est censé annuler le bit de signe -> ERREUR
```

Il est nécessaire d'appliquer une projection de type (voir section 6.4) en entier qui va changer le registre du processeur utilisé pour stocker la représentation. Sur le nouveau registre sera susceptible d'être appliqués des opérations bit-à-bit.

### 6.3.3 Opérateurs logiques, ou relationnels

#### Définition 30 : Type booléen et valeur de vérité d'une expression

Le type abstrait *booléen* est déterminé par l'ensemble  $\{Vrai, Faux\}$  et des opérations logiques  $\vee, \wedge, \oplus$  ou  $\neg$ .

La représentation en mémoire de *Faux* est le mot binaire nul, dont la valeur selon une interprétation entière est 0.

Tous les autres mots binaires correspondent à la valeur *Vrai*.

On dit alors qu'une expression numérique (à valeur dans  $\mathbb{R}$ ) 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 *Vrai* ou *Faux*, représentés par 1 ou 0.

Opération	Arité	Symbole mathématique	Symbole en C
Égalité	2	=	==
Différence	2	$\neq$	!=
ET logique	2	$\wedge$	&&
OU logique	2	$\vee$	
NON	1	$\neg$	!

TABLE 6.4 – Opérateurs logiques

**Remarque :** Il s'agit d'un OU *inclusif*. Le OU exclusif n'a pas de version "logique" en C. Ainsi, en C,  $2 \wedge 1 = 3$  est toujours vrai alors que mathématiquement  $Vrai \oplus Vrai = Faux$

Quelques exemples :

```
1 int a = 5;
2 int b = 7;
3 printf("%d\n" a == b); // -> 0
```

```

4 printf("%d idem que %d\n", a != b, !(a == b)); // -> 1
5 printf("%d\n", a != b && a == b); // -> 0
6
7 printf("%d\n", !(b-a)); // b-a = 2 est vrai donc !(b-a) est faux -> 0
8 printf("%d\n", !(a + b)); // -> !0 = 1
9 printf("%d\n", !(a+b) == (a != b)); // -> 1
10 printf("%d\n", a == b || (a + b)); // -> 1
11 printf("%d\n", a-b && (a != b)); // -> 1 car les deux propositions sont vraies

```

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

### 6.3.4 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 :

```

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 présentes dans ce code sont expliquées dans la suite<sup>10</sup>.

#### 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);

```

10. Bien qu'une exécution et quelques tests accompagnés d'un peu d'intuition pourraient suffire à comprendre

```

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 = ...; // valeur quelconque
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>11</sup>
- `--variable`

Il y a équivalence entre les expressions suivantes :

<pre> int a = 0; a += 1; a -= 1; </pre>	<pre> int a = 0; ++a; // renvoie 1 --a; // renvoie 0 </pre>
---	---

Ainsi, `++a` renvoie la valeur de *a* après incrément 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++; // renvoie 0
a--; // renvoie 1
a; // renvoie 0

```

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

Dans ce cas, la valeur renvoyée est celle *avant* l'incrément ou la décrémentation. Ce qui explique le décalage par rapport au 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;
```

### 6.3.5 Exercices

Voir [2] pour approfondir les calculs binaires optimisés.

**Exercice 16 (Valeur) [08].** 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 17 (Calcul d'expressions) [10].**

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 18 (Priorité des opérateurs) [11].** 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 19 (Interversion sans effet de bord (1)) [15].** 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 20 (Interversion sans effet de bord (2)) [15].** *idem* que pour l'Exercice 19 mais l'unique opérateur autorisé est l'opérateur de OU exclusif (noté  $\wedge$  en C).

**Exercice 21 (Multiplication par décalage) [12].**

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      // ici : code à déterminer
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 22 (Valeur absolue (1)) [18].** En utilisant uniquement des opérations arithmétiques et bit-à-bits, écrire un programme qui calcule la valeur absolue d'un entier signé :

```

1  int x = ...; // valeur quelconque, positive comme négative
2  unsigned int abs_x;
3  // ici : code à déterminer;
4  printf("|%d| = %u", x, abs_x);
```



## 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 :

```

int some_integer = 7851;
int divider = 100;
double a = some_integer/divider;
printf("%.21f\n", a);
```

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 converti 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 reinterpret_variable = (TYPE2)(variable);
```

**IMPORTANT** : La projection de type effectue dans certains cas un changement de valeur sans modification de la donnée de représentation (c'est-à-dire du mot binaire) et dans d'autres cas un changement de la donnée de représentation pour rester au plus proche de l'ancienne valeur<sup>12</sup>. Ainsi :

- projection de nombre à virgule vers entier : représentation modifiée et valeur approximée de l'ancienne (car perte de décimales)
- projection d'entier vers nombre à virgule : représentation modifiée et valeur approximée de l'ancienne (car perte de précision du fait de la norme IEEE 754)
- projection d'entier signé vers entier non signé, ou inversement : représentation non modifiée et valeur modifiée (simple changement d'interprétation)
- projection de pointeurs  $T_1^*$  vers  $T_2^*$  : représentation non modifiée<sup>13</sup>

Dans notre cas, il suffit d'écrire :

```
int some_integer = 7851;
int divider = 100;
double a = (double)(some_integer)/divider;
printf("%.2lf\n", a);
```

Le résultat après exécution est bien : 78.51

Dans le cas d'une projection d'un nombre entier non signé vers un nombre entier signé, on a bien une simple réinterprétation :

```
unsigned short int v = -1;
printf("%d\n", v); // -> 65535
printf("%d\n", (signed short int)(v)); // -> -1 mais il s'agit du même mot binaire
```

## 6.4.1 Exercices

**Exercice 23 (Quelques évaluations entières) [13].**

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

12. Parce-que le langage C est conceptuellement claqué au sol, mais *trènnkile*.

13. et c'est tout car la valeur et la donnée des pointeurs sont identiques



```

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 31 : Flot d'exécution/de contrôle

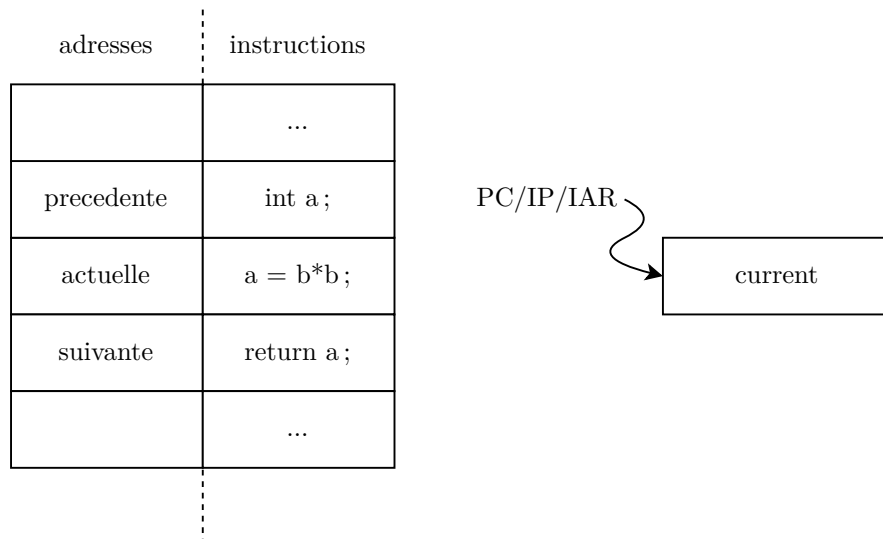
Le flot d'exécution ou flot de contrôle est l'ordre dans lequel les instructions d'un programme impératif sont exécutés. Il est possible de modifier cet ordre grâce à des structures de contrôle de flot.

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 de 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>14</sup> Il est possible d'aller directement à la sous-section 6.5.2 décrivant les structures complexes, majoritairement utilisées en langage C.

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

### 6.5.1 Structures élémentaires

Voir [6] pour plus de détails.

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

- la séquence
- l'arrêt de programme
- le saut inconditionnel
- 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.

La fin d'une 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 contiguës en mémoire.

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>15</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 indiqué par un *label* définie selon la syntaxe suivante :

```
| mon_label:
```

Un exemple :

---

15. Les processeurs Intels utilisent des instructions de taille variable tandis que les processeurs ARM utilisent des instructions de taille fixe

```

1  #include <truc.h>
2
3  int main() {
4      goto fin;
5      // ici : opérations jamais exécutées
6  fin:
7      operations_de_fin();
8      return EXIT_SUCCESS;
9  }

```

Pensez à oublier ça, tant qu'on ne maîtrise pas absolument le reste, il faut éviter... et même là...

## 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.

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>16</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  ; point-virgules pour les commentaires
2
3  section .text ; début du programme
4      global _start
5
6  _start: ; pareil que fonction 'main' en C
7
8  mov al, 200 ; al est un registre 8 bits, et 2^8 = 256
9  add al, 100 ; 200 + 100 > 256, donc CF = 1
10  jc fin ; JC signifie Jump if Carry -> saute si retenue
11      ; le code ici est ignoré car CF != 0
12  fin: ; le code qui suit clôt l'exécution du programme
13  mov rax, 231
14  mov rdi, 0
15  syscall

```

16. 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

`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 3 :** 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 | boucle:
4 | ; trucs à répéter 10 fois
5 | sub al, 1
6 | jnc boucle ; saute si  $al - 1 \geq 0$ 
7 | fin_de_boucle:
8 | ; ...
```

## 6.5.2 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>17</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 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.

---

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

### 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 4 : Branchement conditionnel

---

```

1 si condition  $\neq$  0 alors
2   | Instructions si condition  $\neq$  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  $\neq 0$  est implicite. Ainsi, les deux codes suivants sont équivalents :

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

On pourrait aussi écrire de manière plus lisible :

<pre> int a = 4; int b = 6; if (a != b) {     printf("a different de b."); } else {     printf("a = b"); } </pre>	<pre> int a = 4; int b = 6; if ((a != b) != 0) { // ou a != b != 0     printf("a different de b."); } else {     printf("a = b"); } </pre>
---	--

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) { // ou 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) { // ou 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"); // sera toujours exécuté
```

## 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>18</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<sup>19</sup>.

Un petit exemple d'utilisation :

```
1  /*
2  <math.h> nécessite l'option -lm
3  à la fin de la ligne de commande pour compiler :
4  gcc main.c -o main --pedantic -lm
5  */
6  #include <stdlib.h>
7  #include <math.h> // NaN (Not A Number), sqrt
8
9  int main() {
10     double a = ...; // valeurs
11     double b = ...; // quelconques
12     double c = ...;
13
14     // Résoud une équation à solutions réelles de degré 2
15     double d = b*b - 4*a*c;
16     double x1 = (d < 0) ? NAN : (-b - sqrt(d))/(2*a);
17     double x2 = (d < 0) ? NAN : (-b + sqrt(d))/(2*a);
```

18. 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)

19. Il faut préciser que le compilateur effectue le plus souvent lui-même l'optimisation si vous oubliez d'utiliser une condition ternaire.

```
18 |  
19 |     return EXIT_SUCCESS;  
20 | }
```

## 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 <sup>20</sup>
  - ◆ 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 prédéterminées à la compilation (pas de variables) et *entières*

Cet aiguillage sert en général à effectuer un choix, qu'il s'agisse du parcours d'une structure informatique complexe <sup>21</sup>, ou d'un simple menu dans l'application d'un restaurant <sup>22</sup>.

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

```
unsigned int x;  
switch (x) {  
case 0:  
    printf("Choix 0");  
    // et d'autres instructions  
    break;  
case 1:  
case 2:  
    printf("Choix 1 ou 2");  
    // et d'autres instructions  
    break;  
case 3:  
    printf("Choix 3 puis 4");  
    // et d'autres instructions  
case 4:  
    printf("Choix 4");  
    // et d'autres instructions  
    break;  
default:  
    printf("Autre");  
}
```

20. C'est-à-dire qu'aucun branchement conditionnel n'est effectué quand on a au moins 5 valeurs entières proches (je n'ai pas vérifié ce que signifiait *exactement* "proche"), ce qui est *extrêmement* appréciable en terme de performances.

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

22. C'est pas très sexy dit comme ça, mais c'est le quotidien des examens d'Algo/Prog de l'ISMIN

```
// et d'autres 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"<sup>23</sup> 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  $x = 3$  est suivie 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 5 : 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) {
    // bloc d'instructions exécutés tant que (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);
```

---

23. Littéralement.



```

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 24 [07].** 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 {     // Bloc d'instructions } while (condition); </pre>	<pre> // Bloc d'instructions while (condition) {     // Même bloc d'instructions } </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) {
    // Bloc d'instructions
}

```

et est équivalente au pseudo-code suivant :

---

#### Algorithme 6 : Boucle *for*

---

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

---

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 6.9 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 :

```
// Boucle infinie :
for (;;) {

}

// Boucle infinie incrémentant une variable :
for (int a = 0; ; a++) { // a++ <=> a += 1

}

// Boucle finie qui démarre en affichant du texte
int a;
for (printf("Initialement, a = %d\n", a = 1); a < 10; printf("À présent : a = %d\n", ++a));

// Boucle rigolote (?)
for (unsigned int i = 1; i < 7; i++) {
    printf("%u * 142857 = %u\n", i, 142857*i);
}
```

### 6.5.3 Détails sur l'instruction `break` (et l'instruction `continue`)

L'instruction `break` a été rapidement vue pour l'aiguillage. Mais une question – légitime – subsiste quand aux blocs d'instructions qui peuvent être "cassés" et ceux qui ne peuvent l'être.

La réponse tient simplement : l'instruction `break` casse les blocs d'instructions `while`, `do-while`, `for` et `switch`, sans aucune exception. Le contrôle du programme est ensuite directement rendu à la première instruction suivant le bloc. Par exemple :

```
#define N 28

...

unsigned int i = 0;
while (1) {
    if (i*i >= N) {
        break;
    }
    i++;
}
printf("Plus petit i tel que i*i >= %u : %u", N, i); // -> 6
```

L'instruction `continue` est le dual de l'instruction `break` pour les blocs `while`, `do-while` et `for` – mais pas pour les `switch`. L'instruction `continue` permet de sauter à la première instruction de la boucle, *sans*

exécuter le test de boucle<sup>24</sup>.

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



Plus sérieusement, il est recommandé de ne pas utiliser le `goto` l'immense majorité du temps.<sup>25</sup> En effet, ce genre de saut est effectué sans contrôle (donc augmente les risques de bugs) 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. Par ailleurs, la lecture seule d'une instruction `goto` ne donne aucune information supplémentaire à son sujet comme les raisons pour lesquelles le saut est effectué.

Cela ne signifie pas que l'outil soit tout à fait inutile, simplement que son utilisation doit être restreinte à des cas d'utilité avérée.<sup>26</sup>

### 6.5.4 Exercices

Dans la suite,  $N$  désigne une constante définie par un `#define`.

**Exercice 25 (Question d'âge) [10].** 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>27</sup>

Essayer de minimiser le nombre de branchements conditionnels (objectif : trois `if` maximum, et seule la phrase correcte doit s'afficher<sup>28</sup>).

**Exercice 26 (Suite de Fibonacci) [15].** Écrire un programme qui calcule les  $N$  premiers éléments de la suite de Fibonacci  $(f_n)_{n \in \mathbb{N}}$  définie telle que :

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

24. Ce qui rend compatible l'instruction avec les boucles `do-while`

25. Dire ça c'est un coup à avoir un lecteur à l'ego démesuré qui va penser être le type exceptionnel qui a des problèmes si extraordinaires qu'il lui *faut* le `goto`. Ou qu'il est tellement malin qu'avec son `goto` à tort et à travers, il révolutionne les styles de programmation. Au risque de le décevoir...

26. On peut penser à des transitions de machines d'états.

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

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

**Exercice 27 (Test de primalité)** [24M]. Écrire un programme qui vérifie si  $N$  est premier. On essaiera d'avoir un maximum de  $\sqrt{N}$  tours de boucle. On justifiera la correction du programme<sup>29</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 bugs en tous genres, dont les origines sont humaines.

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 debuguable 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
- relativement facilement optimisable, puisqu'optimiser un sous-programme une seule fois peut optimiser en plusieurs endroits le programme principal si celui-ci appelle régulièrement le sous-programme optimisé.

Nous allons voir un cas particulier de sous-programme appelé la *routine*<sup>30</sup>, dont il n'existe que deux formes<sup>31</sup> en langage C :

- la procédure : effectue une action dépendante de ses entrées et ne renvoie rien
- 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) {     // bloc d'instructions     return; // termine la routine }</pre>	<pre>TYPE function_name(parametres) {     // bloc d'instructions     return VALUE; // VALUE de 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;
}
```

On observe que comme pour une fonction mathématique, les paramètres d'une routine appartiennent chacun à un ensemble qui peut être différent. Les fonctions ne renvoient qu'une seule valeur. La fonction

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

30. 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.

31. 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.

d'addition ci-dessus est équivalente à la fonction mathématique suivante :

$$\begin{array}{ccc} \text{addition} & : & \llbracket -2^{31}; 2^{31} - 1 \rrbracket^2 \rightarrow \llbracket -2^{31}; 2^{31} - 1 \rrbracket \\ & & (a, b) \mapsto a + b \end{array}$$

**Précision :** dans les faits, l'instruction `return` saute à la suite de l'instruction ayant appelée la routine. Ainsi `return` peut être placé n'importe où dans la routine. Elle y mettra fin :

```
void afficher_diff_0(int x) {
    if (x == 0) {
        return; // termine la routine en sautant à la suite de l'instruction qui l'a appelée
    }
    printf("x != 0\n"); // jamais exécuté si x = 0
}
```

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 et pas à la fin.

```
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*.

### 6.6.1 Signature et prototype

#### Définition 32 : Signature

La signature d'une routine est la donnée de :

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

La signature est l'identité de la routine, elle est indépendante du langage.

On retiendra la notation suivante pour les signatures :  $nom(t_1, \dots, t_n) \rightarrow t_r$ , où les  $t_i$  sont les types abstraits des paramètres et  $t_r$  est le type abstrait de la valeur de retour. Dans le cas d'une procédure, on peut "oublier" le retour et écrire  $procname(t_1, \dots, t_n)$

**Exemple :**  $\text{addition}(\text{int}, \text{int}) \rightarrow \text{int}$  est une signature d'une fonction *addition* qui prend en entrée deux paramètres entiers et renvoie un entier.

**Remarque :** Une signature ne contient que des informations syntaxiques. Rien n'est dit de la sémantique de la routine. Les informations syntaxiques sont cependant suffisantes pour déterminer lors d'un appel de routine à quelle routine cet appel fait référence.<sup>32</sup>

### Définition 33 : Prototype

Le prototype est une notion spécifique aux langages C et C++. Il sert à indiquer au compilateur que le corps (c'est-à-dire le contenu) d'une routine sera bien défini plus tard dans le programme. L'utilisation d'un prototype permet d'utiliser la routine avant que le compilateur ne lise son corps.

Un prototype se déclare en C comme une routine absente de code :

```
int addition(int a, int b); // Prototype de la fonction 'addition'

// potentielles utilisations de la fonction 'addition'

int addition(int a, int b) { // Corps de la fonction 'addition'
    return a + b;
}
```

**Attention !** Il ne faut pas oublier le point-virgule à la fin de la déclaration d'un prototype.

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  // Chaque ligne ci-dessous montre l'existence de la fonction 'addition' au compilateur
5  int addition(int a, int b); // Prototype de la fonction 'addition'
6  // OU
7  int addition(int, int); // Les noms des paramètres sont optionnels dans le 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) { // Déclaration de la fonction 'addition'
15     return a + b;
16 }
```

## 6.6.2 Note relative à la copie des arguments

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

32. On pourra se demander pourquoi d'autres informations que le nom seraient nécessaires. En C++ et dans d'autres langages que le C, il est possible de *surcharger* des routines en utilisant le même nom mais une signature différente. Le compilateur peut toujours identifier l'appel et cela permet de simplifier l'interface pour le programmeur.

```

1  #include <stdio.h>
2  #include <stdlib.h>
3
4  void proc(int c) {
5      v = 48; // ERREUR : v non déclarée dans cette fonction
6      return c;
7  }
8
9  int main() {
10     int v = 5;
11     proc(v+1);
12     v = c; // ERREUR : c non déclarée dans cette fonction
13     return EXIT_SUCCESS;
14 }

```

En particulier :

```

1  #include <stdlib.h>
2
3  int polynome(char p) { // Déclaration de la fonction 'addition'
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 34 : 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) { // Déclaration de la fonction 'addition'
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) { // Déclaration de la fonction 'addition'
4      p = (p*p - p + 1);
5      return p; // projection de type implicite
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...

### 6.6.3 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.

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.

Prenons comme exemple le code suivant :

```

1  #include <stdio.h>
2  #include <stdlib.h>
3
4  #define X ... // valeur quelconque
5  #define Y ... // valeur quelconque
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 }

```

L'appel à `multiply` *empile* l'adresse mémoire de l'instruction suivant l'appel à la fonction, empile les arguments sur la pile puis saute à l'adresse mémoire de la fonction `multiply`, qui est ensuite exécutée. Cela est 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 *dépilement* de la pile d'exécution).

#### Petit aparté : À propos de l'allocation de la mémoire

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 bugs.

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.

**Note :** le dépilement de la pile d'exécution ne dépend pas de sa taille. Il ne s'agit que d'un décalage de l'adresse de haut de pile. On peut donc utiliser autant de variables que l'on veut sans craindre de perte de *vitesse* du programme.

Il est possible de gérer par soi-même l'allocation des variables en mémoire. C'est ce qu'on appelle l'*allocation dynamique* (voir section 6.10). La pile n'est jamais <sup>a</sup> utilisée pour ce type d'allocation.

a. à deux trois détails près...

**Remarque 1 :** on minimise le stockage de variables sur la pile grâce à des blocs de code

**Remarque 2 :** si un calcul n'utilise que peu de variables temporaires, il n'est pas intéressant de compartimenter le programme en trop de routines car ces routines utiliseront la mémoire pour stocker les paramètres et pour stocker l'appel de fonction. Un exemple<sup>33</sup> est le suivant :

```

1 | int sum(int a, int b, int c, int d, int e, int f, int g, int h, int i) {
2 |     return a + b + c + d + e + f + g + h + i;
3 | }
4 |
5 | ...
6 |
7 | int result = sum(1, 2, 3, 4, 5, 6, 7, 8, 9);

```

Par ailleurs, l'appel d'une routine prend du temps, puisqu'il faut empiler les arguments, allouer de la mémoire pour les variables temporaires sur la pile d'exécution, etc... L'utilisation de nombreuses routines peut être utile pour structurer le code, mais seulement si ces routines ont un sens en tant que tel. Il est par exemple étrange de programmer une routine qui est si spécifique qu'elle ne sera appelée qu'une seule fois dans tout un programme.

Pour les raisons précitées, on peut penser à minimiser le nombre de routines dans un programme pour optimiser sa vitesse. Cependant, le compilateur s'occupe généralement de "déroutiniser"<sup>34</sup> les routines de peu de lignes. En général, le gain obtenu par un programme bien pensé et bien structuré est donc très largement supérieur à celui d'un programme absent de routines.

### 6.6.4 Effet de bord

Une fonction peut agir sur son environnement, c'est-à-dire sur le résultat de l'exécution d'un programme, grâce à sa valeur de retour. Toutefois, une procédure ne le peut pas. Cela ne signifie pour autant pas qu'une procédure ne puisse pas agir sur son environnement. Ainsi, une procédure peut, sans renvoyer de valeur, agir sur les valeurs d'adresses mémoires extérieures à sa région propre de code. Par exemple :

```

1 | void print_number(int x) {
2 |     printf("%d\n", x);
3 | }

```

est une procédure agissant sur son environnement. Elle interagit avec la console sur laquelle s'exécute le programme<sup>35</sup>.

Une action d'une routine différente d'un renvoi de valeur hors de son environnement local est nommée un *effet de bord*.

33. Extrême et peu réaliste, certes

34. Le compilateur, lorsque les drapeaux d'optimisation sont activés, peut s'occuper de copier le code des routines directement dans le code principal, sans véritable appel de routines, pour aller plus vite et ne plus avoir ce désavantage majeur de l'utilisation des routines. Toutefois, il ne peut optimiser que munit de la certitude de ne pas modifier la sémantique du programme. Cela réduit grandement son champ d'action pour l'optimisation. Il est recommandé d'écrire par soi-même un code optimisé ^^

35. Voir la section 6.8 sur les flux standards pour plus de détails.

### 6.6.5 Exercices

**Exercice 28 (Encore Fibonacci) [11].**

Écrire une fonction de signature `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 29 (Puissances entières) [10].** Écrire une routine équivalente à la fonction mathématique :

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

Dans quelle mesure cette fonction est-elle correcte ?

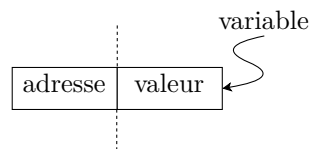


Les pointeurs sont souvent considérés comme la source de bugs la plus fréquente en C. Cela résulte en général d'une mauvaise compréhension de leur manipulation et de manques de rigueur dans les codes utilisant les pointeurs. Ce chapitre espère être suffisamment précis pour éviter au mieux les écueils habituels de la programmation avec des pointeurs... mais il n'est absolument pas certain que cela soit réussi.

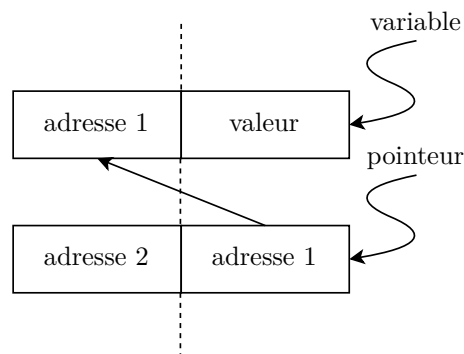
**Rappel :** Une variable peut être approximée comme la collection de trois informations :

- 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 C<sup>36</sup>. En particulier, ils permettent une flexibilité très importante des programmes écrits. Ainsi, une variable d'une routine est locale à cette routine, et n'existe pas en dehors. Pourtant, une adresse mémoire peut potentiellement être utilisée hors de routines spécifiques, et l'utilisation de pointeurs permet de manipuler des variables bien qu'étant hors de l'environnement local de cette variable.

L'utilisation des pointeurs pour manipuler des adresses non locales décuple les possibilités des routines 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 : si les données d'un environnement local sont accessibles hors de celui-ci, il devient facilement possible d'écrire des bugs. Il s'agit de la source d'erreurs la plus classique en programmation en C. Cette erreur est traduite par le fameux message d'erreur *Segmentation Fault*, c'est-à-dire littéralement : accès non autorisé (écriture ou exécution) à un segment de mémoire.

Un pointeur est défini selon la syntaxe suivante :

```
| TYPE* ptr;
```

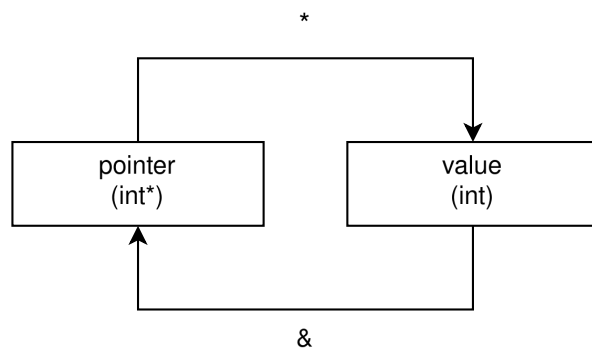
On sait alors que *ptr* contiendra l'adresse d'une variable de type *TYPE* et va alors *référer* à cette variable.

### 6.7.1 Pointeurs et références

On dispose pour la manipulation des pointeurs de deux opérateurs en langage C : l'étoile *\** et l'esperluette *&*.

L'esperluette en tant qu'opérateur binaire permet de signifier l'opération *ET*, qu'elle soit logique ou bit-à-bit. Comme opérateur unaire, *&v* renvoie l'adresse mémoire à laquelle se situe la variable *v*.

L'étoile en tant qu'opérateur binaire permet de signifier la multiplication numérique. Comme opérateur unaire, *\*ptr* est une référence de la variable pointée par *ptr*. Il s'agit d'accéder à la case mémoire contenue à une certaine adresse.



On peut donc écrire :

```
| double v = ...;
| double* ptr = &v; // référence 'v' dans 'ptr'
```

36. Et pas seulement en C...

```
// %p est le caractère de formatage d'une adresse mémoire :
printf("Adresse de v = %p = %p\n", &v, ptr);
printf("Valeur de v = %lf = %lf\n", v, *ptr);
```

Ainsi :

- & est l'opérateur de *référencement* qui permet de contruire la référence d'une variable
- \* est l'opérateur de *déréférencement*<sup>37</sup> qui permet d'accéder effectivement à la référence de la variable pointé

Le référencement permet de modifier la variable indirectement, *via* le pointeur :

```
int v = 42; // déclaration et initialisation d'une variable
int* ptr = &v; // construire d'une "référence" vers 'v', &v est l'adresse de v
*ptr = 1001; // <=> v = 1001 par déréférencement de ptr
```

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 son type est stockée comme le type du pointeur.

On peut décider également qu'un pointeur ne pointe sur rien grâce à la constante `NULL = 0` :

```
int *pointeur = NULL; // ne pointe sur rien
```

**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; // pointe sur rien
char *pointeur2 = (char *)0x12345678; // a priori une adresse invalide

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

---

37. Ou d'indirection

**Petit aparté : Segments d'un programme**

Un programme informatique d'ordinateur est généralement composé de plusieurs segments en mémoire dont les significations diffèrent.

On distingue principalement :

- les segments de code : contiennent les instructions exécutables du programme
- les segments de donnée : contiennent les variables globales du programme <sup>a</sup>
- le segment de pile : contient la pile d'exécution

Une erreur de segmentation provient principalement de la tentative d'accès à un segment de manière non autorisée. Il peut s'agir d'un segment du programme propre ou d'un autre programme. Dans tous les cas, cela est représentatif de l'accès non autorisé par le système d'exploitation ou par le processeur à une adresse mémoire de la RAM.

Il faut alors vérifier que chaque utilisation de pointeur est valide dans la zone du code d'où provient l'erreur.

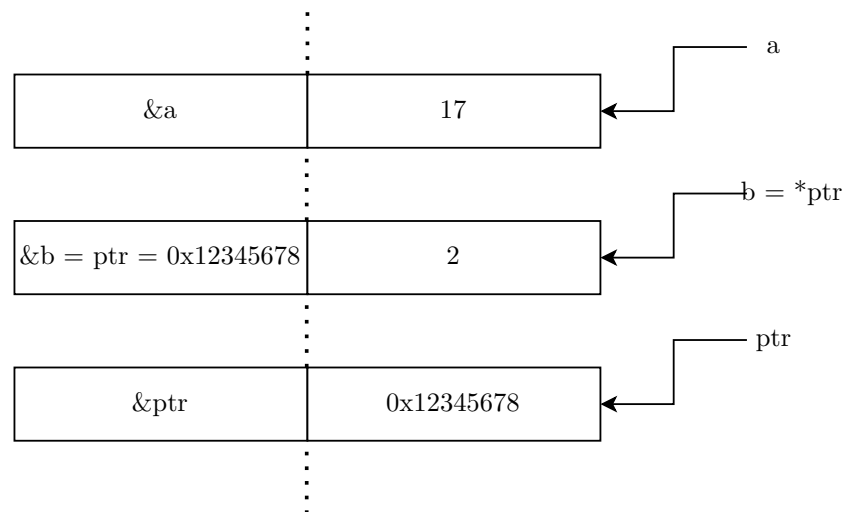
---

a. À peu près, voir les **Classes de stockage** (section 7.6) pour plus de détails.

L'opérateur \*, lorsqu'il est utilisé comme opérateur unaire, change lui aussi 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; // référence vers a
8      printf("Origine : a = ", *ptr);
9      *ptr = 17; // pareil que a = 17
10     printf("a = %d\n", a);
11     ptr = &b; // référence vers b
12     printf("Origine : b = ", *ptr);
13     *ptr = 2; // pareil que b = 2
14     printf("b = %d", b);
15     return EXIT_SUCCESS;
16 }
```

Après exécution du code, on a le schéma suivant :



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>38</sup>.

### 6.7.2 Détails sur les références : *lvalue* et *rvalue*

Les codes ci-dessus peut avoir provoqué chez le lecteur un froncement de sourcils, en particulier si celui-ci est pointilleux sur la rigueur et les détails...

Il pourrait après avoir essayé ceci avec succès :

```
int a = 5;
int b = 6;
*a = b;
```

s'être dit que `*a` est exactement la valeur de `b`. Chose étrange puisqu'il est impossible d'écrire `8 = a`. Et de tenter :

```
int a = 5;
int b = 6;
&a = &b;
```

et lire alors l'erreur de compilation « *lvalue required as left operand of assignment* ».

Et de n'y rien comprendre !

Pourquoi serait-il impossible de modifier l'adresse du symbole `a` ? Certes, le programme ne pourrait plus accéder à la case mémoire d'origine, mais l'instruction en elle-même ne semble pas absurde de prime abord. Et qu'est-ce qu'une "*lvalue*" ? Et pourquoi, alors que `a = 5`, écrire `*a = b` est valide, mais `5 = b` ne l'est pas ?

Posons des mots sur l'intuition.

Commençons par expliquer pourquoi ce dernier code ne peut être valide. L'assignation modifiant l'adresse de `a` effectue en vérité un traitement symbolique. `a` est une étiquette sur une adresse mémoire.

38. En toute originalité.

Vouloir changer l'adresse mémoire de cette étiquette, ce n'est pas ordonner à l'ordinateur d'agir mais uniquement de modifier la manière dont *le compilateur* lit le programme. Et le C ne permet pas de traitement symbolique par l'assignation.

Revenons au cas de l'assignation de valeur par un pointeur. Il existe en langage C deux catégories de valeurs :

- les *lvalues*<sup>39</sup> : symbole qui permet d'évaluer l'*identité* d'un objet, d'un texte binaire
- les *rvalues*<sup>40</sup> : symbole qui permet d'évaluer la *valeur* d'un opérande, le texte binaire lui-même

On différencie l'identification d'un texte binaire avec ce texte binaire lui-même. On comprend alors le code suivant :

```
1  int a;
2  int b = 6;
3  int* p = &a;
4  *p = 5;
```

Ici, *\*p* identifie le mot binaire qui est valeur de *a*. Il n'est pas lui-même ce mot binaire. Alors que *&a* est soi-même le mot binaire adresse de *a* en mémoire.

*\*p* est donc bien une *référence* vers *a*, alors que *&a* n'est pas une référence mais permet d'en construire une.

### 6.7.3 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 doive 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 }
```

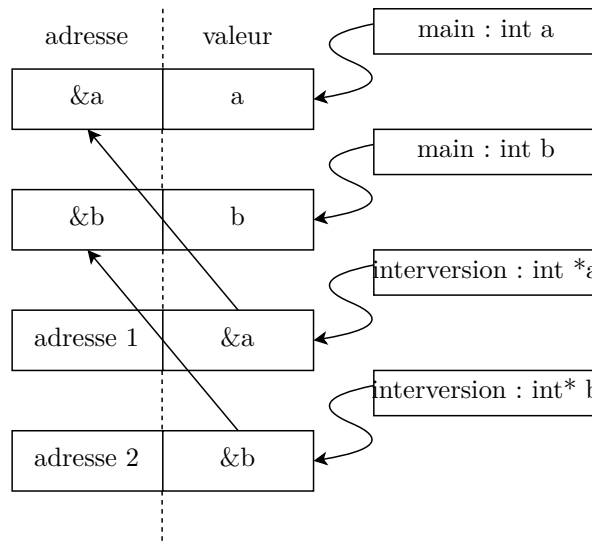
39. Abbréviation de *left value*, c'est-à-dire *valeur apparaissant à gauche dans une expression*

40. Abbréviation de *right value*, c'est-à-dire *valeur apparaissant à droite dans une expression*



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 `intversion(&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 14**.

### 6.7.4 Arithmétique des pointeurs et projection de type

Un point important n'a pas été abordé sur les pointeurs<sup>41</sup> : l'arithmétique des pointeurs, fortement liée à la projection de type sur les pointeurs.

Considérons le code suivant :

```

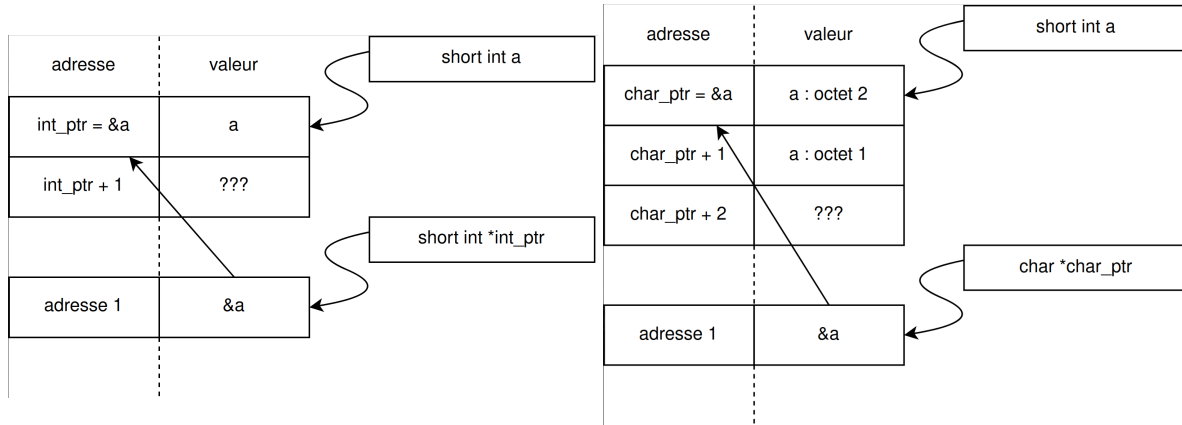
1  #include <stdio.h>
2
3  int main() {
4      short int a = 7187;
5      short int *int_ptr = &a;
6      printf("%d\n", *int_ptr);
7      char *char_ptr = (char *)int_ptr; // projection de type explicite
8      printf("%d\n", *char_ptr);
9      printf("%d\n", *(char_ptr + 1));
10 }
```

On observe que :

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

41. Sans jeux de mots aucun... puisque les jeux de mots laids font des gambettes ^^

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 :



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;

// tous les tests d'égalité suivants sont vrais :
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;
```

Et de manière plus générale pour tout TYPE et pour tout  $n$  :

```
TYPE* ptr = char_ptr;
ptr + n == char_ptr + sizeof(TYPE) * n
```

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>42</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 // le résultat du test suivant est toujours vrai
2 ptr[i] == *(ptr + i);
```

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

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

```
1 // le résultat du test suivant est toujours vrai
2 (i - 1)[ptr + 1] == *(ptr + i);
```

Les raisons de l'existence de cette facilité d'écriture seront détaillées dans la section 6.9 sur les tableaux.

### 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 6.10 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 :

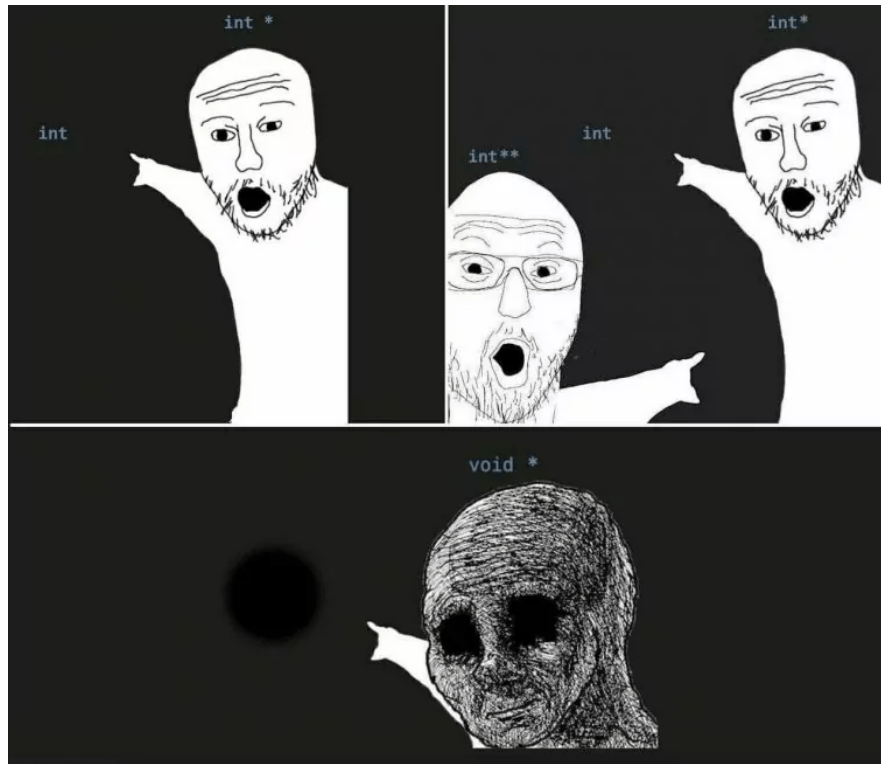
```
1 void *pointeur_quelconque = NULL;
2
3 // projection de type explicite pour éviter les avertissements à la compilation
4 int *int_ptr = (int*)pointeur_quelconque;
```

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

```
1 void *pointeur_quelconque = NULL;
2 printf("%p\n", pointeur_quelconque + 5); // -> 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 6.10 sur les tableaux dynamiques qui montre une application essentielle des pointeurs.

### 6.7.5 Pointeurs itérés



On laisse au lecteur le plaisir d'essayer, ce qui lui permettra de se familiariser avec les concepts.

### 6.7.6 Exercices

**Exercice 30 (Quelques procédures inutiles pour devenir un bot efficace) [10].** 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 31 (Interversion sans effet de bord (3)) [10].** 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 passée en argument pour les deux paramètres)

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

**Exercice 32 (Distance de Manhattan) [25].** Cet exercice est là uniquement comme exercice technique<sup>43</sup>. La méthode utilisée est à la fois inefficace et immonde à lire et à programmer. On se rapportera aux **Structures** pour une implantation correcte.

43. et rigolo (?)

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>44</sup>

**Remarque :** On pourra éviter les avertissements à la compilation en effectuant des projections de type explicites.

**Exercice 33 (Valeur absolue (2)) [18].** En utilisant seulement des projections de type et une opération bits-à-bits, écrire une fonction `float abs(float x)` ; qui calcule la valeur absolue de  $x$ .

**Exercice 34 (Racine carrée inverse rapide (2)) [25].** Reprendre le résultat de l' **Exercice 10** pour écrire une fonction `float fast_inverse_square_root(float x)` ; qui calcule  $\frac{1}{\sqrt{x}}$  par une simple transformation linéaire.

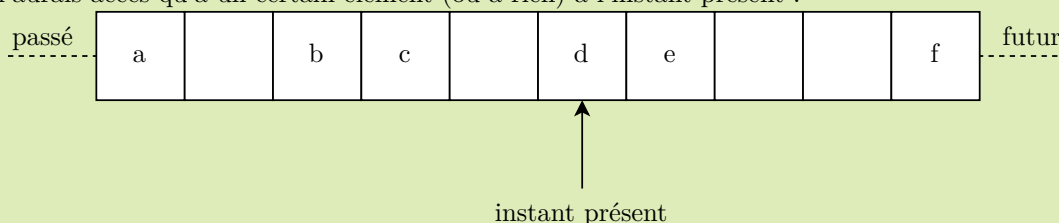


## INTERAGIR AVEC LES FLUX STANDARDS



### Définition 35 : 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 vues 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*

44. Souvent utilisée en informatique du fait de sa vitesse d'exécution.

- 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 la manipulation dans beaucoup de cas.

En particulier, trois flux *textuels* sont créés par défaut au moment de l'inclusion du module `stdio.h` :

- le flux de sortie standard représenté en C par `stdout`
- le flux d'entrée standard représenté en C par `stdin`
- 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>45</sup>

### 6.8.1 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); // appuyer sur "Entrée" pour valider la saisie
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` :

45. Dont les fonctionnements techniques ne reposent pas tout à fait sur le concept de flux ceci étant dit...

```
int x, y;
printf("Entrer deux nombres : ");
int counter = scanf("%d %d", &x, &y); // appuyer sur "Entrée" pour valider la saisie
printf("%d valeurs saisies.", counter);
printf("Les deux nombres saisis sont %d et %d", x, y);
```

## 6.8.2 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 :

```
fprintf(stdout, "Hello World !");
while (1);
```

```
fprintf(stderr, "Petite erreur !");
while (1);
```

```
> ./stdout
```

```
> ./stderr
Petite erreur !
```

Le texte envoyé sur `stderr` s'affiche alors que le texte envoyé sur `stdout` non. La question, simple mais pas si bête, est la suivante : *pourquoi ?*

La réponse, quant-à-elle, n'est pas particulièrement compliqué, mais tout de même un peu plus 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 essaye 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>46</sup>.

### 6.8.3 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) // appuyer sur "Entrée" pour valider la saisie
8      {
9          fprintf(stderr, "Erreur saisie incorrecte.\n"); // \n seulement pour lisibilité
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 }

```

### 6.8.4 Exercices

Dans la suite,  $N$  désigne une constante définie par un `#define`.

46. [https://fr.wikipedia.org/wiki/Historique\\_\(informatique\)](https://fr.wikipedia.org/wiki/Historique_(informatique))



**Exercice 35 (Distance Euclidienne) [11].** 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 36 (Somme d'entiers) [15].** 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.

**Exercice 37 (Jeu du plus ou moins) [20].** L'objectif est de programmer un petit jeu vidéo dans une console : le plus ou moins ! Le principe est très simple : un nombre mystère est tiré au hasard entre 0 et  $N - 1$  inclu au lancement du programme. Le joueur doit deviner en le moins de tentatives possibles ce nombre mystère. À chaque tentative, le programme lui dira si le nombre mystère est plus grand ou plus petit. On pourra compter le nombre de tentatives, et par exemple l'afficher à la fin. Combien de tentatives au maximum est-il nécessaire pour trouver le nombre mystère à coup sûr ?

```
user@computer ~/working_directory> ./main
Un nombre mystere a ete tire entre 0 et 99 inclu !
Quel est le nombre mystere ?
> 74
Le nombre mystere est plus petit
> 34
Le nombre mystere est plus grand
> 51
Youpiiii tu as trouve le nombre mystere en 3 tentatives !
user@computer ~/working_directory>
```

**Génération de nombres pseudo-aléatoires :** Un ordinateur ne peut pas en soi générer des nombres aléatoires puisqu'il est absolument déterministe. La seule chose qui est possible est d'utiliser des suites à comportements chaotiques dont la valeur initiale est différente à chaque exécution du programme. On utilise en général la date à laquelle le programme est exécuté, c'est-à-dire le nombre de secondes écoulées depuis le 1<sup>er</sup> janvier 1970. On peut utiliser pour cela le module `time.h` du C :

```
1  #include <time.h>
2
3  int main() {
4      time_t seconds = time(NULL);
5      printf("Nombre de secondes depuis le 1er janvier 1970 : %ld\n", seconds);
6  }
```

On peut alors utiliser les fonctions `void srand(unsigned int seed);` et `void rand();` du module `stdlib.h` qui respectivement :

- donne une valeur initiale à la suite du générateur de valeurs pseudo-aléatoires (i.e. chaotiques)
- renvoie une valeur pseudo-aléatoire<sup>47</sup> entre 0 et `RAND_MAX`<sup>48</sup> inclu.

Dans la pratique :

```
1  #include <stdlib.h>
2  #include <time.h>
3
4  int main() {
```

47. C'est-à-dire le terme suivant de la suite chaotique.

48. Constante définie dans `stdlib.h`

```

5 | srand(time(NULL));
6 | ...
7 | }

```



## TABLEAUX STATIQUES

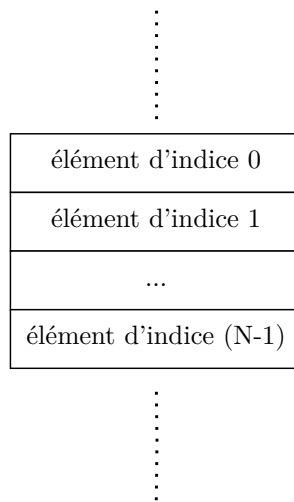


### 6.9.1 Tableaux

#### Définition 36 : 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.

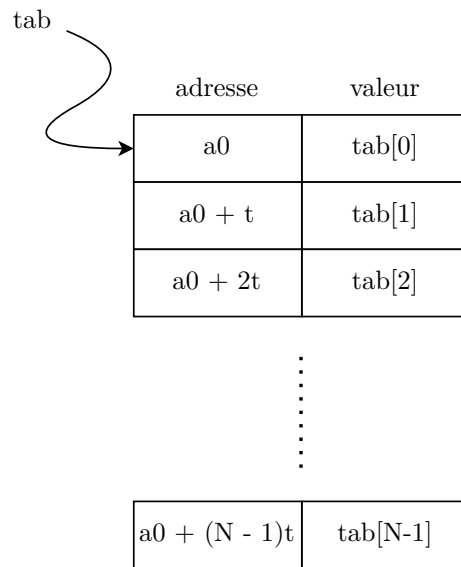
Le *tableau* est une des structures de donnée les plus simples. Il s'agit de l'équivalent informatique des  $N$ -uplets en mathématiques. Un tableau permet de stocker plusieurs variables sous une même étiquette de manière contigüe en mémoire :



Un tableau est dit *indiqué*, 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 sur la pile d'exécution<sup>49</sup> (voir 4.6.3 pour un rappel)
- Par allocation dynamique : allocation par le programmeur un tableau dont la taille dépend d'une variable du programme, sur le tas

Cette section ne décrit que l'allocation statique.

### 6.9.2 Définition

L'allocation sur la pile, dite statique, est effectuée en langage C par la syntaxe suivante :

```
TYPE array[SIZE]; // tableau non initialisé
TYPE array[SIZE] = {v1, v2, ..., vSIZE}; // tableau initialisé
```

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. En effet, la taille d'un tableau statique doit pouvoir être déterminé par le compilateur au moment de la compilation. Celui-ci peut ainsi produire des instructions machines pour stocker ce tableau sur la pile d'exécution comme *n* variables de tailles déterminées.

Certaines versions de certains compilateurs plus récents peuvent cependant accepter cette syntaxe et remplacer le code par une *allocation dynamique* (voir section suivante 6.10). Pour des raisons de compatibilité et pour éviter certains bugs, il est expressément recommandé d'éviter cette pratique douteuse.

49. Le plus souvent. Cela dépend de la *classe de stockage* utilisé, voir le chapitre correspondant.

Il est par contre tout à fait possible de préciser la taille d'un tableau par un symbole définie par une directive pré-processeur :

```
#define N 5
int array[N]; // à la compilation, chaque N sera remplacé par le symbole 5 dans le code
```

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] = {...}; // valeurs quelconques
    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]; // déclaration non initialisée  // ERREUR : array = {3141526535, 0xBEEF, 747, 0xC4, 713705, 666, 1414};  // ERREUR à nouveau : array[7] = {3141526535, 0xBEEF, 747, 0xC4, 713705, 666, 1414};  // Python n'est pas C : array = [i for i in range(7)]; // ERREUR !!!</pre>	<pre>long int array[7];  // pas d'erreurs : array[0] = 3141526535; array[1] = 0xBEEF; array[2] = 747; array[3] = 0xC4; array[4] = 713705; array[5] = 666; array[6] = 1414;</pre>
---	--

La raison, qui « expliquera » au passage l'utilisation des accolades  $\{\dots\}$  sera vue dans la section 7.8 sur la construction de littéraux dans le chapitre 7 sur les concepts avancés du langage.

### 6.9.3 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>50</sup>. On appelle, en programmation, une entité de première classe une entité informatique régit par les mêmes règles générales que les autres entités fondamentales du langage. Il peut s'agir par exemple de pouvoir :

- être créé à l'exécution du programme
- être renvoyé par une fonction
- être assigné à une variable
- être passé en argument à une fonction

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 plus génériques 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>51</sup>

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>52</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.

50. 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)

51. L'appellation n'a rien d'officiel, mais je trouvais qu'elle décrivait bien l'état de fait.

52. Si ça vous paraît tordu... À moi aussi pour être honnête.

## 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 + indice);
```

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'est pas utilisée car assez illisible.<sup>53</sup>

## Passage d'un tableau en argument à une routine

Le langage C autorise à passer un tableau en argument à une routine (voir **Exercice 39**). 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[]) { // donne l'impression d'être un tableau
5      printf("%ld", sizeof(array));
```

---

53. Question d'habitude...

```

6   }
7
8   int main() {
9       int test[5] = {1, 2, 3, 4, 5};
10
11       printf("%ld", sizeof(test)); // -> sizeof(int) * 5 = 20
12       procedure(test); // -> 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'incrément 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     // Déclaré comme un pointeur car test_function renvoie un pointeur :
11     char *ret_array = test_function();
12
13     /* la ligne suivante produit une erreur À LA COMPILATION :
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!<sup>54</sup>

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 !<sup>55</sup>

### 6.9.4 Exercices

**Exercice 38 (Les routines, direction la seconde classe !)** [05]. Justifier que les routines en C ne sont pas non plus des entités de première classe.

**Exercice 39 (Routines classiques de manipulation de tableaux)** [20]. Soit  $T$  un tableau de  $l(T) \in \mathbb{N}$  éléments. On note pour  $i, j \in \llbracket 0; l(T) - 1 \rrbracket$  :

- $T[i]$  : l'élément de  $T$  d'indice  $i$
- $T[i : j]$  : le sous-tableau de  $T$  d'adresse  $T + i$  et de  $l(T[i : j]) = j - i$  éléments, c'est-à-dire que  $T[0 : l(T)] = T$ .
- Pour  $k \in \llbracket 0; l(T[i : j]) - 1 \rrbracket$ , on a donc  $T[i : j][k] = T[i + k]$

1. écrire une fonction `int somme(int array[], unsigned int length)`; qui renvoie la somme des éléments du tableau.

54. C'est-à-dire que la région de la pile utilisée pour l'environnement local de la fonction est vidée pour laisser la place à d'autres environnements locaux d'autres fonctions.

55. Pour celles et ceux qui se demandent : oui, il y a une suite à cette phrase, mais ceci est une autre histoire...



2. écrire une procédure `void display(int array[], unsigned int length)`; qui affiche les éléments d'un tableau d'entiers.  
*Note : cela pourra être utile pour vérifier le résultat de routines agissant sur des tableaux*
3. écrire deux fonctions `unsigned int max(int array[], unsigned int length)`; et `unsigned int min(int tab[], int taille)`; qui renvoient respectivement l'indice du maximum et l'indice du minimum des éléments d'un tableau.
4. écrire une fonction `void swap(int array[], int i, int j)`; qui échange par la méthode de votre choix les valeurs du tableau d'indices  $i$  et  $j$ .
5. en utilisant les questions précédentes, écrire une procédure `void selection_sort(int array[], unsigned int length)`; qui trie dans l'ordre croissant et par effet de bord un tableau  $A$  de  $n \in \mathbb{N}$  éléments selon l'algorithme ci-dessous.

---

**Algorithme 7 : Algorithme de tri de tableau par sélection**


---

**Entrées :** tableau  $A$  et  $l(A)$  le nombre d'éléments de  $A$

```

1 ;
  Sorties : tableau  $A$  trié par effet de bord
2 ;
3  $i \leftarrow 0$ ;
4 tant que  $i < l(A)$  faire
5   Trouver l'indice  $i_{min}$  du minimum de  $A[i : l(A)]$ ;
6   Échanger dans  $A$  les éléments d'indices  $i$  et  $i_{min} + i$ ;
7    $i \leftarrow i + 1$ ;

```

---

6. démontrer par récurrence la correction de l'algorithme précédent, c'est-à-dire une démonstration que cet algorithme trie effectivement le tableau dans l'ordre croissant.

**Exercice 40 (Recherche dichotomique) [12M].** On reprend les notations introduites dans l'**Exercice 39**.

1. Soit  $T$  un tableau d'entiers trié par ordre croissant. Soit  $i \in \llbracket 0; l(T) - 1 \rrbracket$ . Soit  $x \in \mathbb{Z}$ . Montrer que :

$$x \in T \Rightarrow \begin{cases} x \leq T[i] & \Rightarrow x \in T[0 : i] \\ x > T[i] & \Rightarrow x \in T[i : l(T)] \end{cases}$$

2. Écrire une fonction `char is_in(int x, int array[], unsigned int length)`; qui renvoie *Vrai* sur  $x$  appartient au tableau `array` supposé trié dans l'ordre croissant, et *Faux* sinon. Cette fonction devra au pire effectuer  $\lceil \log_2(length) \rceil$  tours de boucle. On le démontrera grâce à la question 1, en choisissant judicieusement  $i$  à chaque tour de boucle.

**Exercice 41 (Liste des nombres premiers) [37HM].** Cet exercice tient à montrer, dans un exemple particulier, l'importance fondamentale du détail, et comment ce qui semble une infime différence dans la manière d'écrire un code peut changer fondamentalement l'efficacité de l'algorithme mis en oeuvre.

1. En programmant une routine du test de primalité de l'**Exercice 27**, écrire un programme qui liste les nombres premiers inférieurs à  $N$  en au plus  $N^{\frac{3}{2}}$  tours de boucles.
2. En utilisant un tableau de taille  $N$  qui stocke les nombres premiers déjà trouvés, utiliser cette fois-ci un test de primalité qui teste si  $n \in \mathbb{N}$  est premier en au plus  $\pi(\sqrt{n})$  tours de boucle
3. Programmer l'algorithme du *crible d'Eratosthène* décrit ci-dessous.
4. Démontrer que l'algorithme de la **Question 2.** est asymptotiquement plus lent que le crible d'Eratosthène

*Note :* On pourra utiliser librement les résultats mathématiques non spécifiques au problème, comme les théorèmes de théorie des nombres ou certains résultats de calcul intégral.

**Algorithme 8 :** Algorithme du crible d'Eratosthène**Entrées :**  $N > 0$  un entier naturel**Sorties :** les entiers naturels inférieurs à  $N$ 

```

1 Déclarer un tableau  $P$  de  $N$  booléens;
2 Initialiser :  $P[0] = \text{Faux}$  et  $\forall i \in \llbracket 1; N-1 \rrbracket, P[i] = \text{Vrai}$ ;
3 On pose  $p := 2$ ;
4 tant que  $p^2 \leq N$  faire
5   pour ( $f := p$  à  $\left\lceil \frac{N}{p} \right\rceil - 1$ ) faire
6      $P[pf] = \text{Faux}$ ;
7   tant que  $P[p] = \text{Faux}$  faire
8      $p \leftarrow p + 1$ ;

```



## ALLOCATION DYNAMIQUE

**6.10.1 Introduction au tas**

Un programme informatique, lorsqu'il est chargé en mémoire, comporte un troisième espace mémoire<sup>56</sup> nommé *le tas* (*heap* en anglais). Il se nomme ainsi car il correspond à un tas de mémoire accessible au programmeur.<sup>57</sup>

Le tas est un espace mémoire de taille extensible<sup>58</sup> contrairement à la pile d'exécution. Il peut être agrandi au fur-et-à-mesure des allocations, c'est-à-dire lorsque trop de variables ont été alloués. Ainsi, il est virtuellement possible d'utiliser l'entièreté de la RAM avec le tas. Il possède cependant certains défauts :

- les variables alloués 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 bugs 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>59</sup> Par ailleurs, libérer incorrectement la mémoire peut amener à des failles de sécurité importantes.

56. Les deux premiers sont :

- le code
- la pile d'exécution

57. 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

58. Avec comme limite la taille de la RAM de l'ordinateur.

59. 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 !

Un autre bug 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>60</sup> Ce bug est rencontré en quasi-permanence chez les débutants, dès lors que des structures complexes sont manipulées en mémoire. Il arrive aussi très fréquemment<sup>61</sup> chez les programmeurs expérimentés. En effet, il est d'une facilité extraordinaire de rencontrer ce bug dès lors que la programmation manque un tout soit peu de rigueur.<sup>62</sup>

## 6.10.2 Les routines `malloc` et `free`

Pour gérer “soi-même”<sup>63</sup> de la mémoire sur le tas, deux fonctions de la bibliothèque `stdlib.h` fournissent les fonctionnalités suivantes :

- allouer un espace mémoire, grâce à `malloc`<sup>64</sup>
- libérer un espace mémoire précédemment alloué, grâce à `free`

Le détail du fonctionnement interne des deux fonctions ne sera pas explicité.<sup>65</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 fait, la taille de chaque bloc mémoire alloué dynamiquement est stocké en interne.

En particulier, tout accès à cet espace mémoire à la suite de la libération 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> // EXIT_SUCCESS, malloc et free
3
4 int main() {
5     int* array = (int*)malloc(sizeof(int) * 5); // Alloue 4 * 5 = 20 octets.
```

60. Si il était possible d'accéder à n'importe quelle case mémoire d'un ordinateur sans protection, le *hacking* serait presque trivial.

61. Quoi que ceux-ci en disent ;)

62. 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 la confrontation directe avec le “mal”

63. C'est les routines qui font le gros du boulot hein ! Faudrait pas se mettre à voler par les chevilles x)

64. Pour *Memory Allocation*

65. 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/>

```

6
7     for (int i = 0; i < 5; i++) {
8         printf("%d\n", *(array + i));
9     }
10
11     free(array); // Autorise de futures allocations à cette adresse
12     array = NULL; // "oublie" l'ancienne position du tableau en mémoire
13     return EXIT_SUCCESS;
14 }

```

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); // Alloue 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> // EXIT_SUCCESS, malloc et free
3
4  int* make_array(unsigned int n) {
5      int *array = (int*)malloc(sizeof(int) * n); // Alloue (4 * n) octets
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); // tableau de 4 * 10 = 40 octets
14
15     for (unsigned int i = 0; i < 10; i++) {
16         printf("%d\n", array[i]);
17     }
18
19     free(array); // la mémoire doit toujours être libérée
20     array = NULL; // et le pointeur réinitialisé
21     return EXIT_SUCCESS;
22 }

```

### 6.10.3 L'importance de réinitialiser le pointeur après libération

Dans le code suivant :

```

void* ptr = malloc(N);
... // utilisation de l'espace alloué
free(ptr);
ptr = NULL;

```

la dernière instruction de réinitialisation du pointeur semble *a priori* inutile. Le pointeur n'est plus utilisé, la mémoire est libérée et pourra donc être à nouveau réutilisée lors d'une prochaine allocation. Tout semble parfait. Et pourtant, là est le problème.

La non réinitialisation d'un pointeur peut dans certains cas provoquer des erreurs de sécurité, de par le fonctionnement de la fonction `malloc`. En effet, un `free` va autoriser le bloc mémoire libéré à être alloué à nouveau. Mais les données à cet endroit sont toujours présentes et peuvent être réutilisées par mégarde. La fonction `malloc` va allouer la même zone mémoire à l'allocation suivante. Le pointeur qui a été libéré pointe toujours vers cette zone.

Le code suivant illustre le problème :

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  #define USER_SIZEOF_DATA_STRUCTURE ...
5  #define IS_ADMIN_OFFSET ... // value < USER_SIZEOF_DATA_STRUCTURE
6
7  int main() {
8      char *guest = NULL, *admin = NULL;
9
10     // les informations d'un premier utilisateur sont stockés dans un tableau de données
11     guest = malloc(USER_SIZEOF_DATA_STRUCTURE);
12     // un des octets stocke les droits d'utilisateur (1 si admin, 0 sinon)
13     guest[IS_ADMIN_OFFSET] = 0; // pas admin par défaut
14
15     // du code
16
17     // quelque part :
18     free(guest);
19     // guest pointe toujours vers sa zone mémoire !!!
20
21     // encore du code
22
23     // et puis autre part :
24     admin = malloc(USER_SIZEOF_DATA_STRUCTURE);
25     admin[IS_ADMIN_OFFSET] = 1;
26     // (guest == admin) est vrai ici !!!!!
27
28     // encore et toujours du code
29
30     if (guest == NULL || guest[IS_ADMIN_OFFSET] == 0) {
31         printf("Cette section nécessite les droits d'administrateur !\n");
32         return EXIT_FAILURE;
33     }
34
35     printf("Acces a la zone d'administration autorise\n");
36
37     // du code administrateur
38
39     free(admin);
40     return EXIT_SUCCESS;
41 }
```

L'exécution de ce programme affiche à l'écran :

```
user@computer ~/working_directory> ./main
Accès à la zone d'administration autorisé
```

La réinitialisation n'est cependant pas toujours *nécessaire*. Par exemple, en sortie de routine, toutes les variables et pointeurs de l'environnement local sont « oubliés ». Dans le code suivant :

```
1 void some_proc() {
2     // code
3     char* t = malloc(N);
4     // code
5     free(t);
6 }
```

l'accès au pointeur  $t$  n'est pas possible à la suite d'un appel à la routine. Un simple `free` suffit amplement.

En général, pour ne pas *oublier* une réinitialisation qui pourrait être importante, on définit une *macro*<sup>66</sup>.

### 6.10.4 Exercices

**Exercice 42 (Conversion en binaire) [12].** Écrire une fonction `char* bin_from_nat(int n)` ; qui renvoie la représentation binaire de  $n$  dans un tableau  $B$ . Ainsi, si  $n = (b_{31} \dots b_0)_2$ , alors pour tout  $0 \leq i < 32$ ,  $B[i] = b_i$ , où  $B[i]$  désigne l'élément d'indice  $i$  de  $B$ . Écrire une procédure d'affichage d'un tableau  $B$  void `bin_display(char* b)` ;. Cette procédure doit afficher le nombre binaire dans le sens de lecture humain, c'est-à-dire avec les bits de poids faibles à droite.

*Indication :* on pourra considérer la représentation binaire d'un nombre sous forme de polynôme de Horner (voir l'égalité 2.1)



## 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 :

$$T_N = \begin{bmatrix} t_{n_0} \\ \vdots \\ t_{n_N} \end{bmatrix} = \begin{bmatrix} t_0[0] & \dots & t_0[n_0] \\ t_1[0] & \dots & t_1[n_1] \\ \vdots & & \vdots \\ t_N[0] & \dots & t_N[n_N] \end{bmatrix} \in \mathbb{R}^S \text{ où } S = \sum_{k=0}^N (n_k + 1)$$

<sup>66</sup>. Voir section 7.10 sur les directives du préprocesseur

**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 :

$$\begin{bmatrix} 1 \\ 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 \\ 9 \end{bmatrix} \quad (6.1)$$

### 6.11.1 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 est la dimension du tableau
```

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]; // valeur en 3ème ligne et 4ème colonne du tableau
```

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

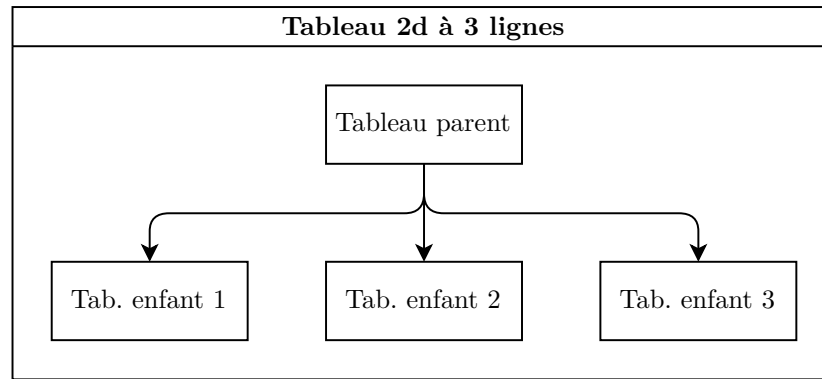
#### Limitations/Inconvénients des tableaux multidimensionnels statiques :

- 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.

### 6.11.2 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  $N_{lines}$  lignes et  $N_{columns}$  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 6.1, 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 // initialise le tableau avec des 0
13 for (int j = 0, j < (1 + min(5 - 1 - i, i)); j++) {
14     triangle[i][j] = 0;
15 }
16 }
17 free(triangle);
18 triangle = NULL; // pas obligatoire car sortie de routine
19 return EXIT_SUCCESS;
20 }
21

```

### 6.11.3 Exercices

**Exercice 43 (Afficher un tableau 2D) [10].** Écrire un programme (et pas une routine) qui affiche les éléments d'un tableau 2d d'entiers.

**Exercice 44 (Affichage du triangle) [13].** Compléter le programme ci-dessus pour remplir le tableau 6.1. 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)

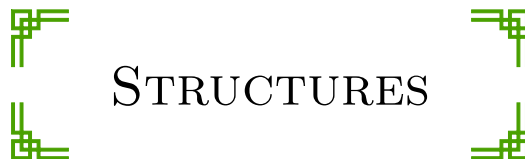
**Remarque :** L'exercice ci-contre met en évidence la possibilité de passer en argument des tableaux multidimensionnels.

**Exercice 45 (Matrices (1)) [13].** L'objet de cet exercice est l'implantation de matrices en langage C.

Soient  $n, m \in \mathbb{N}^*$ . Une matrice  $mat \in \mathcal{M}_{n,m}(\text{TYPE})$  est basiquement un tableau de lignes  $(L_i)_{i \in \llbracket 0;n \rrbracket}$  et chacune de ces lignes est un tableau de valeurs typées :

$$\begin{cases} mat &= (L_0, L_1, \dots, L_{n-1}) \\ \forall i \in \llbracket 0;n \rrbracket & L_i = (a_{i,0}, \dots, a_{i,m-1}) \end{cases}$$

1. Écrire une fonction `double** matrix_new(unsigned int n, unsigned int m);` qui alloue dynamiquement un tableau `double** mat` de  $n$  lignes de `double*` elles-mêmes alloués dynamiquement comme  $m$  cases de `double` et renvoie cette matrice.
2. Écrire une procédure `void matrix_destroy(double** mat, unsigned int n);` qui libère la mémoire associée à une matrice de  $n$  lignes. *Il faut que les  $n \times m$  cases soient libérées à la fin.*
3. En utilisant l'équivalence  $t[i] \equiv *((char*)t + i * sizeof(*t))$ , déterminer l'expression permettant d'accéder à la valeur  $mat_{i,j}$  de la matrice.
4. Écrire une procédure d'affichage d'une matrice de  $mat \in \mathcal{M}_{n,m}(\text{TYPE})$  `void matrix_display(double** mat, unsigned int n);`
5. Écrire une fonction `double** matrix_mul(double** a, double** b, unsigned int n, unsigned int k, unsigned int m);` qui effectue et renvoie le produit de deux matrices  $a \in \mathcal{M}_{n,k}$  et  $b \in \mathcal{M}_{k,m}$



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 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 :

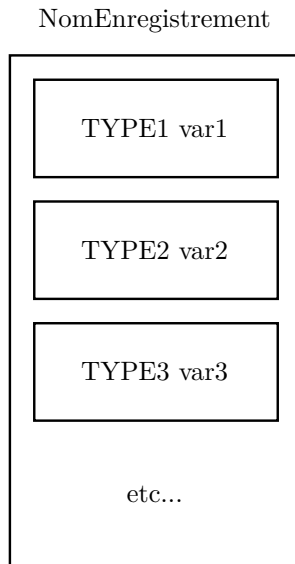
- la modélisation d'une lentille dans une simulation physique par une taille, la modélisation de sa forme par des courbes paramétriques, les propriétés du matériau, etc. . .
- le bouton d'une application par une taille, une position dans l'espace, une couleur, un texte, etc. . .
- un personnage d'un 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 noter dans une documentation que le 34<sup>e</sup> octet correspond à l'accélération selon l'axe  $z$  par exemple. Il faudrait ensuite penser à effectuer une projection de type en nombre flottant, puisqu'un tableau ne peut pas contenir des éléments de plusieurs types différents.

C'est possible. Mais cela semble plutôt complexe, et d'un sens pratique assez. . . comment dire. . . limité.

C'est ici que le type abstrait d'*enregistrement* entre en jeu.

Un enregistrement permet de définir une structure complexe fixe de stockage. C'est un moyen de stocker dans une seule entité plusieurs informations de types différents dont le nombre et la séquence sont fixées à la définition. Chacune de ces informations possèdera sa propre étiquette au sein de cette entité, ainsi que son propre type. C'est un peu comme une boîte qui contient plusieurs variables :



L'enregistrement est analogue aux produits cartésiens en mathématique<sup>67</sup>. Si les ensembles de chaque type sont  $E_1, \dots, E_n$ , l'ensemble des éléments que peut représenter un enregistrement est  $E_1 \times \dots \times E_n$ . Une *manifestation* (*instance* en anglais<sup>68</sup>) d'un enregistrement est un élément du produit cartésien,

67. Il arrive donc qu'on appelle les enregistrements des *types produits*. On peut les comparer avec les *types sommes* définis en C par les *unions* ou les *énumérations* (voir sections 7.4 et 7.3)

68. On notera que l'anglicisme *instance* est utilisé en lieu et place de *manifestation* dans la littérature en français. Je n'aime pas les anglicismes, donc y a le choix entre :

1. mettre en italique toutes les occurrences de *instance*

c'est-à-dire un  $n$ -uplet dont chaque *champ* appartient à un ensemble  $E_i$ .

On appelle aussi les enregistrements des structures de données passives. L'enregistrement est représenté par des champs de valeurs passifs.

On s'intéresse dans cette section à la programmation concrète d'un enregistrement et à la syntaxe C qui permet sa manipulation. On verra de manière beaucoup plus poussée dans le chapitre 9 l'application qui peut être faite de cet outil fourni par le langage. En C, les enregistrements sont implantés par le mot-clé `struct`.

#### Point vocabulaire : Structure ou enregistrement ?

Les enregistrements dans la littérature technique sur le langage C sont appelés simplement des structures, du fait du mot-clé utilisé pour les décrire. Le *Pascal* est un langage plus précis qui utilise le mot-clé *record*. En manipulation de bases de données, c'est aussi le mot "enregistrement" qui est utilisé.

La justification du mot "structure" est que les enregistrements peuvent être utilisés comme support de données pour implanter des structures de données plus complexes. Ils servent comme type d'abstraction inférieur (voir section 9.1) Il faut toutefois faire attention à la distinction entre une structure de données de manière générale (voir chapitre 9) et les structures du langage C qui ne sont qu'une manière d'implanter des produits cartésiens.

Dans la suite, on utilisera le mot "enregistrement" pour parler du type abstrait et le mot "structure" pour parler de l'implantation en C des enregistrements.

La syntaxe de définition d'un enregistrement en C est la suivante :

```
struct RecordName { // Aucun objet n'est créé, il s'agit d'une description
    TYPE1 var1;
    TYPE2 var2;
    TYPE3 var3;
    // etc...
};
```

Il s'agit d'une définition, et non d'une déclaration. *On définit un nouveau type.* Pour cette raison, aucune variable interne, ou *champ*, de l'enregistrement ne peut être initialisée. Pour utiliser un enregistrement, il faut au final en construire des *instances*. On rappelle qu'un enregistrement est la description d'un produit cartésien. Il faut donc déclarer une variable dont le type est cet enregistrement :

```
struct RecordName instance; // 'instance' est créé sans champs initialisés.
```

À tout instant de l'exécution du programme, `instance` représentera un tuple élément de ce produit cartésien.

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 observe qu'un type abstrait tel que défini dans la section 6.1 sur les variables est un ensemble muni d'opérations. L'utilisation d'enregistrements permet de définir des ensembles par produits cartésiens, ce qui facilite la définition de types abstraits. On peut par exemple implanter  $\mathbb{Q}$  comme l'ensemble  $\mathbb{Z} \times \mathbb{N}$ , pour définir les rationnels comme des rapports  $\frac{a}{b}$  :

2. utiliser le mot "manifestation"

Comme le but est ensuite de pouvoir lire plein d'autres livres, on adopte la première convention.

```
struct Rationnal {
    signed int num; // numérateur
    unsigned int den; // dénominateur
};
```

On peut ensuite accéder aux membres d’une instance d’une structure via un nouvel opérateur binaire, l’opérateur d’accès à un champ noté “.” :

```
instance.var1 = ...; // accède au champ 'var1' de 'instance'
instance.var2 = ...; // accède au champ 'var2' de 'instance'
instance.var3 = ...; // accède au champ 'var3' de 'instance'
// 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 }
```

Ici, Point3D est une implantation de  $\mathbb{R}_{f64}^3 = \mathbb{R}_{f64} \times \mathbb{R}_{f64} \times \mathbb{R}_{f64}$

**Remarque 1 :** Une structure en C ne peut contenir au maximum que 127 champs. Il s’agit d’une limitation technique à l’implantation des enregistrements.

**Remarque 2 :** Au contraire des tableaux statiques, toute instance de structure est considérée comme une entité de première classe. Ainsi, il est possible d’écrire des routines dont les paramètres sont des instances de structures, et qui prennent en argument des instances de 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 }
```

Il est immédiat qu'un ensemble d'un produit cartésien soit lui-même un produit cartésien. Ainsi,  $\mathbb{R} \times (\mathbb{N} \times \mathbb{N})$  est toujours un produit cartésien. De même en C, on peut définir un champ d'une structure comme étant 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 // ou encore :
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     // . est un opérateur associatif :
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 }
```

### 6.12.1 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 :

<pre> <b>struct</b> MyStruct {     <b>int</b> a;     <b>char</b> b;     <b>void</b> *c; // <i>pointeur quelconque</i>     <b>long double</b> d; };  // <i>Initialisation séquentielle</i> <b>struct</b> MyStruct s = {     2&lt;&lt;20,     6,     <b>NULL</b>,     3.141592653589 }; </pre>	<pre> <b>struct</b> MyStruct {     <b>int</b> a;     <b>char</b> b;     <b>void</b> *c; // <i>pointeur quelconque</i>     <b>long double</b> d; };  // <i>Initialisation sélective</i> <b>struct</b> MyStruct s = {     .d = 3.141592653589,     .a = 2&lt;&lt;20,     .c = <b>NULL</b>,     .b = 6 }; </pre>
--	---

L'initialisation séquentielle suit exactement l'ordre des champs de la structure.

L'initialisation sélective permet de choisir très exactement quel champ 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, .x = 3.2}; // p.x = 3.2; p.y = 4.5;

```

Il est parfaitement possible de mélanger les initialisations séquentielles et sélectives. Dans un tel cas, l'initialisation séquentielle reprend au dernier champ désigné par une initialisation sélective. Partant, le code suivant initialise le membre *a* à 1 et le membre *d* à 30.560.

```

struct MyRecord s = {1, .d = 30.560};

```

Alors que le code ci-dessous initialise le champ *c* à **NULL**, le champ *d* à 48, le champ *b* à 0 et le champ *c* à *&s*.

```

struct MyRecord s = {.c = NULL, 48, .b = 0, &s}; // d non initialisé

```

Par contre, le code suivant provoque une erreur à la compilation :

```

struct Point p = {.y = 4.5, 3.2}; // erreur, car il n'y a pas de champ avec 'y'

```

### 6.12.2 Mécanisme de construction

La question ici est de comprendre quel est le mécanisme qui permet d'accéder à chacun des champs d'une structure à partir de sa définition, c'est-à-dire comment fonctionne concrètement, techniquement, matériellement<sup>69</sup> l'opérateur d'indirection.

L'objectif substantiel est de montrer que l'utilisation de structures et l'utilisation de l'opérateur d'accès à un champ d'une instance ne ralentissent pas l'exécution d'un programme.

Pour mieux comprendre comment les données sont stockées, on exécute le programme suivant :

```

1  #include <stdio.h>
2  #include <stdlib.h>
3
4  struct Something {
5      int a;
6      short int b;
7      char c;
8  };
9
10 int main() {
11     struct Something s = {.a = 0x01234567, .b = 0x89AB, .c = 0xCD};
12
13     printf("Taille de la structure : %lu\n", sizeof(struct Something));
14
15     // permet d'accéder à la structure octet par octet :
16     unsigned char *ptr = (unsigned char*)&s;
17     for (unsigned int i = 0; i < sizeof(struct Something); i++) {
18         printf("octet %u : 0x%02x\n", i, ptr[i]);
19     }
20
21     return EXIT_SUCCESS;
22 }
```

À l'exécution, on observe le résultat suivant :

```

user@computer ~/working_directory> ./main
Taille de la structure : 8
# s.a :
octet 0 : 0x67
octet 1 : 0x45
octet 2 : 0x23
octet 3 : 0x01
# s.b :
octet 4 : 0xab
octet 5 : 0x89
# s.c :
octet 6 : 0xcd
# padding
octet 7 : 0x00
```

On observe que :

- les champs sont contenus les uns à la suite des autres, de manière compacte

69. Je rajoute des caisses d'adverbes pour le plaisir

- les octets sont inversés, par petit boutisme (<https://fr.wikipedia.org/wiki/Boutisme>)
- un octet est ajouté à la fin de la structure “*sans raisons apparentes*”

On discute du troisième point dans la sous-section 6.12.4 suivante.

La compacité des structures est logique dans un souci d’optimisation spatiale. Elle signifie en particulier que les champs d’une structure n’existent pas indépendamment en mémoire. Cela justifie notamment la possibilité de définir des tableaux de structures (voir section suivante).

En particulier, on peut accéder aux champs de la manière suivante :

```
struct Something {
    int a;
    short int b;
    char c;
};

...

struct Something s = {.a = 0x01234567, .b = 0x89AB, .c = 0xCD};
void *ptr = (void*)&s;

s.a == (int)*(ptr + 0);
s.b == (short int)*(ptr + sizeof(s.a));
s.c == (char)*(ptr + sizeof(s.a) + sizeof(s.b));
```

**Remarque :** les assertions sont correctes car `struct Something` est correctement aligné (voir la sous-section 6.12.4 suivante).

Les adresses des champs d’une instance de structure sont seulement des décalages (*offsets* en anglais) vis-à-vis de l’adresse de l’instance elle-même.

Comme ces décalages sont calculés à la compilation du programme, l’opération d’accès à un membre peut aussi être calculée à la compilation.

L’utilisation de structures ne ralentit donc pas l’exécution du programme.

### 6.12.3 Tableaux de structures

Comme les structures définies sont considérées comme des entités de première classe et sont compactes en mémoire, il est évidemment possible de créer statiquement 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 }

```

On peut également effectuer des allocations dynamiques avec les pointeurs par le `malloc`. Son utilisation est identique à celle permettant l'allocation dynamique de tableaux de types intrinsèques au langage :

```

1     ...
2
3     int main() {
4         int i = ...;
5         struct Time* running_times = malloc(sizeof(struct Time) * 10);
6         running_times[i].hour = ...;
7         return EXIT_SUCCESS;
8     }

```

Il est ainsi 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>70</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 est cependant très lourde à écrire, alors une notation purement facilitatrice existe en C, la flèche :

<sup>70</sup>. En effet, l'opérateur d'accès à un membre 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 (voir sous-section précédente)

```

struct MyStruct {
    int a;
    int b;
};

struct MyStruct *s = ...;

printf("a = %d, b = %d\n", s->a, s->b);

```

```

struct MyStruct {
    int a;
    int b;
};

struct MyStruct *s = ...;

printf("a = %d, b = %d\n", (*s).a, (*s).b);

```

### 6.12.4 Alignement et optimisation des structures

On aborde ici un point technique ayant à voir avec l'optimisation du code de manière générale, et plus précisément l'optimisation des structures.

Si un processeur tente d'accéder à une valeur codée sur  $N$  octets située à une adresse  $a \neq 0 [N]$ , il va devoir effectuer des calculs supplémentaires pour se décaler de la valeur du modulo et se trouve ralenti.

Ainsi, on veut que l'adresse d'un `int` soit un multiple de 4 octets, que l'adresse d'un `char` soit multiple de 1 octet, que l'adresse d'un `double` soit multiple de 8 octets, etc...

#### Définition 37 : Alignement

On dit qu'une donnée d'adresse  $a$  est alignée sur  $N$  octets si  $a \equiv 0[N]$

On cherche donc à aligner les adresses des objets sur leurs taille.

Prenons les deux structures suivantes :

```

1 struct ExampleUnaligned {
2     char c;
3     int a;
4     short b;
5 };
6
7 struct ExampleUnaligned x;

```

```

1 struct ExampleAligned {
2     int a;
3     short b;
4     char c;
5 };
6
7 struct ExampleAligned x;

```

Les deux structures *semblent* équivalentes, pourtant elles ne le sont pas.

Pour garantir l'alignement de la donnée de plus grande taille, on aligne les structures elles-mêmes sur cette plus grande taille. Ainsi, chacune des deux structures est alignée sur 4 octets.

#### Dans le cas de la structure `ExampleAligned` :

- $x.a$  est à l'adresse  $\&x$ , donc aligné sur 4 octets, ce qui est correct
- $x.b$  est à l'adresse  $\&x + 4$ , donc aligné sur 2 octets, ce qui est correct
- $x.c$  est à l'adresse  $\&x + 6$ , donc aligné sur 1 octet, ce qui est correct

#### Dans le cas de la structure `ExampleUnaligned` :

- $x.c$  est à l'adresse  $\&x$ , donc aligné sur 1 octet, ce qui est correct
- $x.a$  est, *a priori*, à l'adresse  $\&x + 1$ , donc *non aligné* sur 4 octets
- $x.b$  est, *a priori*, à l'adresse  $\&x + 5$ , donc *non aligné* sur 2 octets

Ainsi, la structure `ExampleUnaligned` devrait être plus lente que la structure `ExampleAligned`.

Le compilateur effectue cependant un alignement automatique lors de la génération du code des structures pour que chaque adresse des champs d'un objet de type complexe, défini par une structure, soit congrue à la taille de la donnée.

Dans le cas de la structure `ExampleUnaligned` :

- $x.c$  est toujours à l'adresse  $\&x$
- $x.a$  est placé à l'adresse  $\&x + 4$  pour forcer l'alignement
- $x.b$  est placé à l'adresse  $\&x + 8$  pour forcer l'alignement

La taille de la structure devient 10.

Si on veut pouvoir créer un tableau de cette structure sans problèmes d'alignement, il faut arrondir la taille de la structure à son alignement, donc à 12.

Finalement, `sizeof(struct ExampleUnaligned) = 12`, tandis que `sizeof(struct ExampleAligned) = 8`.

### 6.12.5 Exercices

**Exercice 46 (Matrices (2)) [13].** Écrire une structure `struct Matrix` qui contient trois champs :

- `double** mat;`
- `unsigned int n;`
- `unsigned int m;`

et modifier les routines de l' **Exercice 45** pour qu'elles utilisent la structure `struct Matrix`.

**Exercice 47 (Optimisation d'alignement) [03].** Déterminer les tailles des structures `ExampleAligned` et `ExampleUnaligned` avec un `double` à la place d'un `int`.

**Exercice 48 (Listes chaînées) [27].** L'objectif de cet exercice est d'introduire une nouvelle manière de structurer les données. La section 9.2 est consacrée à cette structure <sup>71</sup>

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>72</sup>, comme un carnet d'adresses par exemple.

On commence par construire l'enregistrement 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 :

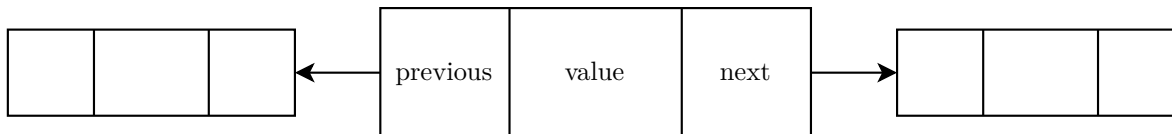


FIGURE 6.2 – Noeud d'une liste

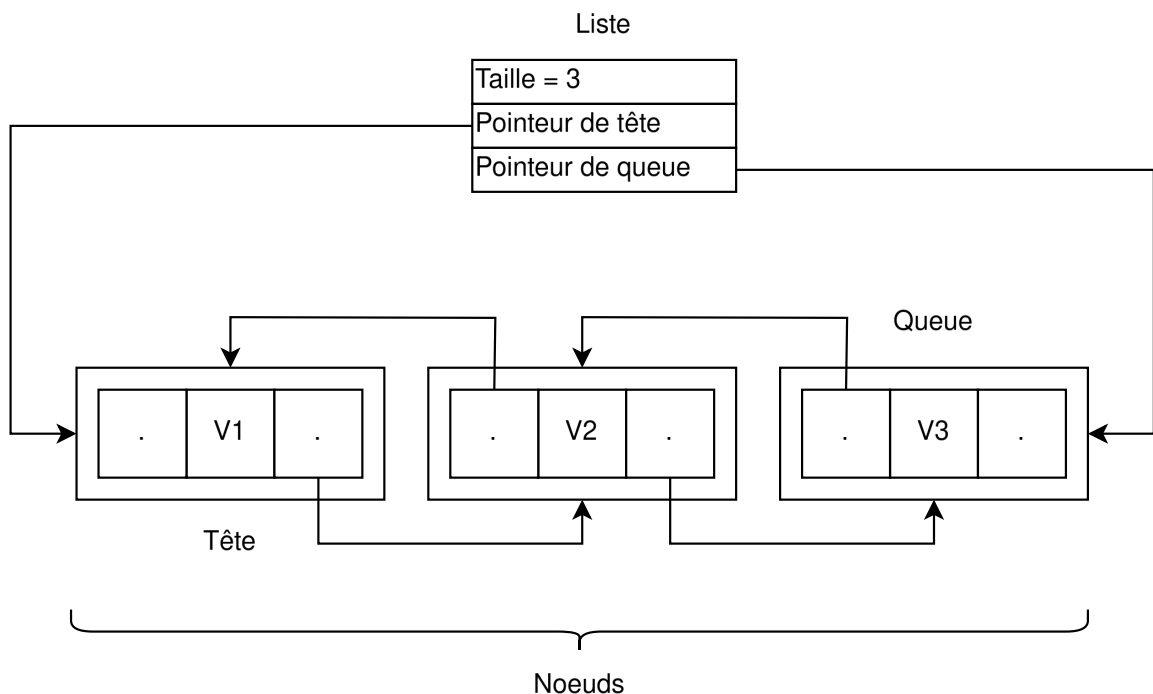
71. L'idée de mettre sous forme d'exercice ce qui apparaîtra plus tard dans le cours est de rendre la partie 2 sur le langage C complète vis-à-vis des cours Algo/Prog 1 de l'ISMIN sans alourdir trop le cours. La section sur les listes semblera parfois un peu de la redite, mais sera bien plus rigoureuse grâce à l'introduction à la section 9.1 des types abstraits de données.

72. Dans la limite de la mémoire de l'ordinateur

L'idée fondamentale du chaînage<sup>73</sup> est que chaque noeud sera alloué dynamiquement et sera donc situé dans un espace mémoire indépendant de celui des autres noeuds. Cela permet de libérer la mémoire allouée pour un noeud en conservant les autres noeuds, ou encore d'allouer de nouveaux noeuds sans avoir à s'occuper des adresses des autres noeuds.

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 *tail*. La *tête* de liste est le premier noeud de la liste. La *queue* de liste est le dernier noeud de la liste. On obtient grâce à cette structure de liste le schéma suivant (ici, une liste de trois éléments de valeurs  $V_1$ ,  $V_2$  et  $V_3$ ) :



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.

<sup>73</sup>. On observera qu'il est possible d'implanter le type abstrait *Liste* sans chaînage, par des tableaux. Les avantages et inconvénients sont différents et dépendent de l'application.

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*.

**Attention :** la difficulté principale réside dans le fait de relier correctement les noeuds entre eux grâce aux pointeurs. Il faut faire preuve de rigueur pour qu'après exécution de la fonction on ait bien le schéma ci-dessus.

7. De même, écrire une procédure `void linkedlist_push_on_tail(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_tail(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. Écrire une fonction `void linkedlist_destroy(struct LinkedList *l)`; qui détruit une liste en libérant la mémoire de chacun de ses noeuds et de la liste elle-même.
10. 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.

*Note :* On pourra implanter l'algorithme 9 ci-dessous par une boucle *for*.

---

**Algorithme 9 :** Parcours de liste

---

**Entrées :** *Liste : l, Procédure : p*

- ```

1 Noeud tmp ← tete(l);
2 tant que successeur(tmp) existe faire
3   | p(contenu(tmp));
4   | tmp ← successeur(tmp);
```
- 



## MODULATION ET ENTÊTES



Quand un programme est suffisamment petit, on peut garder en tête tous les détails de ce programme dans notre tête à chaque instant. Cependant, les applications réelles sont en général plusieurs ordres de magnitude plus grandes que celles écrites durant les cours de programmation. Elles deviennent alors simplement trop complexes pour tenir de bout en bout dans notre tête, et sont développées par plusieurs programmeurs. Dans le cas du développement d'une application complexe, il arrive donc souvent que la division d'un programme informatique en routines dans un unique fichier soit insuffisante pour organiser le code. 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. Le code est alors divisé en plusieurs *modules*. Un module est un composant du programme, et peut lui-même utiliser d'autres modules pour son bon fonctionnement. On appelle *programmation modulaire* le développement d'une application par modules.

En C, on utilise deux fichiers pour définir un module :

- un fichier d'entête (*header* en anglais) d'extension *.h* qui sert d'interface du module, et indique les structures, routines et constantes définies par celui-ci
- un fichier de code source d'extension *.c* qui fournit l'implantation réelle de l'interface, c'est-à-dire des routines du module

On peut penser aux modules informatiques comme aux modules d'une fusée spatiale. Chacun est indépendant dans son fonctionnement interne mais tous les modules sont interdépendants d'un point de vue abstrait puisque chacun est constitutif du programme et ne peut pas résoudre le problème individuellement.<sup>74</sup>

```
user@computer ~/working_directory> ls
main.c module1.c module1.h module2.c module2.h etc...
user@computer ~/working_directory>
```

### 6.13.1 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 définitions de type (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 dans son code. Il agit donc comme un genre de *spécification* du module, nécessaire pour le compilateur et utile pour le programmeur.

Il s'agit en fait 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.

### 6.13.2 Fichier de code source

Le fichier de code source, ou simplement fichier source, fournit l'implantation de l'interface décrite 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 :

---

74. Si cela est le cas, le programme n'est plus modulaire. On peut donc s'interroger sur la qualité de la conception.

```
#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
```

Cela permet au compilateur d'avoir accès aux définitions de l'interface, notamment les types et les constantes du préprocesseur.

**Remarque :** on peut très bien programmer une routine dans le fichier de code source sans donner son prototype dans le fichier d'entête. Cette routine n'est donc accessible qu'en interne du module, et ne l'est pas par un programme extérieur.

### 6.13.3 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>75</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 lisibilité<sup>76</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> // Les directives de préprocesseurs sont écrites dans l'entête
2
3  // définition de type
4
5  struct Vec2 {
6      double x;
7      double y;
8  };
9
10 // prototypes de routines
11
12 struct Vec2 vec2_new(double x, double y);
13 struct Vec2 vec2_copy(struct Vec2 p);
```

<sup>75</sup>. 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.

<sup>76</sup>. 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.

```

14
15 void vec2_display(struct Vec2 p);
16 struct Vec2 vec2_scalar_multiply(struct Vec2 p, double k);
17 struct Vec2 vec2_add(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" // Nécessaire pour accéder à l'interface (décrite dans l'entête)
2
3  // implantation, dans un ordre quelconque, des routines
4
5  struct Vec2 vec2_new(double x, double y) {
6      struct Vec2 v = {x, y};
7      return v;
8  }
9  struct Vec2 vec2_copy(struct Vec2 p) {
10     struct Vec2 v = {p.x, p.y};
11     return v;
12 }
13
14 void vec2_display(struct Vec2 p) {
15     printf("(%lf, %lf)", p.x, p.y);
16 }
17
18 struct Vec2 vec2_scalar_multiply(struct Vec2 p, double k) {
19     struct Vec2 v = {p.x * k, p.y * k};
20     return v;
21 }
22
23 struct Vec2 vec2_add(struct Vec2 p1, struct Vec2 p2) {
24     struct Vec2 v = {p1.x + p2.x, p1.y + p2.y};
25     return v;
26 }

```

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" // Accède à l'interface décrite
4  // -> permet d'utiliser les routines et structures décrites
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 }

```



**Remarque :** tous les prototypes sont connus par le fichier de code source grâce à l’inclusion de l’entête. Le problème de devoir définir une routine strictement avant son utilisation est donc résolu. On peut donc écrire le code de chaque routine dans un ordre quelconque.

### 6.13.4 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/ccCAnFoX.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 [18]
4. l’édition des liens [18][15]

### 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>77</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 :

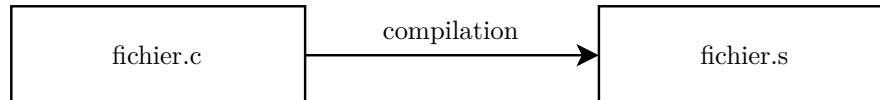
1. l’analyse lexicale : identifie dans le code source les différentes unités lexicales<sup>78</sup>
2. l’analyse syntaxique : construit un arbre de la syntaxe du programme et vérifie que celle-ci est correcte vis-à-vis d’une grammaire définie formellement

<sup>77</sup>. Par exemple grâce à `#ifdef`

<sup>78</sup>. 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.
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<sup>79</sup>

À ce niveau, est associé à chaque fichier source un fichier d'extension `.s` qui correspond au langage assembleur *gas* utilisé par *gcc*<sup>80</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>81</sup>

On peut générer le code assembleur d'un programme grâce à l'argument `-S` :

```

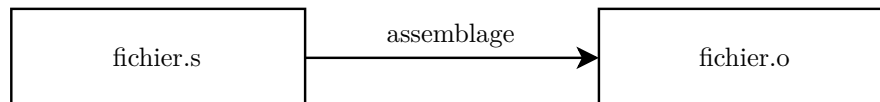
user@computer ~/working_directory> ls
main.c vec2.c vec2.h
user@computer ~/working_directory> gcc main.c -S
user@computer ~/working_directory> ls
main.c main.s vec2.c vec2.h
user@computer ~/working_directory>
  
```

qui peut être lu par un éditeur de texte classique puisqu'il s'agit simplement d'un programme écrit dans un langage bas niveau.

## Assemblage

*gcc* fait appel à un programme tiers nommé *assembleur* qui génère le code machine associé au code assembleur.<sup>82</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
  
```

79. Il ne faut pas comprendre le « au passage » comme si ces opérations d'optimisations étaient triviales. Il s'agit de la partie la plus complexe du compilateur.

80. Pour plus de détails : [https://en.wikipedia.org/wiki/GNU\\_Assembler](https://en.wikipedia.org/wiki/GNU_Assembler)

81. 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.

82. 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.

```

user@computer ~/working_directory> ls
main.c main.o vec2.c vec2.h
user@computer ~/working_directory>

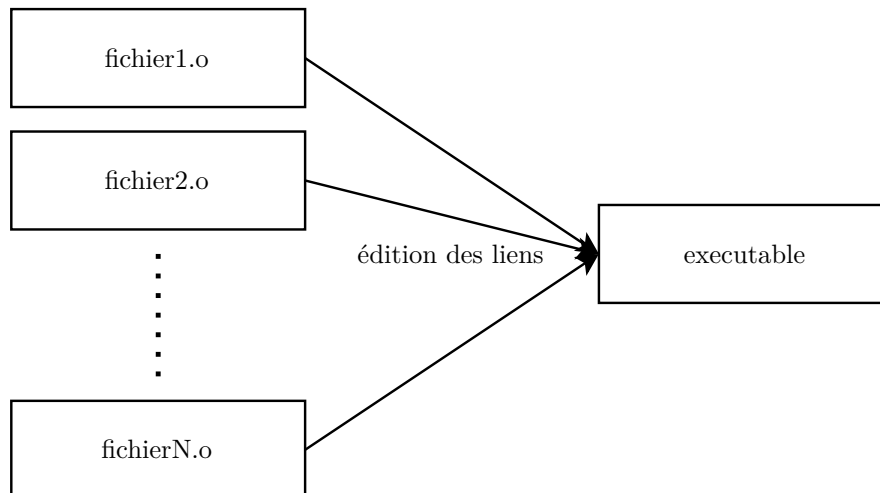
```

Cette fois-ci, le fichier objet devient assez illisible par un éditeur de texte puisqu'il contient du code machine.

Cependant, il n'est pas encore exécutable par l'ordinateur. En effet, certaines fonctions ne sont pas définies dans le fichier objet lui-même mais ailleurs sur l'ordinateur comme ce peut être le cas avec la fonction `printf` du C.

## 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. Il va aller chercher les définitions de chacun des symboles du programme, c'est-à-dire les instructions de chaque routine qui n'est pas définie dans le fichier objet lui-même.



Le programme exécutable résultant est donc l'agrégation du code de chacune des routines qui est utilisé dans 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 lors de l'édition des liens.

L'éditeur de liens relève donc une erreur.<sup>83</sup>

### 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>84</sup>. Certains outils d'automatisation simplifient ces opérations et permettent de gagner en productivité (voir la section 10.1 sur les *makefiles*).

De manière générale, on écrit :

```
user@computer ~/working_directory> gcc main.c module1.c module2.c ... moduleN.c -o main
user@computer ~/working_directory>
```

Chaque fichier source va être compilé en un fichier objet *moduleN.o*, et l'éditeur de lien va lier chacun de ces *N* fichiers objets en un unique exécutable.

### 6.13.5 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* :

module1.h

```
struct SomeStructure {
    int thing;
};
```

module2.h

83. Et même deux en l'occurrence.

84. Qui est en général beaucoup plus complexe, avec l'utilisation de plus de paramètres.

```
#include "module1.h" // Accède à l'interface de "module1"

/*
Ici : Une interface quelconque
*/
```

main.c

```
#include "module1.h" // Accède à l'interface de "module1"
#include "module2.h" // Accède à l'interface de "module2"

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 définit deux fois du point de vue de “*main.c*”.

On peut aussi 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" // Accède à l'interface de "module1"
```

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.

### 6.13.6 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> <i>#ifndef SOME_CONST</i> <i>/*</i> <i>Code exécuté si SOME_CONST est définie</i> <i>*/</i> <i>#else // pas obligatoire</i> <i>/*</i> <i>Code exécuté si SOME_CONST n'est pas définie</i> <i>*/</i> <i>#endif</i> </pre> | <pre> <i>#ifndef SOME_CONST</i> <i>/*</i> <i>Code exécuté si SOME_CONST n'est pas définie</i> <i>*/</i> <i>#else // pas obligatoire</i> <i>/*</i> <i>Code exécuté si SOME_CONST est définie</i> <i>*/</i> <i>#endif</i> </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 // Cette section ne peut être lu plus d'une fois
#define MODULE_INCLUDED // car le symbole est définie juste en dessous
/*
Le module n'est lu qu'une fois
*/
#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" // Accède à l'interface de "module2"

struct SomeStructure {
    int thing;
};

#endif

```

module2.h

```

#ifndef MODULE2_H_INCLUDED
#define MODULE2_H_INCLUDED

#include "module1.h" // Accède à l'interface de "module1"

void array_sort(int array[], unsigned int length);

#endif

```

### 6.13.7 Exercices

**Exercice 49 (Un module de listes chaînées)** [13]. Écrire un module qui contiennent les structures et fonctions définies dans l' **Exercice 48** .



## CHAÎNES DE CARACTÈRES



**Rappel :** La représentation ASCII est détaillé en table 2.5. On peut l'obtenir depuis un terminal Linux en tapant *man ascii*.

### 6.14.1 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'entièreté 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>85</sup>

### 6.14.2 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>86</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>87</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 texte[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 :

<sup>85</sup>. 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 ;)

<sup>86</sup>. Autant dire qu'en C, il n'y a que le strict minimum hein !

<sup>87</sup>. Coïncidence ? Je ne crois pas.

```
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 ?*”<sup>88</sup>, 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 après le 'l', et 0 en fin de chaîne :
char text[50] = {'H', 'e', 'l', 'l', '\0', ' ', 'W', 'o', 'r', 'l', 'd', ' ', '!', 0};
printf("%s\n", text); // affiche 'Hell' dans la console
```

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); // Affiche 'Hello World !' dans la console
```

### 6.14.3 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>89</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 :

88. Traduction : *Nom de Dieu de putain de bordel de merde de saloperie de connard d’enculé de ta mère... Vous voyez, c’est aussi jouissif que de se torcher le cul avec de la soie. J’adore ça. C’est quoi ce zéro à la fin de la chaîne ?*

89. Étonnant n’est-ce pas ? Qui l’aurait crû ?



```
| printf("%s", "%s\n"); // 0 est ajouté automatiquement
```

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 manipulée par `printf`, c’est-à-dire la première.

### 6.14.4 Exercices

**Exercice 50 (Calculatrice) [12].** 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
> 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 car le formatage de `scanf` les comprend.

*Indication :* un caractère est un nombre, il peut donc être utilisé dans un aiguillage ‘switch-case’

**Exercice 51 (Atoi) [15].** Écrire une fonction `int atoi(char* string)` ; qui prend en entrée une chaîne de caractères représentant un nombre. Le signe doit être pris en considération. Si la chaîne contient des espaces, ils doivent être ignorés. Le programme ignore tout ce qui suit la première lettre rencontrée. Si l'entrée n'est pas valide, le comportement est indéterminé (*i.e.* laissé au choix du programmeur).

Exemples de sortie :

- “42” : 42
- “-53” : -53
- “ +74TEXTE” : +74

*Note :* Aucune routine de la bibliothèque standard ne doit être utilisée dans `atoi`!!!<sup>90</sup>

---

90. Trop facile sinon...



## LES FLUX DE FICHIERS



*Recommandation* : revoir le vocabulaire sur les fichiers de la partie 1 du cours (section 3.2).

La manipulation de fichiers est vitale pour l'écriture de programmes persistents<sup>91</sup>, 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 fournit différentes fonctions pour manipuler les fichiers, toutes déclarées dans l'en-tête `<stdio.h>`. Celles-ci manipulent non pas directement 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>92</sup>, ou rediriger le flux de lecture d'un fichier vers la console pour en afficher le contenu, comme le ferait la commande *cat* du terminal.

### 6.15.1 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 plus proche qu'il n'y paraît par certains aspects. Il ne sera pas traité.<sup>93</sup> Le but est simplement de comprendre l'existence de droits.

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
- 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 au niveau local comme au niveau réseau.

Par ailleurs, chaque fichier sous Linux :

- a un propriétaire<sup>94</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>95</sup>

91. À ne pas confondre avec la persistance de structures de données immutables.

92. [https://fr.wikipedia.org/wiki/Historique\\_\(informatique\)](https://fr.wikipedia.org/wiki/Historique_(informatique))

93. Pour plus d'informations : <https://learn.microsoft.com/en-us/windows/security/identity-protection/access-control/access-control>

94. Il ne s'agit pas nécessairement du créateur du fichier puisque la propriété peut être transmise

95. Appeler "utilisateur" le propriétaire est au mieux vague, au pire dénué de sens. Mais bon, c'est comme ça.

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 -l
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 : Taille virtuelle et taille sur disque des fichiers

Le total indiqué est le nombre de ko utilisés (1 *ko* = 1024 *octets*) pour stocker en mémoire les données affichés par  $ls$ . On 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.

Les droits d'utilisateurs sont indiqués à gauche. Le format est le suivant :

$\underbrace{\quad} \quad \underbrace{\quad} \quad \underbrace{\quad} \quad \underbrace{\quad}$   
 type propriétaire groupe autres

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 :

$\underbrace{\quad} \quad \underbrace{rwx} \quad \underbrace{rw-} \quad \underbrace{---}$   
 type propriétaire groupe autres

donne les informations suivantes :

- 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>
```

## 6.15.2 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
- `fprintf` pour écrire dans un flux de fichier *textuel*
- `fscanf` pour lire dans un flux de fichier *textuel*
- `fwrite` pour écrire dans un flux de fichier *binaire*
- `fread` pour lire dans un flux de fichier *binaire*
- `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*.

### Les fonctions `fprintf` et `fscanf`

Elles s'utilisent comme vu précédemment dans la section sur les flux standards. Le flux à préciser est le pointeur de type `FILE*`.

**int `fclose(FILE *stream);`**

**Entrée :** *stream* est le flux à fermer

**Sortie :** 0 en cas de succès. EOF en cas d'erreur.<sup>96</sup>

### Un classique de l'ISMIN

Rien ne vaut un exemple à ce stade.<sup>97</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 (fd == NULL) {
21         fprintf(stderr, "Erreur d'ouverture du fichier %s\n", pathname);
22         return;
23     }
24     if (!fwrite(&p, sizeof(struct Person), 1, fd)) {
25         fprintf(stderr, "Erreur d'écriture dans annuaire.data\n");
26     }
27     fclose(fd); // nécessaire : permet au fichier d'être réouvert plus tard
28 }
29
```

96. EOF est une constante définie dans `<stdio.h>` et vaut généralement `-1`

97. 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 seuls avec eux-mêmes. Ce serait inconscient.

```

30 void person_print_numbers_from_file(const char *pathname) {
31     FILE *fd = fopen(pathname, "rb");
32     if (fd == NULL) {
33         fprintf(stderr, "Erreur d'ouverture du fichier %s\n", pathname);
34         return;
35     }
36     struct Person p;
37     while (fread(&p, sizeof(struct Person), 1, fd) == 1) {
38         printf("%s : %ld\n", p.name, p.number);
39     }
40     fclose(fd); // nécessaire : permet au fichier d'être réouvert plus tard
41 }
42
43 int main() {
44     struct Person p = person_make();
45     person_add_to_file("annuaire.data", p);
46     person_print_numbers_from_file("annuaire.data");
47     return EXIT_SUCCESS;
48 }

```

### 6.15.3 Rediriger un flux standard vers un fichier

La redirection de flux a été évoquée précédemment et on montre ici comment la réaliser.

#### Petit aparté : Liste des remarques les plus pertinentes de ChatGPT

Série de prompts suivante :

*“Liste des quelques remarques inutiles à propos des remarques inutiles.”*

*“Essaie de les rendre plus vides de sens”*

*“Reformule ces remarques en utilisant le moins de vocabulaire possible”*

*“Peux-tu écrire le même texte sans métaphores ?”*

- Les remarques inutiles existent sans but précis.
- Une remarque inutile est simplement un mot sans effet.
- Les remarques inutiles n'ont aucune conséquence.
- Une remarque inutile n'a de valeur que si quelqu'un l'écoute.
- Les remarques inutiles n'ont pas de réelle fonction.
- Les remarques inutiles n'ont de sens que dans un contexte où rien n'a d'importance.
- Une remarque inutile n'est souvent même pas retenue.

Ce que ça peut être con une I.A. quand même...

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  }
```

### 6.15.4 Exercices

Voir les annales d'*Algorithmes et Programmation 1*. Toutes les notions nécessaires ont été abordées (voir l'**Exercice 48** de la section 6.12 pour les listes chaînées). Une correction par Minitel (et par les profs (beurk!)) sera proposé aussi vite que possible sur le NextCloud.

# CONCEPTS AVANCÉS



## VIRGULE ET EXPRESSIONS



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<sup>1</sup>.

### 7.1.1 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 virgule permet d'étendre la notion d'effet de bord aux expressions.

La syntaxe est la suivante :

```
| expression = (expr1, ..., exprN);
```

Elle est équivalente à :

```
| expr1;  
| expr2;
```

1. Et très honnêtement, ça ne sert pas à grand-chose. . . Mais si jamais, c'est là ! La plupart des sections de ce chapitre sont utiles, mais c'est rigolo d'en mettre une inutile dès le début :)



```
...;
expr<N-1>;
expression = exprN;
```

Un exemple purement technique consistant à compter le nombre d’assignations à des variables<sup>2</sup> :

```
1  #include <stdio.h>
2  #include <stdio.h>
3
4  unsigned int _assign_count = 0;
5
6  int main(int argc, char **argv) {
7      int a = (_assign_count++, 42);
8      int b = (_assign_count++, 64);
9      b = (_assign_count++, a + b);
10     int c = (_assign_count*b + a);
11     printf("Nombre d'assignations : %u\n", _assign_count);
12     return EXIT_SUCCESS;
13 }
```

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);
```

**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. D’après le tableau 6.1, la virgule est le moins prioritaire des opérateurs. Ainsi :

```
int x = 1, 2, 3;
```

est incorrect car équivalent à :

```
((int x = 1), 2), 3);
```

qui n’a pas de sens.

### Expression à partir d’un bloc

L’opérateur virgule permet d’exécuter des expressions par effet de bord. Mais ces expressions sont assez courtes, alors qu’on voudrait plutôt pouvoir exécuter des blocs de code entier.

Cette technique sera aussi utile pour la définition de macros.

2. On pourra “cacher” le comptage par une macro dans la section 7.10

Un bloc de code entre accolades définit un environnement local. On peut en fait transformer ce bloc de code en expression par un parenthésage.

```
// Ceci est une expression :
({
    // CODE
});
```

La valeur de l’expression est la valeur de la dernière ligne :

```
int sum = ({
    int x = 4;
    int y = 5;
    x + y;
});
// sum = 9
```

Le programme suivant est donc syntaxiquement parfaitement valide, compile et s’exécute sans problème :

```
int result = ({
    printf("Du CODE !\n");
    result = ...; // symbole défini assignation donc 'result' est bien défini ici.
    f(result);
});
```

En particulier, un bloc de code “expressionnisé” peut être sans problème utilisé par l’opérateur virgule :

```
(..., ({}), ...); // valide (si le reste l'est)
```

### 7.1.2 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;
```

### 7.1.3 Conclusion

Ben voilà c’est tout...<sup>3</sup>

---

3. J’espère que cette conclusion dithyrambique plaira ^^



## 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 livre 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*

### 7.2.1 Le (véritable ?) prototype de *main*

Il existe en vérité deux formes standards possible pour le prototype de *main* :

```
int main(void);  
int main(int argc, char *argv[]);
```

Si le programme n'a pas besoin de lire d'arguments fournis au programme, on peut utiliser le premier prototype. Dans le cas contraire, les deux paramètres de la seconde version 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>4</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  $argv[0] = \text{nom du programme}$  :

---

4. Le double pointeur est ainsi justifié :

- ◆ les crochets désignent le tableau de caractères
- ◆ l'étoile désigne le tableau des tableaux de caractères

```
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 ~>
```

## 7.2.2 Exercices

**Exercice 52 (Liste des arguments) [5].** 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>
```

**Exercice 53 (Un cat minimaliste) [13].** 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>
```



# ÉNUMÉRATIONS



## 7.3.1 Motivation

Il arrive régulièrement d'utiliser des constantes d'état lors de l'écriture de programmes. La valeur réelle de ces constantes importe souvent peu. Il faut simplement que chacune soit unique. Par exemple, les

codes d'erreurs de fonctions peuvent requérir de tels constantes d'états. Il faut pouvoir donner un symbole à chaque constante pour qu'un programmeur tiers puisse lire le programme.

On imaginer un programme qui effectue une requête HTTP <sup>5</sup> par exemple :

|                                                                                                                                                                                                                                    |                                                                                                                                                                                                                                            |
|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <pre> 1  int c = requete(); 2  switch (c) { 3      case 200: 4          ... 5          break; 6      case 408: 7          ... 8          break; 9      default: 10         printf("Code inconnu\n"); 11         break; 12 } </pre> | <pre> 1  enum HttpStatusCode c = requete(); 2  switch (code) { 3      case E_OK: 4          ... 5          break; 6      case E_TIMEOUT: 7          ... 8          break; 9      default: 10         printf("Code inconnu\n"); 11 } </pre> |
|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|

À gauche, c'est illisible à moins de connaître par coeur tous les codes, tandis qu'à droite, la compréhension du code est aisée.

### 7.3.2 Syntaxe

Le mot-clé `enum` permet de déclarer des énumérations.

```

1  enum HttpStatusCode {
2      E_CONTINUE = 100,
3      E_SWITCHING_PROTOCOLS, // = 101, incrémente de 1 depuis la dernière valeur explicite
4      E_PROCESSING, // = 102
5      E_EARLY_HINTS, // = 103
6      E_OK = 200,
7      E_CREATED, // = 201
8      // etc...
9  };

```

**Remarque :** Une constante définie par une énumération est systématiquement un `int`

#### *Compile time vs Running time*

Comme les `DEFINE`, les énumérations sont remplacés dans le code à la compilation. La différence principale tient au fait que les énumérations ne permettent pas de définir autre chose que des entiers (voir section 7.10).

Comme les énumérations sont définies et calculés au moment de la compilation du programme, il est impossible durant l'exécution du programme d'étendre une énumération.

L'intérêt des énumérations vis-à-vis des `define` tient surtout à l'incrémentation automatique et à la lisibilité du code, ce qui est *très* important. On pourrait s'en passer et n'utiliser que des `define`, bien qu'un lecteur tiers ait moins d'informations. Ce n'est pas conseillé.

5. J'aime les chats : <https://http.cat/>



### 7.4.1 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.

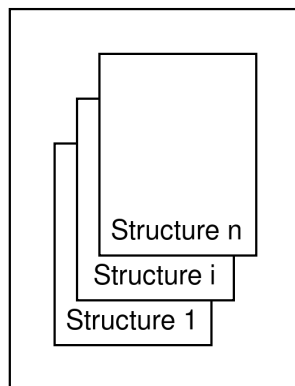
En soi, il suffirait pourtant de seulement 8 octets pour stocker soit la sous-structure `mouse`, soit la sous-structure `keyboard`. C'est à cela que sert l'union.

### 7.4.2 Principe et syntaxe

L'idée de l'union est assez simple. Alors que la structure agrège les données<sup>6</sup> les unes à la suite des autres en mémoire, l'union les superpose dans le même espace mémoire :

6. En respectant ou non l'alignement

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>7</sup>
- 8 octets pour union Event

Dans le cas de la programmation de systèmes embarqués 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 événements. La bibliothèque *Xlib* qui sert d'interface au serveur de fenêtrage X11 de Linux utilise une union de 33 structures pour stocker les événements.<sup>8</sup>

### 7.4.3 Le problème de sélection

Dans le cas précédent de l'union `union Event`, il est impossible étant seulement donné l'union de déterminer lequel des deux champs utiliser. On peut imaginer un programme qui renvoie une liste d'événements pour lequel il serait parfaitement envisageable que chaque événement utilise un champ

7. Et pas 14 du fait de l'alignement mémoire, voir la section 6.12.4

8. Voir : <https://tronche.com/gui/x/xlib/events/structures.html>

différent. Une solution pourrait être de renvoyer autant de listes qu'il y a de type d'évènement différents. Si on a 100 évènements différents et seulement 2 évènements réellement utilisés, beaucoup de temps est perdu à vérifier chaque liste vide<sup>9</sup>. Une autre solution consiste à utiliser une énumération pour indiquer par un champ supplémentaire le type de l'évènement :

```
enum Type {
    T_MOUSE,
    T_KEYBOARD
};
struct Event {
    union {
        struct {
            unsigned x;
            unsigned y;
        };
        struct {
            int keycode;
            char is_maj_enabled;
            char is_ctrl_enabled;
        };
    } e;
    enum Type t;
};
```

On peut alors utiliser un aiguillage lors du parcours des évènements sur le champ `t` pour adapter le code :

```
for (struct Noeud nd = first(list); not_last(list, nd); nd = next(list, nd)) {
    struct Event event = value(nd);
    switch (event.t) {
        case T_MOUSE:
            // Code pour souris sur :
            event.e.x;
            event.e.y;
            break;
        case T_KEYBOARD:
            // Code pour clavier sur :
            event.e.keycode;
            event.e.is_maj_enabled;
            event.e.is_ctrl_enabled;
            break;
        default:
            // Inconnu
            break;
    }
}
```

Utiliser les unions à la place de structures quand cela est possible est une bonne pratique de programmation à assimiler.

9. Sans parler des allocations mémoire supplémentaires, l'impossibilité de tout stocker dans un seul tableau et les problèmes de cache qui vont avec.



# CHAMPS DE BITS

Cette section vient en continuité de la section précédente 7.4 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.

## 7.5.1 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 :

$\underbrace{000000000000}_{\text{inutilisées}} \quad \underbrace{aaaaaaaaaaaa}_{12 \text{ bits pour l'année}} \quad \underbrace{mmmm}_{4 \text{ bits pour le mois}} \quad \underbrace{dddd}_{5 \text{ bits pour le jour}}$

```

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
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 & 0xFFF) << 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>10</sup>

## 7.5.2 Syntaxe

Les champs de bits<sup>11</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>12</sup>

Pour revenir à l'exemple des dates :

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  struct Date {
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)); // -> 12
18     printf("%ld\n", sizeof(struct BFDDate)); // -> 4
19
20     struct BFDDate d = {.day=22, .month=12, .year=2003};
21     printf("%u/%u/%u\n", d.day, d.month, d.year); // -> 22/12/2003
```

10. 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...

11. 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 :|

12. 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.

```

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 7.5.1 précédente.

Par ailleurs, l'accès en lecture/écriture des champs est bien plus aisée !

### 7.5.3 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 BFDDate {     unsigned year : 12;     unsigned day : 5;     unsigned month : 4; };  int main(int argc, char *argv[]) {     struct BFDDate 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>13</sup> du compilateur par :

```
gcc main.c -o main -O0
```

13. 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 débogage.

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.  $\text{rapport} = \frac{2.42}{0.94332} = 257\%$
2.  $\text{rapport} = \frac{2.15}{0.00299} = 71906\%$
3.  $\text{rapport} = \frac{2.82}{0.00281} = 100356\%$

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.

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>14</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.

**Remarque :** Les mesures de performance sont appelés des *benchmarks*. Il peut s'agir de mesurer la qualité de la sortie du programme, son temps d'exécution, la quantité de mémoire utilisée, etc... Le *benchmarking* sera abordé plus proprement<sup>15</sup> dans la quatrième partie.



## CLASSES DE STOCKAGE



Cette section vient apporter des outils pour préciser la durée de vie d'une variable et sa portée d'utilisation.

### Définition 38 : 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. C'est-à-dire les régions de code correspondant à l'environnement local auquel elle appartient.
- la durée de vie (*lifetime* en anglais) d'une variable est l'intervalle de temps de l'exécution du programme durant lequel l'espace physique de stockage de la variable est assuré comme réservé pour elle.

14. 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.

15. Je veux dire que le "benchmarking" dans cette section est fait complètement à l'arrache pour simplifier. Juste histoire de rassurer ceux qui seraient un peu plus puristes.

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.

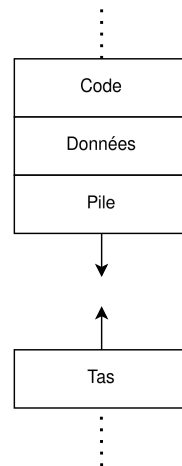
### 7.6.1 Quelques détails de la structure d'un programme en mémoire

**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 écrivent et lisent dans la mémoire vive
- section de données initialisées et non initialisées : contient les données globales ou *statiques*<sup>16</sup>
- la pile : contient les variables temporaires, les arguments de routines, les adresses de retour des routines au moment de l'appel, etc. . .
- le tas : contient les variables allouées dynamiquement

Schématiquement, cela ressemble<sup>17</sup> à ça :



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 :

<sup>16</sup>. i.e. durables, la notion sera détaillée dans la suite.

<sup>17</sup>. 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. . .

- 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 comme variables temporaires pour les calculs du processeur<sup>18</sup>.

Il est très difficile<sup>19</sup> d'accorder correctement les registres aux variables pendant les calculs. Cette opération est en général effectuée de manière automatisée par le compilateur.

### 7.6.2 Choisir la classe de stockage d'une variable en C

Il existe quatre classes de stockage en C : *auto*, *register*, *static* et *extern*.

- *auto* : explicite le fait que les choix du compilateur sont automatiques<sup>20</sup>
- *register* : force l'utilisation d'un registre pour une variable particulière
- *static* permet de restreindre la portée d'une variable persistente à un environnement local<sup>21</sup>
- *extern* permet d'étendre une variable globale sur plusieurs fichiers sources.

Les deux classes de stockage les plus importantes sont *static* et *extern*.<sup>22</sup>

On peut les utiliser de la façon suivante :

```
auto TYPE x; // équivalent à ne rien préciser
register TYPE x;
static TYPE x;
extern TYPE x;
```

**Remarque :** l'ordre des spécificateurs de la variable n'importe pas. Les deux notations suivantes sont équivalentes :

18. 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.

19. Au sens informatique de complexité, il s'agit d'un problème d'ordonnancement *NP*-difficile.

20. En deux mots : complètement inutile

21. au contraire des variables globales qui sont persistentes mais... globales

22. Par "plus importantes" on entend : possèdent des applications pratiques régulières.

```
CLASSE TYPE x;
TYPE CLASSE x;
```

### La classe *automatique*

La classe *auto* est celle utilisée par défaut pour toutes les variables dont la classe de stockage n'est pas précisé par le programmeur. Les propriétés des variables définies par la classe *auto* sont celles vues depuis le début de ce livre. C'est-à-dire qu'en fonction du contexte, le compilateur choisi lui-même comment stocker ces variables. Ce peut-être :

- en pile d'exécution : pour des variables cantonnés à un environnement local, comme dans une routine
- dans le tas : pour des variables non globales mais qui peuvent être manipulés dans plusieurs routines différentes par exemple
- dans un registre : pour des accès répétés sur une courte période
- en section de données : ce qui peut arriver notamment dans le cas des variables globales

### La classe *register*

La classe *register* permet de forcer le compilateur à utiliser un registre pour stocker une variable si cela est possible<sup>23</sup>. Son utilisation n'est pas recommandée pour la plupart des cas d'usages. Ainsi, dans l'exemple suivant :

```
int main() {
    register int a = 0;
    int x = 1;
    printf("%d", a);

    // pleins de calculs sur 'x' n'utilisant pas 'a'

    a = (x = 0) ? 5 : 3;
    printf("%d", a + x);
}
```

l'exécution des calculs ne nécessite pas la connaissance de *a*, par contre *a* est nécessaire au début et à la fin. Le compilateur est dans l'impossibilité de réassigner le registre utilisé pour *a* pour les calculs sur *x*. Les registres sont rares, mais l'un d'entre eux est gaspillé par l'utilisation abusive du mot-clé *register*.

### La classe *static*

La classe *static* force le compilateur à placer la variable en section de données. La durée de vie de la variable devient alors la durée de vie du programme. Ainsi :

```
1  #include <stdio.h>
2
3  void example() {
4      static int i = 0; // l'initialisation n'a lieu qu'à la première exécution
5      printf("%d", i++);
```

23. Puisqu'il n'existe pas une infinité de registres.

```

6  }
7
8  int main() {
9      ...
10     for (unsigned int u = 0; u < 10; u++) {
11         example();
12     }
13     ...
14 }

```

affiche les nombres de 0 à 9.

**Remarque :** la portée de la variable n'est pas augmentée par la classe *static*. L'ajout de la ligne d'affichage de *i* dans la fonction *main* provoque une erreur de compilation :

```

int main() {
    ...
    for (unsigned int u = 0; u < 10; u++) {
        example();
    }
    printf("%d", i); // erreur !
    ...
}

```

Le mot-clé *static* permet donc de construire des variables à valeur persistante dont la portée est locale. En effet, l'utilisation de variables globales est sinon la seule alternative, et cela pose des problèmes de stabilité du code dans le cas de grands projets.

### La classe *extern*

Il s'agit de la classe de stockage dont l'utilisation est la plus particulière car elle met en jeu plusieurs fichiers. Le principe d'*extern* est d'indiquer au compilateur qu'un symbole définie dans un fichier est globale vis-à-vis de plusieurs autres fichiers, c'est-à-dire qu'il représente à chaque fois la même entité.

Si une variable globale *x* est définie dans un premier fichier *fichier1.c* :

fichier1.c

```
unsigned int x = 5;
```

on peut y faire référence par le mot-clé *extern* dans un deuxième fichier *fichier2.c* :

fichier2.c

```
extern unsigned int x;
```

Comme le symbole représente la même entité, la modification de *x* dans *fichier1.c* entraîne la modification de *x* dans *fichier2.c* et *vice versa*.

La variable globale *x* est ici étendue aux deux fichiers *fichier1.c* et *fichier2.c*.





## CONST ET RESTRICT



On s'est intéressé dans la section précédente 7.6 aux différents moyens permettant d'indiquer au compilateur où les variables déclarées devraient être placées en mémoire, et ainsi de déterminer quelle portée effective dans le code ces variables peuvent avoir et/ou combien de temps elles peuvent rester accessibles.

Cette section va présenter les *indicateurs d'intention d'utilisation des identifiants*. Ces indicateurs en C sont au nombre de deux<sup>24 25</sup> :

- **const** : indique au compilateur que la *rvalue* d'un identifiant ne doit pas être modifiée, *par quelque moyen que ce soit*.
- **restrict** : permet d'indiquer lors de la déclaration d'un pointeur que seul le pointeur lui-même sera utilisé pour accéder à l'objet pointé (et pas un autre pointeur indépendant)

Il est particulièrement important d'indiquer ces précisions dans un code. D'une part, cela permet au compilateur de mieux optimiser le code puisque sa connaissance de la sémantique voulue par le programmeur est augmentée. D'autre part, cela permet d'assurer dans une certaine mesure que le comportement du programme est bien celui attendu.

### 7.7.1 const

#### Types simples

Le mot-clé **const** s'utilise à la déclaration d'une variable devant son type :

```
const int x = 42;
printf("x = %d\n", x); // lecture de la constante
x++; // écriture de la constante : erreur de compilation
```

L'ordre d'écriture de la classe de stockage, du type et du spécificateur de constante n'importe pas :

```
const auto int x = 42;
// équivalent à :
int const auto x = 42;
```

Il faut faire attention à un détail : la valeur d'une constante ne peut lui être donnée qu'à sa déclaration. Ainsi, le code suivant provoque une erreur à la compilation :

```
1 const int x; // valeur indéfinie
2 x = 5; // erreur de compilation
```

24. Sans blague, genre y a pas un titre de section pour l'indiquer...

25. Certains langages plus complexes peuvent en avoir plus, comme le Rust par exemple qui est une vraie usine à gaz syntaxique.

## Types complexes

Dans le cas des types complexes, chaque champ peut être défini individuellement comme constant :

```
struct Type {
    const int a;
    float b;
    const char c;
};
```

Lors de la déclaration d'une variable, chaque champ constant doit être initialisé. En effet, chaque champ est lui-même un type simple et doit être traité comme tel. Un champ constant non initialisé à la déclaration ne pourra plus être modifié :

```
struct Type v = {.a = 42, .c = 'z'}; // pas d'erreur

struct Type v_e;
v_e.a = 42; // erreur de compilation
```

Dans le cas où un type complexe est *déclaré* comme constant, chacun de ses champs l'est :

```
const struct Type v;
v.b = 3.14; // erreur de compilation, chacun des champs est une constante
```

## Protection des pointeurs

Les pointeurs pointent vers un espace mémoire quelconque<sup>26</sup>. Cet espace peut être déclaré constant ou non<sup>27</sup>. En cela, l'application d'un `const` sur un pointeur peut être ambiguë. Il peut s'agir de l'espace pointé qui est constant, ou du pointeur lui-même qui doit ne pas pouvoir pointer ailleurs. Prenons le programme suivant :

```
int a = 0;
int b = 1;
const int* p = &a; // pointeur vers un 'const int'
*p = 42; // erreur
p = &b; // pas d'erreur, car le pointeur lui-même n'est pas constant
```

Il faut y penser comme avec des parenthèses “virtuelles” :

```
int a = 0;
(const int)* p = &a; // parenthèses juste pour visualiser, mais ne compile pas
```

L'utilisation de pointeurs vers des espaces constants permet de sécuriser certains codes en indiquant que le pointeur est immuable.

Par exemple, dans le cas d'un parcours de tableau :

26. Sans blague !

27. Malgré des apparences trompeuses, ceci n'est pas un concours de lapalissades.

```
// affichage ne doit pas modifier tableau
void array_display(const int array[], unsigned int length);

// tri peut potentiellement modifier tableau
void array_sort(int array[], unsigned int length);
```

**Remarque :** Il est inutile d'indiquer qu'un paramètre est constant ou non s'il ne s'agit pas d'un pointeur. En effet, l'argument sera copié, donc sa modification dans la routine ne modifie en rien sa valeur à l'appel.

Par ailleurs, passer en argument d'une routine un pointeur vers un espace constant à un paramètre non constant provoque une erreur :

```
1  const int somestats[42] = {...};
2
3  array_sort(somestats, 42); // erreur
```

On a pas l'assurance que l'espace mémoire de *somestats* reste constant par la routine *array\_sort*. Le compilateur relève une erreur pour assurer au mieux le bon comportement du programme.

Si on veut que le pointeur lui-même soit constant, ce n'est pas son type qui doit être indiqué comme constant, mais le pointeur lui-même :

```
int a = 0;
int b = 1;
int* const p = &a; // pointeur 'const' vers un 'int'
*p = 42; // pas d'erreur
p = &a; // erreur car p est constant
```

Par exemple, dans le parcours d'une liste doublement chaînée avec un nœud virtuel<sup>28</sup>, le pointeur vers le nœud virtuel peut être déclaré lui-même constant :

```
struct LinkedList {
    struct Noeud *const vhd; // virtual head
    unsigned int length;
};
```

On évite alors dans toutes les manipulations de la liste de l'invalider totalement par la modification du pointeur de nœud virtuel.

## Passer outre const

On aborde ici le point le plus délicat du mot-clé `const`, c'est-à-dire l'absence totale d'assurance de la constance des espaces mémoires déclarés et voulus comme tels.

Par exemple, le compilateur relève une erreur sur un code de cet acabit :

<sup>28</sup>. L'idée est d'ajouter une sécurité de programmation et d'optimiser les accès à la tête et à la queue. Le pointeur virtuel n'est jamais NULL et il y a donc une condition de moins à vérifier.

```

1  | const int x = 42;
2  | x = 5;

```

pourtant, il est aussi possible de modifier la valeur de  $x$  en passant par son adresse :

```

| int *y = &x;
| *y = 5; // avertissement à la compilation

```

C'est pour cette raison que `const` est seulement un déclarateur d'*intention*, et ne force en rien le programmeur dans l'absolu.

Pire que cela, il est possible d'éviter ces avertissements en écrivant du code moins propre :

```

1  | #include <stdio.h>
2  | #include <stdlib.h>
3  |
4  | struct Type {
5  |     const int a;
6  |     float b;
7  |     const char c;
8  | };
9  |
10 | int main() {
11 |     struct Type v = {.a = 5, .c = 'a'};
12 |     int *p = &v.a; // avertissement à la compilation
13 |     p = (int*)((&v.a)); // aucun avertissement (gcc comme clang)
14 |     return EXIT_SUCCESS;
15 | }

```

En effet, les capacités d'analyse *sémantique* du compilateur sont limitées<sup>29</sup>. L'idée du mot-clé `const` est seulement d'indiquer pour le programmeur consciencieux ce qui est en droit d'être modifié et ce qui ne devrait pas l'être. Le compilateur utilise ces informations pour tenter d'empêcher le programmeur d'écrire n'importe quoi et pour optimiser le code.

L'optimisation du code du fait de l'utilisation du mot-clé `const` a une conséquence importante : la modification d'un espace mémoire précisé constant induit un comportement indéfini du programme si celui-ci est optimisé.

Ainsi, le mot-clé `const` sert d'aide pour aider le programmeur à limiter les erreurs mais ne constitue rien une assurance absolue du bon comportement du programme. Il s'agit seulement d'une assurance que le comportement du programme est indéfini si une constante est modifiée. C'est une assurance *grammaticale*.

C'est aussi un indicateur pour un lecteur tiers du comportement du programme. Une routine dont un paramètre pointeur est déclaré `const` "assure" à l'utilisateur de celle-ci qu'elle ne modifiera pas l'espace mémoire pointé.

29. On en déduit que sont aussi limités ses capacités d'optimisation qui sont directement liés.

### 7.7.2 restrict

#### Définition 39 : Aliasing

On considère une donnée  $D$  accessible par symbole  $S$ . Un *alias*  $A$  de  $S$  est un second symbole qui accède très exactement à la même donnée  $D$ . Ainsi l'écriture de  $D$  en  $D'$  *via*  $A$  entraîne que la lecture de cette donnée *via*  $S$  renverra  $D'$ , et inversement. En C,  $A$  et  $S$  sont deux pointeurs vers un même espace mémoire.

Si un espace mémoire est modifié dans par une instruction et que cet même espace mémoire est lu depuis une seconde instruction, le comportement de la seconde instruction et des instructions utilisant son résultat est entièrement dépendant de la toute première instruction.

Si un bloc d'instruction n'a pas la certitude que les espaces mémoires manipulés par deux instructions différentes sont différents, de nombreuses optimisations sont perdues car il faut traiter une possible dépendance.

Prenons par exemple la routine suivante :

```

1 void swap(int *x, int *y) {
2     *x = *x ^ *y;
3     *y = *x ^ *y;
4     *x = *x ^ *y;
5 }
```

Cette routine échange les valeurs présentes dans les espaces mémoires pointés par  $x$  et  $y$ . Toutefois, si  $x = y$ , on a  $*x = 0$  après la première instruction et  $*y = 0$  après la seconde. Le comportement est donc incorrect si  $x$  est un alias de  $y$  et  $*x \neq 0$

Il faut corriger cette erreur par :

```

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 }
```

Ce branchement conditionnel ralentit considérablement l'exécution de cette routine dans le cas général. Il faudrait que la vérification n'ait lieu que dans les cas particuliers où il est possible que  $x = y$ .

Le mot-clé **restrict** est un déclarateur d'*intention* qui permet d'affirmer à la déclaration d'un pointeur que celui-ci est le seul à pointer vers son espace mémoire, c'est-à-dire qu'il n'existera jamais pendant la durée de vie du pointeur d'alias à celui-ci.

La syntaxe est similaire à celle de **const** sur les pointeurs eux-mêmes :

```

1 {
2     void *restrict ptr = ...;
```

```

3 | // déclaration qu'aucun alias de ptr n'est présent dans ce bloc.
4 | }

```

Ce déclarateur d'intention autorise surtout l'optimisation par le compilateur qui croit en la rigueur du programmeur. L'utilisateur des pointeurs est alors optimale. Par ailleurs, utiliser le mot-clé **restrict** *indique* au programmeur qu'il faut faire attention à ne pas utiliser d'alias, dans la déclaration d'une routine par exemple.

La routine d'échange précédente devient :

```

1 | void swap(int *restrict x, int *restrict y) {
2 |     *x = *x ^ *y;
3 |     *y = *x ^ *y;
4 |     *x = *x ^ *y;
5 | }

```

**Remarque :** contrairement à **const**, le compilateur n'essaie pas d'avertir ou de relever d'erreurs relatives à une incohérence sémantique du programme. En effet, à l'exception de cas très particuliers, les pointeurs rendent inutile l'analyse grammaticale du programme pour en déduire une analyse sémantique utile.

Ainsi :

```
| swap(&x, &x);
```

ne provoque aucune erreur et le comportement est tout à fait *incorrect* vis-à-vis de ce qui est souhaité par le programmeur. Mais le mot-clé **restrict** a “prévenu” le programmeur que l'appel `swap(&x, &x);` provoque un comportement indéterminé.



## CONSTRUCTION DE LITTÉRAUX



### 7.8.1 Motivation

Dans la partie sur les chaînes de caractères ont été abordés les chaînes littérales, c'est-à-dire l'écriture de tableaux de caractères initialisés par le programmeur mais déclarés par le compilateur :

```

printf("Hello World !\n");
// est strictement équivalent à :
const char _anonym_array[15] = {'H', 'e', 'l', 'l', 'o', ' ', '!', '\n', '\0'};
printf(_anonym_array);

```

L'objectif est de montrer la généralisation la construction de littéraux pour des types quelconques.

## 7.8.2 Syntaxe

Les littéraux sont des variables dites *anonymes*, c'est-à-dire qu'elles ne possèdent pas d'identifiants. On utilise pour les initialiser les *listes d'initialisation*, qui ont déjà été vues pour l'initialisation de tableaux ou de structures :

```
struct Point {
    double x, y, z;
};

int x[5] = {0, 1, 2, 3, 4};
//      liste d'initialisation

struct Point p = {.x = 0.5, .y = 0.7, .z = 1.5};
```

En fait, on peut aussi utiliser les listes d'initialisation pour initialiser d'autres variables que des tableaux<sup>30</sup> :

```
double pi = {3.14159};
```

La syntaxe pour définir un littéral est alors la suivante :

```
(TYPE){liste}
```

Cette notation a le même comportement que :

```
TYPE ANONYM = {liste};
```

Il est à noter qu'un littéral est une *lvalue*, c'est-à-dire que l'expression renvoie l'identité de l'objet. Ainsi, on peut écrire :

```
(int[5]){0, 1, 2, 3, 4}[0] = 1; // bon... inutile car l'adresse du tableau est perdue ensuite
```

## 7.8.3 Cas d'utilisation

On peut utiliser les littéraux pour passer des arguments sans créer d'objets spécifiques. Cela peut être utile pour ne pas surcharger l'espace des identifiants dans le programme.

```
1  #include <stdlib.h>
2  #include <stdio.h>
3
4  struct Point {
5      double x, y, z;
6  };
7
8  void display_point(struct Point p) {
9      printf("(%lf, %lf, %lf)\n", p.x, p.y, p.z);
```

30. Bien que la notation soit lourde pour aucune raison...

```

10 }
11 int main() {
12     for (unsigned int i = 0; i < 10; i++) {
13         display_point((struct Point){.x = (double)i, .y = 1.2, .z = -0.4});
14     }
15     return EXIT_SUCCESS;
16 }

```



## POINTEURS DE ROUTINES



### 7.9.1 Motivation

Considérons la structure suivante :

```

1 struct Point {
2     double x, y;
3 };

```

et un tableau de points :

```

| struct Point *array = malloc(10*sizeof(struct Point));

```

On se pose la question du tri du tableau *array*. Il est nécessaire pour effectuer un tri de poser une relation d'ordre sur les éléments du tableau. Si  $p = (x, y) \in \mathbb{R}^2$ , l'ordre qui vient en premier à l'esprit est l'ordre lexicographique, qui consiste à trier d'abord les points selon  $x$ , puis selon  $y$ , qui peut être implémenté par :

```

// -1 <=> a > b
// 0 <=> a == b
// 1 <=> a < b
char cmp_x_y(struct Point a, struct Point b) {
    if (a.x == b.x) {
        if (a.y == b.y) {
            return 0;
        } else {
            return (a.y > b.y) ? -1 : 1;
        }
    } else {
        return (a.x > b.x) ? -1 : 1;
    }
}

```

Cependant on peut aussi envisager le deuxième ordre lexicographique qui trie d'abord les points selon  $y$  puis selon  $x$ , dont le prototype serait le suivant :



```
// -1 <=> a > b
// 0 <=> a == b
// 1 <=> a < b
char cmp_y_x(struct Point a, struct Point b);
```

La question devient alors : *Comment préciser à une fonction de tri de tableau la relation à utiliser, sans avoir à programmer deux fonctions de tri – potentiellement longues – ?*

Il serait agréable de simplement passer la relation d'ordre en argument à la procédure de tri :

```
struct Point *array = malloc(10*sizeof(struct Point));

...

sort_proc(array, 10, cmp_x_y);
// OU
sort_proc(array, 10, cmp_y_x);
```

#### Définition 40 : Fonction d'ordre supérieur

Une fonction d'ordre supérieur, en mathématiques, est une fonction qui a comme espace d'arrivée ou de sortie un espace de fonctions.

La possibilité d'utiliser des fonctions d'ordre supérieur en informatique est très intéressante puisqu'elle permet par exemple de programmer la fonction de composition de fonctions, ou encore une procédure qui appliquerait une même procédure à chaque élément d'un conteneur (liste, tableau, etc...)

Le langage C n'est cependant pas un langage fonctionnel, c'est-à-dire qu'il n'est pas pensé pour la manipulation de fonctions d'ordres supérieurs. Sans parler de la syntaxe relativement immonde, la construction des fonctions à la compilation complique sérieusement la création de fonctionnelles <sup>31</sup>.

### 7.9.2 Syntaxe

Une routine en mémoire est une simple adresse, stockée comme l'est une variable statique. L'unique différence est que l'adresse d'une routine est en section de code, alors que l'adresse d'une variable statique est en section de données. Il est donc tout à fait possible d'utiliser un pointeur de routines, et même de définir des tableaux de routines. Par contre, il est extrêmement difficile de construire des fonctions à la volée durant l'exécution du programme <sup>32</sup>.

La syntaxe pour déclarer un pointeur de routine est la suivante :

```
// T1, ..., Tn sont les types des paramètres
TYPE (*ptr)(T1, ..., Tn); // TYPE est le type de retour
```

Ainsi, le pointeur vers la relation d'ordre de l'exemple précédent serait déclaré comme suit :

31. Il est nécessaire de passer par des *macros* (voir section 7.10 suivante), ce qui en dit long sur l'absence de support natif pour la programmation fonctionnelle.

32. Ce qu'on appelle de la compilation JIT (*Just In Time*), qui consiste à faire écrire et exécuter un programme par le programme principal lui-même. La compilation JIT est utilisée pour l'interpréteur Python par exemple.

```
char (*cmp_ptr)(struct Point, struct Point);
```

Comme une routine est déjà une adresse, on peut initialiser directement :

```
1 cmp_ptr = cmp_x_y;
2 // OU
3 cmp_ptr = cmp_y_x;
```

Une fonction de tri par sélection<sup>33</sup> peut alors être implantée de la manière suivante :

```
1 // definitions de struct Point, cmp_x_y et cmp_y_x ici.
2
3 void swap(struct Point array[], unsigned int i, unsigned int j) {
4     struct Point tmp = array[i];
5     array[i] = array[j];
6     array[j] = tmp;
7 }
8
9 // le pointeur vers 'r' ne doit pas être modifié (relation modifiée) durant la fonction i_min
10 unsigned int i_min(const struct Point a[], unsigned int l, char (*const r)(struct Point, struct Point)) {
11     unsigned int m = 0;
12     for (unsigned int i = 1; i < l; i++) {
13         if (r(a[i], a[m]) == 1) { // a[i] < a[m] selon la relation donnée
14             m = i;
15         }
16     }
17     return m;
18 }
19
20 // idem
21 void selection_sort(struct Point a[], unsigned int l, char (*const r)(struct Point, struct Point)) {
22     for (unsigned int i = 0; i < l; i++) {
23         for (unsigned int j = i; j < l; j++) {
24             unsigned int m = i_min(a + i, l - i, r);
25             swap(a, i, m + i);
26         }
27     }
28 }
29
30 int main() {
31     ...
32
33     struct Point arr[10] = {...};
34     selection_sort(arr, 10, cmp_x_y);
35
36     ...
37 }
```

33. Allons au plus simple, même si le code n'est pas "efficace" en soi

### 7.9.3 Tableaux de routines

On peut faire des tableaux de routines de même qu'on définirait des tableaux de pointeurs :

```
#include <stdio.h>

double add(double a, double b) {return a + b;}
double sub(double a, double b) {return a - b;}
double mul(double a, double b) {return a * b;}
double div(double a, double b) {return a / b;}

int main() {
    ...

    double (*operators[4])(double, double) = {add, sub, mul, div};
    printf("%lf\n", operators[1](2, 4)); // 2 - 4 = -2

    ...
}
```

### 7.9.4 Exercices

**Exercice 54 (Mapping) [10].**

1. Écrire une fonction `void array_map(int array[], unsigned int length, void (*const f)(int*))`; qui applique une procédure  $f$  à chacun des éléments du tableau `array`.
2. Reprendre l' **Exercice 48** et implanter une fonction `void linkedlist_map(struct LinkedList *l, void (*const p)(int*))`; qui applique la procédure  $p$  à chacun des éléments de la liste chaînée.



## DIRECTIVES DU PRÉPROCESSEUR (2)



Le préprocesseur C est un programme qui intervient avant le compilateur dans le traitement d'un fichier de code pour le transformer en exécutable. Le préprocesseur n'effectue pas de *compilation* à proprement parler puisqu'il ne traduit pas le langage du code source. Il ne fait que préparer et simplifier les fichiers de code en vue de leur compilation.

Le préprocesseur effectue une série de transformations textuelles sur un fichier traité. Celles-ci se produisent avant tout autre traitement. Conceptuellement, l'ensemble du fichier traité est exécuté à travers chaque transformation avant que la suivante ne commence. Dans la pratique, le préprocesseur effectue toutes les transformations en même temps pour des raisons de performances.

Ces "phases" de transformation sont les suivantes :

1. Initialisation :
  - le fichier en entrée est chargé en mémoire et découpé en lignes.
  - si les digraphes et trigraphes<sup>34</sup> sont autorisés, ils sont remplacés par le caractère simple correspondant

---

34. Voir 7.13

- les lignes continues sont fusionnées une seule ligne
  - tous les commentaires sont supprimés et remplacés par un espace
2. Identification des symboles : chaque ligne est découpé en symboles qui seront la base de l'analyse dans la suite
  3. Certains symboles définissent des directives de préprocesseur, ou des *macros* qui doivent être remplacés par le texte correspondant<sup>35</sup>.

Cette section vise à préciser en détails les directives de préprocesseurs déjà introduites en section 5.5 et à les compléter avec les directives de définition de *macroinstructions* (appelées simplement *macros* par la suite).

On peut observer le résultat du traitement du préprocesseur en exécutant l'instruction :

```
> gcc main.c -E -o main.preprocessed # génère un fichier texte
> cat main.preprocessed # affichage du fichier texte
... (assez long)
```

### 7.10.1 Inclusion de fichiers externes

On revient sur l'instruction du préprocesseur C `#include` :

```
#include "nom_de_fichier_quelconque"
#include <nom_de_fichier_quelconque>
```

Cette instruction permet de copier un code d'un fichier externe dans un fichier de code. Elle est principalement utilisée pour copier des prototypes et des définitions de structure, comme cela a été vu dans la section 6.13 sur la modulation.

Son usage peut toutefois être généralisé :

filename.ext

```
1 | const char t[] = "Hello World !\n";
```

main.c

```
1 | #include <stdio.h>
2 | #include <stdlib.h>
3 |
4 | int main() {
5 |     #include "filename.ext"
6 |     printf("%s", t); // Hello World !\n
7 |     return EXIT_SUCCESS;
8 | }
```

Ce type de code ne doit être utilisé que dans des cas particuliers.

<sup>35</sup>. Le traitement de ces symboles est ce qui est communément appelé le *preprocessing* bien qu'il ne s'agisse en réalité que de la dernière étape.

**Exemple :** dans le cas de la programmation en C d'applications pour les calculatrices *Casio*, les images dessinés sur l'écran ne peuvent être stockés dans un fichier externe au programme<sup>36</sup> et doivent être programmés “en dur” dans le programme. L'image peut être représentée par du code C :

```
1 | const unsigned char image[] = {  
2 |     // octets de couleur des pixels de l'image  
3 | }
```

Ces fichiers images peuvent être très long. Pour ne pas encombrer le code avec des données, on peut écrire ce tableau de pixels dans un fichier externe “*image.c*” et l'inclure dans le programme :

```
| #include "image.c"
```

**Remarque :** cette technique est particulièrement utile quand des données ou des définitions sont nécessaires à plusieurs endroits du programme, et potentiellement dans des fichiers sources différents.

On peut se défendre des inclusions multiples grâce à des garde-fous (voir section 6.13 sur la modulation).

## 7.10.2 Compilation conditionnelle

### Motivation

La compilation conditionnelle a été abordée succinctement dans la section 6.13.6 sur les garde-fous.

Elle permet de choisir au moment de la compilation quel région du code sera compilée ou non. Cela est particulièrement intéressant quand le programme dépend fortement de son environnement d'exécution. Prenons l'exemple d'un logiciel présentant une interface graphique. Selon le système d'exploitation sur lequel ce programme s'exécute, le système de fenêtrage va différer. On utilisera *X* ou *Wayland* sous Linux, tandis que sous Windows, le fenêtrage est directement intégré au système d'exploitation. Le code C diffère donc largement dans chacun des cas.

### Syntaxe

La compilation conditionnelle en C se base sur la syntaxe suivante :

```
| #if condition  
| // INSTRUCTIONS  
|  
| #else // optionnel  
| // INSTRUCTIONS  
|  
| #endif
```

On peut naturellement imbriquer des conditions entre elles. On peut raccourcir :

```
| #if condition  
| // instructions
```

---

36. ou difficilement

```
#else
    #if condition
        // instructions
    #endif
#endif
```

par :

```
#if condition
// instructions
#elif condition
// instructions
#endif
```

### Expression d'une condition

Les conditions préprocesseurs utilisent les mêmes opérateurs relationnels qu'en C (`||`, `&&`, `!`) et les mêmes opérateurs de comparaison.

On y ajoute l'opérateur unaire `defined`, spécifique au préprocesseur. Cet opérateur renvoie *Vrai* si le symbole qui le suit est défini, et *Faux* sinon :

```
#if !(defined MODULE_H_INCLUDED) // <=> #ifndef MODULE_H_INCLUDED
#define MODULE_H_INCLUDED

// interface du module

#endif
```

**Remarque :** on a introduit dans la section 6.13.6 sur les garde-fous les instructions raccourcies `#ifdef` et `#ifndef`

### 7.10.3 Diagnostiques

Les instructions de diagnostic permettent de faire échouer la compilation ou d'avertir dans le cas de soucis potentiels d'exécution au moment de la compilation. Ces avertissements et erreurs provoqués sciemment par le préprocesseur peuvent par exemple permettre d'indiquer très précisément si une bibliothèque de code est manquante ou si une bibliothèque n'est pas à jour lors de l'inclusion. En effet, les bibliothèques contiennent généralement une macro de donnée indiquant la version de la bibliothèque. On peut aussi par exemple relever une erreur si une certaine macro de donnée est définie, si le programme ne fonctionne pas dans un certain environnement.

Pour une erreur on utilise le mot-clé `#error` :

```
#ifndef SDL_VERSION
#error "Please include SDL for the application to work."
#else
// ...
#endif
```

Et selon la même syntaxe, on utilise le mot-clé `#warning` pour lever un avertissement :

```
// le "..." ci-dessus :  
#if defined SDL_MAJOR_VERSION && SDL_MAJOR_VERSION < 3  
#warning "The application does not officially support SDL versions 1.X and 2.X."  
#endif
```

### 7.10.4 Pragma

L'instruction `#pragma` permet d'exécuter des directives spécifique au compilateur. Ainsi, il n'est jamais assuré qu'un programme utilisant `#pragma` soit portable entre tous les systèmes d'exploitation et compilateurs. Il s'agit d'une instruction à éviter le plus possible.

On ne décrit que l'instruction `#pragma` la plus courante : `#pragma once`.

#### `#pragma once`

La directive `#pragma once` indique que le fichier lu ne devra jamais être lu à nouveau durant la compilation. Il s'agit d'une version moins portable des garde-fous décrits en section 6.13 :

```
#pragma once  
  
// contenu du fichier
```

Il faut toutefois noter que cette directive est la plus reconnue parmi les directives non standards, et est utilisable avec la plupart des compilateurs les plus importants (comme `gcc` ou `clang`).

L'avantage principal de `#pragma once` est d'éviter l'erreur humaine d'utiliser plusieurs garde-fous de même nom comme cela peut être possible dans de gros programmes. Le préprocesseur s'occupe lui-même de générer un symbole de garde-fou.

Le problème principal qui empêche la standardisation de `#pragma once` est la forte dépendance d'une telle instruction avec le système de fichiers utilisé et le compilateur. En effet, un même fichier physique peut se trouver dans plusieurs répertoires différents d'un système de fichiers si celui-ci le lie physiquement ou virtuellement en un autre point. Les fichiers peuvent aussi être différenciés de plusieurs manières selon le compilateur.

À éviter en général comme toutes les autres instructions `#pragma`.

### 7.10.5 Macros

L'instruction `#define` n'a été utilisée depuis le début de ce cours que pour définir des symboles en tant que tels ou des constantes associés à des symboles :

```
#define JUST_A_MEANINGLESS_SYMBOL  
#define PI 3.14159
```

Le mot-clé `#define` permet beaucoup plus que de définir des constantes.

Il existe deux types de macros :

- les macros de données
- les macros fonctionnelles

## Macros de données

Les macros de données sont les macros utilisées depuis le début de ce cours. Il s'agit d'une correspondance simple entre un symbole et un fragment de code. À la suite de la définition d'un symbole par un texte, chaque apparition de ce symbole dans le fichier source est remplacé par le texte correspondant.

Ces macros sont utilisés comme de simples données textuelles dans le code où elles apparaissent.

Les symboles définis peuvent très bien être des mots-clés du C, puisque le préprocesseur ne sait rien du langage C lui-même. Les mots-clés sont remplacés par leur correspondance, comme n'importe quels autres symboles définis. Ainsi, le programme suivant est tout à fait valide :

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  #define float int
5
6  int main() {
7      float x = 3.14;
8      printf("%d\n", x); // affiche 3 sans avertissements
9      return EXIT_SUCCESS;
10 }
```

On peut définir des textes plus longs :

```
1  #include <stdio.h>
2
3  #define SALUT printf("Salut !!!\n");
4  #define FIN printf("Au revoir !\n"); return 0;
5  #define DIRE_SALUT_ET_PARTIR SALUT FIN
6
7  int main() {
8      DIRE_SALUT_ET_PARTIR
9  }
```

## Macros fonctionnelles

Les macros fonctionnelles sont des macros contenant des instructions qui peuvent être utilisées comme des fonctions. Elles peuvent prendre en paramètre du texte et produisent en sortie du code C :

```
#define MACRO_ID(param1, param2, etc...) <sortie>
```

On peut par exemple utiliser les macros pour définir des routines *inline*, c'est-à-dire exécutés sans appel de routine. Cela permet d'accélérer l'exécution, puisqu'il n'y a pas besoin de pousser sur la pile d'exécution les paramètres.

On notera que les paramètres ne peuvent pas être typés :



```

1  #include <stdio.h>
2  #include <stdlib.h>
3
4  #define MAX(A, B) ((A) > (B) ? (A) : (B)) // ne dépend ni du type de 'a' ni de celui de 'b'
5
6  int main() {
7      int a = 2;
8      int b = 5;
9      printf("%d\n", MAX(a, b)); // 5
10     double x = 3.14;
11     float y = 1.414;
12     printf("%f\n", MAX(x, y)); // 3.140000
13     return EXIT_SUCCESS;
14 }
```

Par ailleurs, les arguments de la macro ne sont pas copiés. C'est le code de la macro qui est copié à l'endroit de l'appel. Cela signifie que la modification d'un argument dans la macro est une modification de cet argument *tout court* :

```

1  #include <stdio.h>
2  #include <stdlib.h>
3
4  #define SYRACUSA(X) X = (!(X%2)) ? (X >> 1) : (X * 3 + 1);
5
6  int main() {
7      int n = 53;
8      SYRACUSA(n);
9      printf("%d\n", n); // 160
10     return EXIT_SUCCESS;
11 }
```

### 7.10.6 Opérateurs de macros sur les symboles

Le préprocesseur permet la manipulation des symboles du texte. Il permet la définition de données associées à des symboles *via* les macros, et ces symboles sont remplacés par les données correspondantes dans le texte.

Il semble manquer toutefois la possibilité de manipuler les symboles en tant que tels dans une macro. On observe que si une telle utilisation peut avoir lieu, les symboles conserveront par nature un caractère statique. On peut imaginer une conversion du symbole en une chaîne de caractère littérale correspondante, ou la possibilité de construire un nouveau symbole sur la base de précédents.

#### stringizing

Le préprocesseur C offre exactement ce service, en anglais nommé *stringizing*, traduit littéralement par la périphrase “construction de chaîne”, en un mot *cordelage*. On conservera la dénomination anglaise, plus parlante. On a vu que le symbole `#` permet d'introduire les directives de préprocesseur. Mais cela n'est vrai que hors des macros. Dans une macro, le `#` perd cette fonctionnalité. Il en gagne cependant une seconde, qui permet à une macro d'interagir avec les symboles passés en argument.

Dans une macro, le # suivi d'un argument renvoie la chaîne de caractère représentant exactement le texte-argument. L'intérêt d'une telle fonction provient de l'inefficacité du code suivant :

```
1 | #define AFFICHER(texte) printf("texte""\n") // affiche texte\n quelque soit la valeur de l'argument
```

Le # permet de mettre entre guillemets le texte de l'argument de la macro :

```
1 | #include <stdio.h>
2 | #include <stdlib.h>
3 |
4 | #define AFFICHER(texte) printf(#texte "\n")
5 |
6 | int main() {
7 |     AFFICHER(Salut les pingouins !!!); // affiche Salut les pingouins !!!\n
8 |     return EXIT_SUCCESS;
9 | }
```

On peut écrire ainsi des macros utilitaires :

```
1 | #define WARN(expression) fprintf(stderr, "Avertissement : "#expression"\n")
2 |
3 | ...
4 |
5 | WARN(x < 0); // Avertissement : x < 0\n
```

**Remarque :** Le # a *exactement* le comportement de mise entre guillemets du texte-argument. Ainsi l'appel suivant :

```
| AFFICHER(\n)
```

affiche bien un retour à la ligne.

## Concaténation de symboles

L'opération fondamentale que l'on peut opérer sur des chaînes de caractères est leur concaténation, c'est-à-dire la mise bout-à-bout de ces chaînes. On peut ainsi à partir d'un alphabet donné construire n'importe quel texte sur cet alphabet.

Par exemple, avec l'alphabet  $\Sigma = \{'0', '1'\}$ , on peut écrire '01110' qui est la concaténation d'un zéro, de trois uns et d'un zéro.

Le préprocesseur C offre la possibilité de concaténer deux symboles dans une macro grâce à l'opérateur ## :

```
1 | #define AB int
2 | #define C A##B // <=> #define C int
3 |
4 | #define PRINT(text) pr##intf(#text "\n") // <=> printf(...)
```

**Remarque :** cet opérateur, tout comme l'opérateur de *stringizing*, n'est disponible que dans les macros. Ainsi, le code suivant ne compile pas :

```
| printf("Salut les pingouins !\n"); // error: stray '##' in program
```

## Exemple d'application concret

Cet exemple, pertinent, est extrait directement de la documentation du préprocesseur C.

Supposons qu'on écrive un programme qui définisse plusieurs procédures de commandes dans un tableau :

```
// définitions de quit_command et de help_command

struct Commande {
    char *name;
    void (*procedure)(void);
};

struct Commande commands[] = {
    {"quit", quit_command}, // name = "quit", procedure = quit_command
    {"help", help_command} // idem
};
```

Le “souci” ici est d’avoir à écrire deux fois le nom de la procédure lors de l’initialisation du tableau, à la fois pour la chaîne de caractère et à la fois pour le nom de la procédure elle-même.

On peut simplifier<sup>37</sup> l’initialisation par une macro fonctionnelle utilisant à la fois le *stringizing* et la concaténation :

```
// proc_name est remplacé dans la concaténation par la valeur de l'argument
#define COMMAND(proc_name) {#proc_name, proc_name##_command}

struct Command commands[] = {
    COMMAND(quit),
    COMMAND(help)
};
```

### 7.10.7 Exercices

**Exercice 55 (Tout dépend du système) [07].** Le compilateur définit par défaut certaines constantes selon l’environnement de compilation. Le site <https://sourceforge.net/p/predef/wiki/OperatingSystems/> liste les macros définies selon le système d’exploitation.

Écrire un programme qui affiche “Chouette !” si le programme est compilé et exécuté sous Linux et affiche “Beeerk !” si le programme est compilé et exécuté sous Windows. Le cas de MacOS n’est pas traité car ce système d’exploitation n’existe pas.

**Exercice 56 (Libération sécurisée) [08].** Définir une macro `safe_free(p)` qui libère la mémoire associée à un pointeur `p` et le réinitialise à `NULL`.

37. Dans le cas où il y aurait une très grande quantité de procédures



# ROUTINES VARIADIQUES

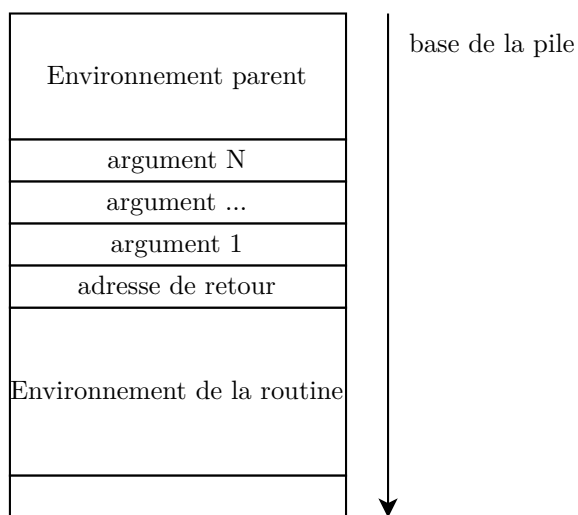


## 7.11.1 Motivation

Comprendre comment la fonction `printf` du C permet de prendre un nombre d'arguments variable à son appel, parce-que j'espère que tous les lecteurs arrivés à ce point se sont posés la question... Ah! Et faire pareil aussi, ce serait chouette ^^

## 7.11.2 Appels de routines

Il existe plusieurs standards d'appel de routines. Lors d'un appel de routine *suivant le standard du langage C*<sup>38</sup>, les arguments sont poussés sur la pile suivant le schéma simplifié suivant :



Ils sont ensuite dépilés lors de la sortie de la routine, pour revenir à l'environnement précédent l'appel.

Le compilateur doit donc savoir à la compilation combien d'arguments sont présents lors de l'appel pour pouvoir les empiler. De même pour pouvoir y accéder à l'intérieur d'une routine. Le nombre d'arguments ne peut donc être dynamique et déterminé par une variable du programme. Il faut qu'à l'appel de la routine, le nombre d'arguments soit déterminé. Il faut également que la taille en octets de chacun des arguments puisse être déterminé à la fois à la compilation et à la lecture de ceux-ci dans la routine.

C'est le cas des appels à `printf` :

```
// nombre et taille des arguments connues à la compilation
// la première chaîne de caractères donne ces informations à la routine
printf("%d + %d = %d\n", a, b, a + b);
```

Le nombre d'arguments à fournir n'est pas donné par la routine intrinsèquement. Il est donné par le programmeur, qui doit tout de même se conformer à la spécification de la routine.

<sup>38</sup>. Pour des raisons d'optimisation, le compilateur peut utiliser d'autres standards, voir [https://en.wikipedia.org/wiki/X86\\_calling\\_conventions](https://en.wikipedia.org/wiki/X86_calling_conventions) pour des détails.

### 7.11.3 Syntaxe et module *stdarg*

Une routine variadique doit contenir *a minima* un paramètre nommé qui permette de déterminer le début des paramètres variadiques.

Pour définir une routine variadique, on utilise la syntaxe suivante :

```
| TYPE function(des paramètres déterminés, ...);
```

Les trois points ... ne sont pas là comme dans le reste du livre pour indiquer “mettre ce que l’on veut”. Il s’agit véritablement de la syntaxe permettant de définir des routines variadiques. Ainsi, on peut donner le prototype d’une fonction calculant la somme d’un nombre  $n > 0$  d’entiers :

```
| int variadic_sum(unsigned int n, ...);
```

qui peut par exemple être appelée par :

```
| int x = variadic_sum(3, 7, 9, -5); // x = 7 + 9 - 5
```

Intéressons nous maintenant à l’implantation d’une telle fonction `variadic_sum`.

Il est en théorie possible, en utilisant l’adresse du premier argument, de retrouver les arguments donnés à la routine. Il y a certains problèmes à cela :

- les conventions d’appel de routines et de passage d’arguments dépendent à la fois du processeur et du compilateur
- si on active les optimisations du compilateur, les conventions d’appel changent

Ainsi, le seul qui possède toutes les informations pour lire de manière certaine les arguments passés à la routine variadique est *le compilateur*.

Heureusement, celui-ci possède des routines intégrées capables d’initialiser et d’effectuer la lecture des arguments variadiques. Le module *stdarg* de la bibliothèque standard permet d’appeler ces routines intégrées, dites “*builtins*”<sup>39</sup>.

Il définit pour cela le type `va_list` qui permet de parcourir les arguments variadiques suivant un paramètre indiqué. Il s’agit en fait d’un pointeur vers les arguments. On utilise pour initialiser et fermer proprement la lecture des arguments de deux macros :

- `va_start(va_list args, PARAMETER_NAME)`
- `va_end(va_list args)`

```
#include <stdarg.h>

int variadic_sum(unsigned int n, ...) {
    va_list args;
    va_start(args, n); // on démarre la lecture des arguments après le paramètre 'n'

    // lecture/utilisation des arguments variadiques

    va_end(args);
}
```

39. <https://github.com/gcc-mirror/gcc/blob/master/gcc/ginclude/stdarg.h> pour le vérifier.

On dispose de deux macros pour lire les arguments :

- `va_arg(va_list args, TYPE)` : renvoie l'argument suivant, de type `TYPE`, et passe à l'argument suivant.
- `va_copy(va_list dest, va_list source)` : initialise une liste d'arguments *dest* comme une copie de la liste *source*

Ainsi, on peut écrire :

```
#include <stdarg.h>

int variadic_sum(unsigned int n, ...) {
    int sum = 0;
    va_list args;
    va_start(args, n); // on démarre la lecture des arguments après le paramètre 'n'
    for (unsigned int i = 0; i < n; i++) {
        sum += va_arg(args, int);
    }
    va_end(args);
    return sum;
}
```

### 7.11.4 Macros variadiques

Les macros fonctionnelles du préprocesseur peuvent aussi être variadiques et prendre un nombre quelconque de paramètre. La syntaxe et les possibilités sont beaucoup plus limitées que dans le cas des routines variadiques “classiques”.

La syntaxe est la suivante :

```
1 | #define VAR(named_args, ...) // code using __VA_ARGS__ symbol
```

En fait, moins qu’une macro à nombre indéfini de variables, il s’agit d’une possibilité de passer à la macro un argument qui contient des virgules.

L’argument “...” peut contenir des virgules et est nommé dans la macro par `__VA_ARGS__`.

Ainsi, on peut écrire :

```
1 | #define debug_printf(...) \
2 |     fprintf(stderr, "At line %d : ", __LINE__); \
3 |     fprintf(stderr, __VA_ARGS__); \
4 |     fprintf(stderr, "\n")
5 | // plus loin
6 | debug_printf("Dividing by %d", x);
```

### 7.11.5 Exercices

**Exercice 57 (À un doigt du zéro) [10].** Écrire une fonction `unsigned int mult(unsigned int first, ...)`; qui parcourt ses arguments jusqu’au premier 0 rencontré, et renvoie le produit de ces arguments, 0 exclu.

**Exercice 58 (Fonctions à paramètres par défaut) [20].**

On veut implanter en C le concept de paramètres par défaut. Il s'agit de définir une fonction dont certains paramètres, nommés, ont une valeur par défaut à l'appel et n'ont pas besoin d'être précisés. Par exemple, on peut imaginer une fonction d'addition avec retenue avec un paramètre  $r$  dont la valeur par défaut est 0. Idéalement, on voudrait pouvoir écrire :

```
1 | add_r(4, 5); // par défaut, retenue = 0 -> resultat = 4 + 5
2 | add_r(4, 5, .r = 1); // on modifie la valeur par défaut à 1 -> resultat = 4 + 5 + 1
```

1. Écrire une structure `add_r_opt` contenant un unique champ  $r$
2. Écrire une fonction `int add_r(int a, int b, add_r_opt opt);` dont le comportement est décrit ci-dessus.
3. Écrire une macro `add_r` qui se comporte exactement comme ci-dessus.
4. Faire de même pour la fonction `int atoi(char* string)` de l'Exercice 51 en ajoutant un paramètre *base* qui correspond à la base dans laquelle est écrit le nombre à convertir (*base* = 10 par défaut).



## ASSEMBLEUR ET C



Il est possible d'intégrer de l'assembleur directement dans du code C.

La programmation en assembleur nécessite elle-même un ouvrage complet, voire plusieurs selon l'architecture de l'ordinateur. Comme l'insertion de code d'assembleur en C nécessite des compétences préalables en assembleur, et que flemme, ce chapitre finit au point de cette phrase.



## 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.

### 7.13.1 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>40</sup>.

Cela ouvre des possibilités pour écrire des codes très moches.

### 7.13.2 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}
```

### 7.13.3 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 |
```

40. 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>41</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>

```

### 7.13.4 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

```

41. 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, ne fonctionnant que sur architecture petit-boutiste, à compiler par la ligne de commande :

```

user@computer ~/working_directory> gcc main.c -o main -std=c11
user@computer ~/working_directory> ./main 1
1
user@computer ~/working_directory>

```

```

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     /* NE PAS SUPPRIMER CETTE LIGNE DE COMMENTAIRE */
13     int** _; // Mais putain c'est quoi ce bordel ???/
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 }
```

Le défi est évidemment de comprendre ce que fait ce programme, et comment il le fait, hihhi! <sup>42</sup>

---

42. (rire vraiment très diabolique)

Partie III

PETITE  
PARENTHÈSE  
THÉORIQUE



# INTRODUCTION



*The aim of theory really is, to a great extent, that of systematically organizing past experience in such a way that the next generation, our students and their students and so on, will be able to absorb the essential aspects in as painless a way as possible, and this is the only way in which you can go on cumulatively building up any kind of scientific activity without eventually coming to a dead end.*<sup>43</sup>

– M.F. Atiyah, *How research is carried out*

Les deux premières parties de ce livre se sont concentrées sur des aspects assez pratique. Toutefois, on peut observer que certaines notions ont été définies assez rapidement et avec peu de précision, comme par exemple la notion d'*algorithme*. Par ailleurs, certains exercices comme l' **Exercice 41** pointent du doigt l'importance d'un certain aspect théorique mathématique, en particulier en ce qui concerne :

- la vitesse d'exécution des algorithmes selon la taille de l'entrée<sup>44</sup>
- la démonstration du comportement correct d'un algorithme

Le premier point a un intérêt pour des raisons purement pratico-pratique : on veut que les programmes informatiques s'exécutent le plus vite possible dans un contexte donné. Le second point a plus à voir avec certaines applications très précises, notamment pour ce qui est des programmes informatiques dits « à risques » comme les programmes informatiques présents sur les ordinateurs de bord d'avions. On peut aussi penser à la démonstration d'algorithmes pour être certain qu'un circuit microélectronique possède exactement le comportement souhaité<sup>45</sup>.

D'autres exercices comme l' **Exercice 40** et l' **Exercice 48** pointent du doigt le conditionnement de l'efficacité de nos algorithmes selon la manière dont on stocke les données. Dans le cadre d'une formation d'ingénieur, c'est ce point qui sera à retenir, puisque ses implications pratiques sont directes.

Cette partie va donc s'intéresser à présenter :

- la notion, très importante dans la pratique, de complexité, qui fournit un cadre théorique pour analyser les performances d'algorithmes
- les structures de données élémentaires qui conditionnent l'efficacité des algorithmes, ainsi que quelques algorithmes élémentaires sur ces structures
- les approches algorithmiques fondamentales pour la résolution de problèmes

Les exercices seront une opportunité de découvrir des cas d'applications concrets de ces structures de données et des extensions de celles-ci.

Dans le cadre de l'ISMIN, cette partie devrait suffire à apporter au lecteur toutes les connaissances théoriques fondamentales exigées durant le cursus<sup>46</sup> en ce qui concerne l'algorithmique.

---

43. Le but de la théorie est en réalité, dans une large mesure, de systématiquement organiser l'expérience passée de façon à ce que les prochaines générations, nos étudiants, leurs étudiants, et ainsi de suite, soient capables d'en absorber les aspects essentiels le moins douloureusement possible. Il s'agit là du seul moyen par lequel il est possible de construire par accumulation quelque sorte d'activité scientifique que ce soit sans en venir éventuellement à un point mort.

44. Dans cet exemple, deux types de vitesse d'exécution sont observées dans la correction : la vitesse en moyenne et la vitesse dans le pire des cas. On y reviendra dans la suite.

45. Dans les faits, on s'appuie pour cela d'automates temporels de propositions logiques, ce qui dépasse très largement le cadre de ce livre. C'était pour illustrer...

46. Voir parfois plus.

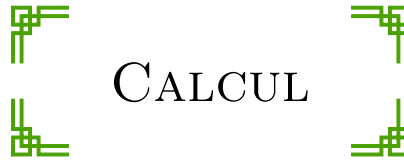
Il faut avoir conscience que tout ceci n'est qu'une *mise en bouche* qui présente les bases les plus basiques de l'informatique fondamentale/théorique.

**Lectures d'approfondissement :**

- Le cours d'informatique fondamentale de l'école Polytechnique [5], tout en restant introductif, couvre plus largement les notions autour des modèles de calcul, non abordées ici
- On pourra lire un approfondissement à propos de la notation de Landau [17] accessible sur *ArXive*
- Le Froidevaux [9] présente de manière systématique les types abstraits algébriques et les structures de données les plus communes.
- Les *Éléments d'algorithmique* [4] couvrent largement le programme d'algorithmique de licence d'informatique, et fournissent une vision bien plus complète des bases de l'algorithmie

Ces livres proposent eux-mêmes d'autres lectures complémentaires.

# COMPLEXITÉ



On définit ici le minimum de notions nécessaires pour être un minimum “précis” par la suite. Il ne s’agit que d’un minimum, donc de pas grand-chose. Faudra voir dans le futur pour développer dans les détails. Pour l’instant, c’est bâclé, notamment par l’absence de la description d’au moins un modèle de calcul. L’idée n’est pas de prouver et détailler tout dans les moindres détails mais quand même. . .

## Définition 41 : Étoile de Kleene usuelle

Soit  $E$  un ensemble. On note l’étoile de Kleene de  $E$  l’ensemble  $E^* = \bigcup_{d \in \mathbb{N}} E^d$

Si  $E$  comporte un neutre  $0_E$ , on notera  $E \setminus \{0_E\}$  l’ensemble privé du neutre <sup>a</sup>.

<sup>a</sup>. Au lieu du  $E^*$  habituel. . . il faut en vouloir à Kleene.

### 8.1.1 Modèles de calcul

## Définition 42 : Modèle de calcul

Un modèle de calcul est une structure mathématique munit d’un ensemble d’opérations élémentaires qui manipulent cette structure.

## Exemples <sup>1</sup> :

1. Faudra voir la motivation pour décrire formellement au moins un modèle de calcul, pour l’instant, ce n’est qu’une liste de noms simplement pour observer la multiplicité des modèles.

- Machines RAM
- Machines de Turing
- Systèmes de Post
- Lambda calcul
- Fonctions récursivement énumérables
- ...

## Problèmes et algorithmes

### Définition 43 : Problème

Un problème est une fonction  $P : X \rightarrow Y$ .

$X$  est l'ensemble des entrées du problème et  $Y$  l'ensemble des sorties du problème.

**Exemple :** Le problème consistant à trier par ordre croissant des listes d'entiers a comme ensembles d'entrées et de sorties  $X = Y = \mathbb{Z}^*$  (étoile de Kleene).

On choisit un élément  $x \in X = \mathbb{Z}^*$  comme par exemple  $(7, 9, -2, -7, 5, 4, -3, 1, 0) \in \mathbb{Z}^8 \subset \mathbb{Z}^*$  (par définition de l'étoile de Kleene).

C'est un problème de tri croissant, donc  $P((7, 9, -2, -7, 5, 4, -3, 1, 0)) = (-7, -3, -2, 0, 1, 4, 5, 7, 9)$ .

### Définition 44 : Problème de décision

Un problème  $P : X \rightarrow Y$  est dit *de décision* si  $Y = \mathcal{B} = \{\text{Vrai}, \text{Faux}\}$ .

**Exemple :** Déterminer si un élément  $e \in E$  appartient ou non à un tableau d'éléments d'un ensemble  $E$  est un problème de décision. Son ensemble d'entrées est  $X = E \times E^*$ .

### Définition 45 : Algorithme

Un algorithme qui résout un problème  $P$  dans un certain modèle de calcul est une suite d'instructions élémentaires de ce modèle qui pour tout  $x \in X$  en entrée calcule  $P(x)$ <sup>a</sup>.

C'est donc une réalisation effective du calcul de  $P$  selon un modèle de calcul donné.

<sup>a</sup>. La définition manque un peu de précision et de rigueur... Il faudrait définir précisément le modèle de calcul utilisé

**Notation :** Soit  $P : X \rightarrow Y$  un problème. L'ensemble des algorithmes qui résolvent  $P$  dans un certain modèle de calcul est noté  $\mathcal{A}(P)$ .

**Exemple (Algorithmes *brute force*) :** Les algorithmes *brute force* ne font pas état de la structure sous-jacente du problème. Pour chaque entrée  $x \in X$ , un algorithme *brute force* parcourt toutes les sorties envisageables<sup>2</sup>  $y \in Y$  et teste si  $P(x) = y$

<sup>2</sup>. On précise "envisageable" malgré le côté *tout* tester, puisqu'on qualifie toujours de *brute force* un algorithme qui trierait un tableau de  $n$  éléments en essayant toutes les permutations de ces  $n$  éléments. Il pourrait *vraiment* tester tous les éléments de  $\mathbb{Z}^*$ , ce serait juste *infiniment* plus long.





## MESURES DU COÛT



### Définition 46 : Modèle de coût

Le modèle de coût spécifie la quantité de ressources que consomme une opération élémentaire d'un modèle de calcul sur une entrée donnée. Plusieurs modèles de coût différents sont possibles pour chaque modèle de calcul.

**Exemple (modèle à coût unitaire) :** chaque opération élémentaire coûte une unité, quelque soit l'entrée. On peut potentiellement spécifier que certaines opérations sont de coût nulle pour concentrer l'analyse sur des coûts spécifiques.

**Exemple :** Dans un modèle RAM, les calculs sont effectués par la lecture et l'écriture des registres selon certaines opérations. On peut spécifier le coût d'une opération dépendamment des entrées. Par exemple, si  $x, y \in \mathbb{N}$  sont non bornés, on peut poser que  $x + y$  a un coût proportionnel au  $\log_2$  de  $x$  et  $y$ .

**Exemple (modèle de coût de stockage) :** au lieu de considérer que la ressource est le temps, comme cela était implicitement supposé dans les deux exemples précédents, on peut considérer que la ressource est l'espace. Par exemple, l'écriture d'un espace mémoire *non encore utilisé* par un algorithme peut avoir un coût 1 et tout autre écriture d'espace mémoire *déjà utilisé* par l'algorithme est de coût 0. La *libération* d'un espace mémoire peut avoir un coût  $-1$ . On peut alors mesurer la "complexité spatiale" d'un algorithme.

Si le modèle de coût n'est pas spécifié et qu'on parle d'une ressource "temporelle" on suit par défaut un modèle à coût unitaire. Dans le cas où la ressource considérée dans le contexte est spatiale, on suit par défaut le modèle de coût de stockage décrit juste précédemment.

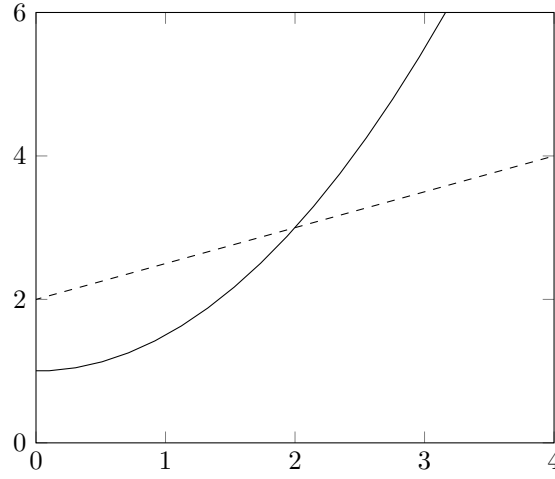
Toute la question est de comparer les différents algorithmes qui résolvent un même problème  $P : X \rightarrow Y$ , c'est-à-dire de comparer leurs coûts respectifs.

### Définition 47 : Fonction de coût

Soit  $P : X \rightarrow Y$ . La fonction de coût d'un algorithme  $A \in \mathcal{A}(P)$  est une fonction  $f_A : X \rightarrow \mathbb{R}_+$  qui à chaque entrée  $x \in X$  du problème donne la quantité de ressources utilisées par  $A$  pour calculer  $P(x)$ .

### 8.2.1 Notations de Landau

**Exemple :** Soient deux fonctions de coût  $f_A$  et  $f_B$  de  $\mathbb{N}$  dans  $\mathbb{R}_+$  définies par  $f_A(n) = 0.5n + 2$  et  $f_B(n) = 0.5n^2 + 1$ .

FIGURE 8.1 –  $f_A$  en ligne pointillée,  $f_B$  en ligne continue

Selon que  $n$  est inférieur ou supérieur à 3, on a soit  $f_A(n) > f_B(n)$ , soit  $f_A(n) < f_B(n)$ . On veut toutefois pouvoir comparer les algorithmes  $A$  et  $B$  sans ambiguïté. Par ailleurs, on veut aussi que l'analyse de  $A$  et  $B$  soit robuste, c'est-à-dire que de petites modifications de  $A$  et  $B$  n'invalident pas l'analyse de la fonction de coût de ceux-ci. Sinon, toutes les analyses d'algorithmes utilisant  $A$  et  $B$  sont aussi invalidés. Comme la modification légère d'un algorithme est monnaie courante en programmation, une absence de robustesse est inenvisageable.

Il faut définir avec plus de précision ce que signifie “modification légère”. Si on modifie un algorithme  $A$  en un nouvel algorithme  $A'$ , on veut que le coût de  $A$  par rapport à  $A'$  soit petit. Par “petit”, disons *borné*. C'est-à-dire que  $\frac{f_A}{f_{A'}}$  soit borné.

On en induit que étant donnés deux algorithmes  $A$  et  $B$ ,  $A$  est meilleur que  $B$  si  $\frac{f_B}{f_A}$  est non borné.

#### Définition 48 : Domination linéaire

Soit  $X$  un ensemble. On considère  $\preceq_X$  la relation de préordre<sup>a</sup> définie par :

$$\forall f, g \in \mathcal{F}(X, \mathbb{R}_+), f \preceq_X g \Leftrightarrow \exists c \in \mathbb{R}_+ \setminus \{0\} / f \leq cg$$

Si  $f$  et  $g$  sont deux fonctions telles que  $f \preceq_X g$ , on dit que  $f$  est *dominée linéairement* par  $g$ .

<sup>a</sup>. Réflexivité et transitivité, immédiates ici.

**Relations déduites :** On déduit du préordre de domination linéaire les relations suivantes, pour toutes fonctions  $f, g : X \rightarrow \mathbb{R}_+$

- $f \approx_X g \Leftrightarrow (f \preceq_X g) \wedge (g \preceq_X f)$
- $f \prec_X g \Leftrightarrow (f \preceq_X g) \wedge (f \not\approx g)$
- $f \succ_X g \Leftrightarrow g \prec_X f$
- $f \succeq_X g \Leftrightarrow g \preceq_X f$

**Définition 49 : Notation de Landau**

Soit  $X$  un ensemble, et  $f : X \rightarrow \mathbb{R}_+$  une fonction coût. On note :

$$O_X(f) = \{g : X \rightarrow \mathbb{R}_+ \mid g \preceq_X f\}$$

C'est la classe des fonctions de coût d'algorithmes "au moins aussi rapides" que celui associé à  $f$

**Notations déduites :**

- $\Theta_X(f) = \{g : X \rightarrow \mathbb{R}_+ \mid g \approx_X f\}$  est la classe d'équivalence de coût de  $f$
- $\Omega_X(f) = \{g : X \rightarrow \mathbb{R}_+ \mid g \succeq_X f\}$  est la classe des fonctions de coût d'algorithmes "plus lents" que celui associé à  $f$

**Remarque :** dans la pratique, une habitude des informaticiens est de ne pas préciser l'ensemble  $X$ , qui doit être déduit selon le contexte. Le plus souvent,  $X \subseteq \mathbb{N}$ .

**8.2.2 Analyse de cas pour la complexité**

On note par la suite, pour toute fonction  $f : X \rightarrow Y$  :

- pour  $S \subset X$ ,  $\vec{f}(S) = \{f(x) \mid x \in S\}$  est l'image de  $S$  par  $f$
- pour  $S \subset Y$ ,  $\overleftarrow{f}(S) = \{x \in X \mid f(x) \in S\}$  est l'ensemble des antécédents des éléments de  $S$  par  $f$

L'analyse de complexité d'un algorithme n'a pas pour but de déterminer entièrement sa fonction de coût. On s'intéresse à des cas particuliers d'entrées (pire cas ou meilleur cas) ou à des cas très généraux (cas moyen ou amorti).

**Définition 50 : Cas**

Soient  $X$ ,  $Y$  et  $Z$  trois ensembles. Soit  $P : X \rightarrow Y$  un problème.

On considère une fonction de groupement  $g : X \rightarrow Z$  qui à chaque entrée va associer une "mesure". Un cas  $s : \vec{g}(X) \rightarrow X$  selon ce groupement est une fonction qui à chaque mesure d'une entrée de  $g$  va renvoyer un élément de  $X$  de même mesure. C'est-à-dire que  $g \circ s = id_Z$ .

**Exemple pour comprendre :** Si  $X$  est l'ensemble des tableaux d'entiers, un groupement peut être défini par :

$$\begin{array}{ccc} g & : & \mathbb{Z}^* \rightarrow \mathbb{N} \\ x & \mapsto & |x| \end{array}$$

qui à chaque tableau associe sa longueur. Un cas est alors une fonction de  $\mathbb{N}$  dans  $\mathbb{Z}^*$  qui à chaque longueur  $n \in \mathbb{N}$  possible de tableau va attribuer un tableau de longueur  $n$  (parmi l'infinité des tableaux de taille  $n$  possibles) qui sera celui traité par l'algorithme dans ce cas.

L'utilisation de groupements permet d'indiquer n'importe quel moyen de mesure de l'entrée d'un problème.

**Remarque :** si le groupement choisi est à valeurs dans  $\mathbb{N}$  ou  $\mathbb{R}_+$  et représente la complexité spatiale de l'entrée (comme le nombre de bits nécessaire pour sa représentation), on peut parler de  $\vec{g}(X)$  comme de l'ensemble des *tailles des entrées* du problème. C'est généralement le cas<sup>3</sup>.

3. Sans jeu de mots aucun  $\wedge \wedge$

**Définition 51 : Pire cas**

Soit  $g : X \rightarrow Z$  un groupement. Soit  $s : \vec{g}(X) \rightarrow X$  un cas selon  $g$ . Soit  $f : X \rightarrow \mathbb{R}_+$  une fonction coût.

$s$  est un pire cas de  $f$  si pour tout  $z \in \vec{g}(X)$ ,  $(f \circ s)(z) = \sup \vec{f}(\overleftarrow{g}(\{z\}))$

Intuitivement, cela signifie que la fonction coût sur le pire cas est *maximale* pour toutes les “tailles” d’entrées (toutes les mesures d’entrées) possibles prises individuellement.

**Définition 52 : Meilleur cas**

Soit  $g : X \rightarrow Z$  un groupement. Soit  $s : \vec{g}(X) \rightarrow X$  un cas selon  $g$ . Soit  $f : X \rightarrow \mathbb{R}_+$  une fonction coût.

$s$  est un meilleur cas de  $f$  si pour tout  $z \in \vec{g}(X)$ ,  $(f \circ s)(z) = \inf \vec{f}(\overleftarrow{g}(\{z\}))$

Intuitivement, cela signifie que la fonction coût sur le pire cas est *minimale* pour toutes les “tailles” d’entrées (toutes les mesures d’entrées) possibles prises individuellement.

**Remarque :** Comme la borne inférieure et la borne supérieure de la fonction coût ne sont pas nécessairement atteintes, le pire et le meilleur cas n’existent pas toujours.

**Définition 53 : Analyse de complexité du pire cas**

L’analyse de complexité du pire cas d’une fonction de coût  $f : X \rightarrow \mathbb{R}_+$  selon un groupement  $g : X \rightarrow Z$  est le processus de recherche de la fonction de  $Z$  dans  $\mathbb{R}_+$  :

$$z \mapsto \sup \vec{f}(\overleftarrow{g}(\{z\}))$$

ou d’un ensemble  $O_X$  qui la contienne.

**Définition 54 : Analyse de complexité du meilleur cas**

L’analyse de complexité du pire cas d’une fonction de coût  $f : X \rightarrow \mathbb{R}_+$  selon un groupement  $g : X \rightarrow Z$  est le processus de recherche de la fonction de  $Z$  dans  $\mathbb{R}_+$  :

$$z \mapsto \inf \vec{f}(\overleftarrow{g}(\{z\}))$$

ou d’un ensemble  $\Omega_X$  qui la contienne.

On remarque que l’interprétation des analyses du pire ou un meilleur cas dépend du groupement choisi.

**Exemple :** Dans le cas de l’analyse de complexité d’un algorithme de tri de tableaux, on aurait pu choisir un groupement qui à chaque tableau associe la somme de ses éléments. La fonction de coût dans le pire cas doit être maximale pour toutes les valeurs possibles de sommes, c’est-à-dire que pour chaque somme  $s \in \mathbb{N}$ , il faut trouver un tableau qui maximise le coût de tri. Comme on peut choisir un tableau de taille arbitrairement grande, pour tout  $z$ ,  $\sup \vec{f}(\overleftarrow{g}(\{z\})) = \infty$ . De même, pour le meilleur cas, on a un tableau d’une unique valeur qu’on fait varier, d’où pour tout  $z$ ,  $\inf \vec{f}(\overleftarrow{g}(\{z\})) \approx 0^4$ .

4. Le symbole *approx* au lieu de  $=$  est dû aux quelques opérations du modèle de calcul qui peuvent être nécessaires pour donner le résultat effectif. En vérité, on a presque toujours  $\inf \vec{f}(\overleftarrow{g}(\{z\})) > 0$

Cela signifie que l'opération consistant à trier un tableau ne dépend pas de la somme du tableau. On arrive à des conclusions différentes selon le groupement choisi.

# STRUCTURES DE DONNÉES

On a défini à la section 6.1 les types de données et à la section 6.12 les enregistrements. Ce chapitre, intitulé *Structures de données*, semble d'un premier abord répétitif vis-à-vis de ces deux sections. Pourtant, les structures du langage C et les structures de données de manière générale doivent être distinguées puisqu'il s'agit en fait d'un même mot pour désigner deux choses tout à fait différentes.



## RETOUR SUR LES TYPES ABSTRAITS



Le type abstrait de données est un objet mathématique, qui décrit un ensemble abstrait et les opérations abstraites que l'on peut effectuer sur les éléments de cet ensemble. Il s'agit d'une structure algébrique pure. Par exemple, le corps  $\mathbb{R}$  muni de l'addition et de la multiplication peut être assimilé à un type abstrait de données. Les anneaux  $\frac{\mathbb{Z}}{2^N\mathbb{Z}}$  munis de l'addition et de la multiplication présents nativement en langage C sont aussi des types abstraits de données. Les enregistrements munis des opérations sur chaque champ sont aussi des types abstraits de données.

L'idée fondamentale est de ne pas avoir à se soucier de la représentation lors de la programmation. Il suffit de connaître le type. Ainsi, il n'est pas nécessaire de connaître la représentation des nombres flottants (signe, mantisse et exposant) pour les manipuler. Il n'y a pas besoin de connaître l'*offset* de chaque champ d'une structure en mémoire pour manipuler les instances d'enregistrement qu'elle implante.

On peut considérer deux démarches possibles pour la construction de types abstraits :

- *ascendante* : on se donne une représentation concrète du type de données en terme d'objets du langage et des routines de manipulation correspondant aux opérations du type. Si on ne manipule plus la représentation que par ces routines, on manipule en fait le type abstrait.
- *descendante* : on se donne une spécification d'un type abstrait et on conçoit l'algorithme à ce niveau. On donne une représentation concrète des types et opérations pour obtenir un programme exécutable.

**Remarque :** on peut considérer plusieurs couches d'abstraction entre la représentation binaire et le plus haut niveau d'abstraction. Le type le plus abstrait a comme représentation "concrète" d'autres types abstraits eux-mêmes représentés par des types abstraits, etc..., jusqu'à des types abstraits *élémentaires* qui ont comme représentation concrète un mot binaire.

### 9.1.1 Spécification d'un type abstrait sur un exemple

Les types abstraits servent à décrire des objets bien plus complexes que des réels ou des booléens. Il faut pouvoir définir avec précision un type, qui ne dépende pas de l'implantation dans un langage spécifique.

Il faut pouvoir décrire :

- les types prédéfinis utilisés par le type défini
- la signature des opérateurs sur le type, qui décrit aussi leur syntaxe
- la sémantique des opérateurs sur le type

On appelle signature d'un type l'ensemble des signatures de chacune de ses opérations (au sens donné à la **Définition 32**).

On rappelle qu'il est possible de spécifier avec précision la syntaxe d'une opération sur un type grâce à la signature de l'opération. Ainsi, un type abstrait *Tableau* pourrait avoir la signature suivante :

*Tableau*<*Element*> utilise *Entier* et *Element*

- *creer*(*Entier*) → *Tableau*
- *copier*(*Tableau*, *Entier*) → *Tableau*
- *lire*(*Tableau*, *Entier*) → *Element*
- *ecrire*(&*Tableau*, *Entier*, *Element*)
- *taille*(*Tableau*) → *Entier*

Les chevrons indiquent que *Element* est un type abstrait quelconque dont *Tableau* dépend. Ainsi, *Tableau*<*Flottant*> est un tableau de **float** (*Element* ≡ *Flottant*)

#### Point vocabulaire

On appelle :

- *accesseur* un opérateur qui prend en paramètre un élément d'un type défini et renvoi un élément d'un type prédéfini. C'est le cas par exemple de *lire* ci-dessus
- *constructeur* un opérateur qui prend ne prend en paramètre des éléments de types prédéfinis ou de type défini et renvoi un élément de type défini. C'est le cas par exemple de *creer* et de *copier* ci-dessus.
- *mutateur* un opérateur qui prend en paramètre une référence<sup>a</sup> vers un élément d'un type défini et renvoi un élément d'un type défini. C'est le cas par exemple de *ecrire* ci-dessus

Dans la théorie, on peut assimiler un mutateur à une fonction équivalente renvoyant l'objet modifié. Cela permet de raisonner sur les propriétés de l'objet résultantes de la mutation, ce qui est particulièrement utile pour l'écriture des équations de spécification. Si le mutateur devait renvoyer une valeur *v*, on peut considérer que la fonction équivalente renvoie un couple (*v*, *objet modifié*).

<sup>a</sup>. On rappelle que les références en C sont construites et manipulées grâce aux opérateurs unaires \* et &.

On observe que les noms des signatures ne sont pas suffisants pour décrire la sémantique du type. En effet, ces noms sont arbitraires. La signature ci-dessus est équivalente à la suivante : *T*<*E*> utilise *I* et *E*

- $c(I) \rightarrow T$
- $cc(T, I) \rightarrow T$
- $l(T, I) \rightarrow E$
- $e(T, I, E) \rightarrow T$
- $t(T) \rightarrow I$

En on ne comprend rien, car aucune sémantique n'est définie par les signatures.

Il faut pouvoir déterminer le comportement voulu d'une opération sans l'implanter dans un langage. En effet, il faut laisser toute sa liberté au programmeur dans la manière d'implanter le type (pour des raisons d'optimisation par exemple) tout en contraignant son comportement.

Une solution mathématique à ce problème est de décrire par des axiomes le comportement de chaque opération. Ces axiomes doivent être *cohérents* et *complets*.

Par *cohérence*, on entend qu'il doit être impossible par une séquence d'opérations d'arriver à des comportements contradictoires comme  $acces(t, 0) = e_1 \wedge acces(t, 0) = e_2$  avec  $e_1 \neq e_2$ , c'est-à-dire qu'on ne doit pas pouvoir *prouver* deux états différents de la structure de données par un même algorithme.

Par *complétude*, on entend qu'il faut que les axiomes permettent de déterminer avec certitude le comportement de tous les accesseurs du type sur leur domaine de définition. En effet, c'est le comportement observable du type qui correspond à sa sémantique. La restriction au domaine de définition est importante, puisqu'il est par exemple impossible de lire un élément dans un tableau hors de celui-ci.

On appelle un type abstrait décrit par de tels axiomes algébriques un *type abstrait algébrique*.

Dans l'exemple ci-dessus, on peut poser les axiomes suivants, qui sont vrais pour tous  $t : \text{Tableau}$ ,  $i, j : \text{Entier}$  et  $e : \text{Element}$  :

- $taille(creer(i)) = i$
- $taille(ecrire(t, i, e)) = taille(t)$  (conservation de la taille par modification)
- $taille(copier(t)) = taille(t)$  (conservation de la taille par copie)
- $0 \leq i < taille(t) \Rightarrow lire(copier(t), i) = lire(t, i)$  (conservation des valeurs par copie)
- $0 \leq i < taille(t) \Rightarrow ecrire(lire(t, i), i, e) = e$  (conservation après modification)
- $0 \leq i, j < taille(t) \wedge i \neq j \Rightarrow lire(ecrire(t, j, e), i) = lire(t, i)$  (modification locale)

Cela *semble* suffisant pour décrire complètement le type. Mais si on utilise *creer*, on ne peut pas déterminer les valeurs de sortie de l'observateur *lire* puisque aucune valeur n'a été écrite. On pourrait supposer l'existence d'un neutre  $0_E$  de type *Element* et ajouter l'axiome  $0 \leq i < j \Rightarrow lire(creer(j), i) = 0_E$ .

Pour éviter cette supposition sur *Element*, on peut ajouter un prédicat supplémentaire de signature  $init(\text{Tableau}, \text{Entier}) \rightarrow \text{Booléen}$  qui renvoie *Vrai*. On ajoute alors une contrainte sur l'existence de  $lire(t, i)$  : il faut non seulement que  $0 \leq i < taille(t)$  mais aussi  $init(t, i) = \text{Vrai}$ . On ajoute alors les trois axiomes :

- $0 \leq j < i \Rightarrow init(creer(i), j) = \text{Faux}$
- $0 \leq i < taille(t) \Rightarrow init(ecrire(t, i, e), i) = \text{Vrai}$
- $0 \leq i < taille(t) \wedge i \neq j \Rightarrow init(ecrire(t, i, e), j) = init(t, j)$
- $0 \leq i < taille(t) \Rightarrow init(copier(t), i) = init(t, i)$

On a alors complètement défini le type abstrait algébrique *Tableau*, contenant des éléments de type *Element*. On observe qu'aucune hypothèse n'a été conduite sur le type *Element*, et que celui-ci peut donc être quelconque.



### 9.1.2 Structure de données

C'est bien beau de décrire tous les types abstraits algébriquement, cela permet d'éviter tout malentendu et toute erreur d'interprétation. On peut utiliser ces axiomes pour prouver formellement les algorithmes que l'on construit (voir l' **Exercice 64** pour un exemple "simple"). D'un point de vue théorique c'est super et pour certaines applications critiques qui mettent en jeu des vies humaines, cela se révèle même assez nécessaire.

Mais dans la pratique la plus commune, on voudrait bien pouvoir lire facilement la description d'un type sans avoir à décortiquer des axiomes. Dans la suite, on ne décrira donc pas la spécification formelle de chaque type abstrait<sup>1</sup>.

**Compromis clarté/rigueur :** Dans la suite, on se contentera de décrire les signatures des opérations sur les types suivis d'une description et d'explications *informelles* qui se fonderont sur la formalisation mathématique des types. Si une opération nécessite une justification/explication/démonstration de son comportement, on la donnera le plus souvent possible.

**À propos des pseudo-codes :** On utilise la signature dans les algorithmes en pseudo-code pour garantir la non-interprétation de ceux-ci. Si possible, on utilise directement la formalisation mathématique équivalente pour être le plus précis possible.

C'est dans l'implantation des corps de fonctions d'un type que l'on va trouver tous les détails de manipulation des représentations et des types d'abstraction inférieurs. C'est cette implantation qu'on appelle *structure de données*. On décrira plusieurs possibilités d'implantation des types abstraits de données, c'est-à-dire plusieurs *structures de données* associées au type abstrait.

#### Définition 55 : Structure de données

Une structure de données est l'implantation concrète explicite d'un type abstrait de données. Elle est donc dépendante du langage, du système d'exploitation et de l'architecture du/des processeur(s).

Une structure de données est alors formée des données spécifiques à l'objet décrit et des routines de manipulation de cet objet, qui implantent les signatures des opérateurs. L'enregistrement détermine l'ensemble sur lequel travaillent les opérateurs et il est standard d'utiliser ce type abstrait pour représenter l'ensemble des objets décrits.

**Remarque :** dans le langage courant, on prend parfois un raccourci en parlant de structure de données en lieu et place de type abstrait de données. Certains précisent structure de données *abstraite* pour conserver la rigueur. Une structure est tout de même plus à rapprocher de l'implantation d'un enregistrement que comme équivalence d'un type.

En langage C, on pourra donc décrire le stockage des données par un enregistrement `struct Objet {...};` et préfixer chaque fonction de manipulation par `objet_` pour faciliter la lecture et la programmation. Ainsi, on peut proposer l'entête pour l'implantation du type abstrait *Tableau* dans un module spécialisé<sup>2</sup> :

```
1  #ifndef TABLEAU_H_INCLUDED
2  #define TABLEAU_H_INCLUDED
3
```

1. On laisse la lecture du livre de Froidevaux [9] en complément.

2. Il s'agit évidemment d'un exemple d'illustration seulement. Planter un module "tableau" n'a aucun sens en C, à part à écrire trop de code pour rien.

```

4  #include "element.h"
5
6  typedef struct Tableau {
7      Element* array;
8      unsigned int length;
9  } Tableau;
10
11  Tableau tableau_creer(unsigned int length);
12  Tableau tableau_copier(Tableau t);
13  Element tableau_lire(Tableau t, unsigned int i);
14  void tableau_ecrire(Tableau t, unsigned int i, Element e);
15  unsigned int tableau_taille(Tableau t);
16
17  #endif

```

L'interface du module respecte la signature du type abstrait. On observe qu'il faudrait effectuer autant d'implantations que de types *Element* différents utilisés (par exemple *Tableau<Entier>*  $\neq$  *Tableau<Flottant>*). C'est une limitation du langage C, dont ne souffre pas le C++ par exemple, qui peut automatiser l'implantation de dépendances à d'autres types.

### 9.1.3 Exercices

#### Exercice 59 (Tableaux dynamiques) [20].

- Implanter en C la structure de donnée *Tableau* dont la spécification a été donnée dans le cours. *Note : on pourra poser  $Element \equiv int$  pour simplifier. Dans le cadre de la gestion de la mémoire en C, une procédure de libération de la mémoire du tableau est bienvenue.*
- Étendre la spécification de la structure *Tableau* à une procédure *redimensionner*(&*Tableau* : *t*, *Entier* : *n*) qui ajoute ou supprime des cases de *t* de sorte à ce qu'il ne fasse plus que *n* cases après la mutation. Implanter cette opération selon la spécification écrite.

#### Exercice 60 (Entiers relatifs et rationnels) [35].

1. En représentant les nombres de  $\mathbb{Z}$  sur un nombre d'octets déterminé dynamiquement selon la taille des éléments, implanter une structure de données *Integer* de l'anneau  $\mathbb{Z}$  muni de l'addition et de la multiplication. Les choix de *design* et d'implantation sont laissés à la liberté du lecteur.
2. Implanter le type abstrait du corps  $\mathbb{Q}$  muni de l'addition et de la multiplication en représentant chaque élément comme un couple  $(a, b) \in \mathbb{Z} \times \mathbb{N}$



Une liste est un type qui permet le stockage linéaire de données sous forme d'une suite finie  $L = [e_1, \dots, e_n]$ . La liste est dite *vide* si  $n = 0$ . Chacune de ces données est repérée par son indice  $i \in \llbracket 1; n \rrbracket$  dans la suite  $L$ . Les éléments de la suite sont soit accessibles directement par leur indice, soit indirectement, quand la liste est représentée par une suite d'objets *chaînés*.

On considère alors :

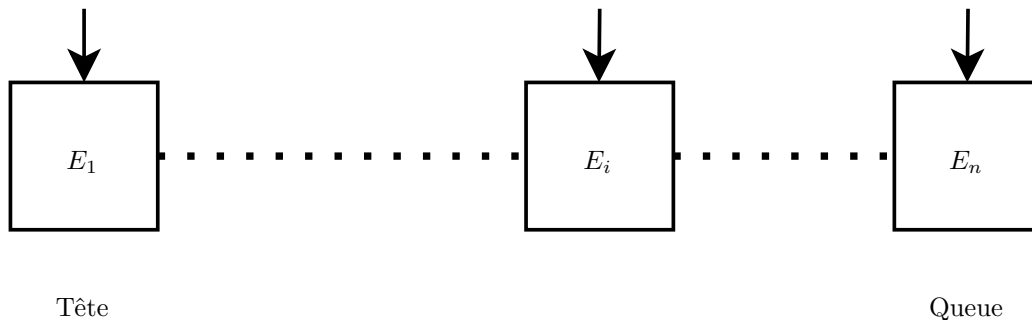
- la *place* d'un élément, qui représente son indice et permet alors la manipulation de cet indice (par exemple pour calculer le successeur dans la liste).
- une fonction *contenu* qui pour une place donnée renvoie l'élément  $e$  associé

Les données elles-mêmes ne sont pas *nécessairement* ordonnés. Par contre, les places assignées à chaque donnée le sont selon leur rang dans la liste.

On appelle :

- *nœud* de la liste la donnée d'une place et d'un élément contenu à cette place. Un enregistrement de nœud dans l'implantation peut contenir d'autres valeurs, comme la place du nœud suivant ou précédent.
- *tête* de la liste le premier nœud, contenant  $e_1$
- *queue* de la liste le dernier nœud, contenant  $e_n$

**Remarque :** la distinction entre “indice” et “place” est importante pour s'abstraire de l'implantation. La place peut être représentée par une adresse mémoire distincte de l'indice par exemple. Concrètement, on peut envisager de manière assez générale la place comme l'indication d'emplacement mémoire de la donnée représentant l'élément dans la liste.



La différence fondamentale entre les places et les éléments contenus à ces places tient à l'absence de contraintes sur les valeurs des éléments. Ainsi, il est possible d'avoir deux éléments de valeur identique à deux places différentes, alors que toutes les places sont différentes.

Une signature minimale d'une liste peut alors être :

*Liste* <Élément> utilise *Booléen*, *Élément* et *Place*

- *liste\_vide()* → *Liste* renvoie une liste vide, sans éléments
- *premier(Liste)* → *Place* renvoie la place du premier élément, indéfini si la liste est vide
- *est\_dernier(Liste, Place)* → *Booléen* Teste si la place est la dernière de la liste.

- $\text{successeur}(\text{Liste}, p : \text{Place}) \rightarrow \text{Place}$  calcule la place suivant une place, indéfini si  $p$  est la dernière place de la liste
- $\text{contenu}(\text{Liste}, \text{Place}) \rightarrow \text{Élément}$  renvoie l'élément contenu à une certaine place
- $\text{est\_vide}(\text{Liste}) \rightarrow \text{Booléen}$  renvoie *Vrai* si la liste est vide, *Faux* sinon
- $\text{supprimer\_en\_tete}(\text{mut Liste}) \rightarrow \text{Élément}$  supprime le premier élément de la liste et renvoie sa valeur
- $\text{insérer\_en\_tete}(\text{mut Liste}, \text{Élément})$  insère un élément comme premier élément de la liste

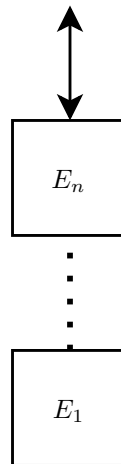
On peut étendre cette signature en ajoutant d'autres opérations. Plus on ajoute d'opérations, plus la structure de donnée implantée peut devenir complexe pour rester efficace. On peut ainsi envisager d'ajouter des éléments en queue de la liste ou même d'en insérer/supprimer au milieu.

Ajouter des éléments en queue de liste induit soit de devoir parcourir toute la liste pour arriver à la fin, soit d'ajouter un accesseur qui renvoie la dernière place de la liste. Pour calculer cet accesseur en  $\mathcal{O}(1)$ , il faut ajouter un pointeur vers la queue, ce qui constitue une complexification.

### 9.2.1 Piles

Une pile est un cas particulier plus simple de la liste qui explicitement n'autorise la suppression et l'ajout d'éléments que d'un seul côté d'une liste linéaire. Les piles ont un intérêt algorithmique pour la manipulation de structures plus complexes, par exemple pour le parcours en profondeur de graphes.

Pour visualiser, on peut faire assez correctement l'analogie avec une pile d'assiettes : on ne peut prendre que l'assiette du sommet de la pile, et une assiette ajoutée ou enlevée de la pile le sera toujours à son sommet :



En anglais, cette structure est aussi appelée *LIFO* (*Last In First Out*), ce qui résume parfaitement son fonctionnement.

La signature d'une pile est la suivante :  $\text{Pile} \langle \text{Élément} \rangle$  utilise *Booléen*, *Élément*

- $\text{pile\_vide}() \rightarrow \text{Pile}$  renvoie une pile vide, sans éléments
- $\text{sommet}(\text{Pile}) \rightarrow \text{Élément}$  renvoie l'élément au sommet de la pile qui doit être non vide
- $\text{empiler}(\text{Pile}, \text{Élément}) \rightarrow \text{Pile}$  insère un élément au sommet de la pile
- $\text{depiler}(\text{Pile}) \rightarrow \text{Pile}$  supprime l'élément au sommet de la pile, qui doit être non vide
- $\text{est\_vide}(\text{Pile}) \rightarrow \text{Booléen}$  Teste si la pile est vide

L'avantage de considérer un type abstrait *Pile* distinct du type *Liste* n'est pas évident au premier abord. L'avantage est pourtant dans la grande simplicité de la pile qui laisse à l'implantation la possibilité

d'être grandement optimisé. En effet, tout comme l'ajout d'opérations amène à la complexification de l'implantation, avoir peu d'opérations est plus à même de garantir une implantation plus simple et plus efficace.

On peut par exemple envisager l'implantation suivante d'une Pile :

```

1  #include <stdlib.h> // malloc et free
2  #include <stdbool.h> // type bool
3
4  #include "element.h" // type E
5
6  typedef struct Stack {
7      E* elts;
8      unsigned int sp; // stack pointer -> place du sommet
9      unsigned int capacity;
10 } Stack;
11
12 Stack stack_empty() {
13     Stack s;
14     s.elts = (E*)malloc(10*sizeof(E));
15     s.capacity = 10;
16     s.sp = 0;
17     return s;
18 }
19
20 E stack_top(Stack s);
21 inline E stack_top(Stack s) {
22     return s.elts[s.sp - 1];
23 }
24
25 Stack stack_push(Stack s, E e) {
26     s.elts[s.sp++] = e;
27     if (s.sp == s.capacity) { // dépasse les capacités du tableau => doubler sa taille
28         unsigned int new_capacity = s.capacity * 2; // * 2
29         E* new = (E*)malloc(new_capacity*sizeof(E));
30         for (unsigned int i = 0; i < s.capacity; i++) {
31             new[i] = s.elts[i];
32         }
33         s.capacity = new_capacity;
34         free(s.elts);
35         s.elts = new;
36     }
37     return s;
38 }
39
40 Stack stack_pop(Stack s) {
41     s.sp--;
42     return s;
43 }
44
45 bool stack_is_empty(Stack s);
46 inline bool stack_is_empty(Stack s) {
47     return s.sp == 0;
48 }
49
50 Stack stack_destroy(Stack s) {

```

```

51 |   free(s.elts);
52 | }

```

On peut encore simplifier l'implantation de *push* si on connaît d'avance la taille maximal de la pile. Cela évite d'avoir à effectuer des copies de tableau puisqu'on peut immédiatement allouer un tableau de la taille nécessaire pour stocker les éléments.

### 9.2.2 Files

Une pile est un type plus simple qui n'accorde l'ajout d'éléments qu'à un bout de la liste et la suppression d'éléments qu'à l'autre bout de la liste. Les piles ont un intérêt algorithmique pour la manipulation de structures plus complexes, par exemple pour le parcours en largeur de graphes.

Pour visualiser, on peut faire assez correctement l'analogie avec des perles enfilées sur un fil. À ce fil on n'ajoute des perles que d'un côté, et on n'en enlève que de l'autre.



En anglais, cette structure est aussi appelée *FIFO* (*First In First Out*), ce qui résume parfaitement son fonctionnement.

La signature d'une file est la suivante :

*File* <Élément> utilise *Booléen*, *Élément*

- *file\_vide()* → *File* renvoie une file vide, sans éléments
- *premier(File)* → *Élément* renvoie le premier élément de la file (celui le plus “à gauche” sur le schéma)
- *dernier(File)* → *Élément* renvoie le dernier élément de la file (celui le plus “à droite” sur le schéma)
- *enfiler(File, Élément)* → *File* insère un élément *en tête* de la file
- *defiler(File)* → *File* supprime l'élément *en queue* de la file, qui doit être non vide
- *est\_vide(File)* → *Booléen* Teste si la file est vide

De même que pour les piles, la signature simplifiée des files par rapport aux listes laisse une plus grande liberté d'implantation et donc d'optimisation.

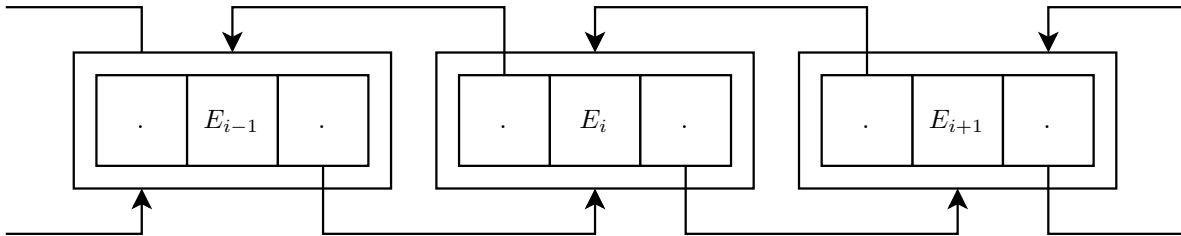
On ne décrira pas une implantation par un tableau des files, qui est laissée en exercice.

### 9.2.3 Implantation par chaînage

Le problème de l'implantation des listes par un tableau est le suivant : dans le cas où le tableau devient trop petit, il n'est pas possible de *l'agrandir*. En effet, les cases mémoires adjacentes peuvent très bien être utilisées par d'autres programmes ou d'autres parties de notre programme. Il faut donc créer un nouveau tableau plus grand et copier les éléments du premier dans le second. Cette opération est très coûteuse en temps ( $\Theta(n)$ ). On observe toutefois qu'en choisissant un bon facteur d'agrandissement, cette opération de copie sera appelée assez rarement et la complexité temporelle d'un ajout en tête ou en queue de liste reste constante ( $1 - O(\frac{1}{n})$ ) (la preuve est laissée en exercice). Cela ne résout pas le

problème de l'insertion en milieu qui demande de copier à chaque insertion une partie du tableau pour le décaler.

On s'intéresse ici à une autre manière de procéder qui garantit l'ajout en tête et en queue de liste en  $O(1)$  dans le pire des cas. L'idée fondamentale est de dissocier les emplacements mémoire des noeuds de la liste. Chacun sera stocké indépendamment des autres, de manière non contiguë, grâce à une allocation sur le tas<sup>3</sup>. Pour conserver les informations de tous les noeuds de la liste, chaque noeud stocke la place du noeud suivant et du noeud précédent :



En partant de la tête et en sachant calculer la place suivante, on peut donc parcourir toute la liste. De même, en sachant calculer la place précédente, on peut partir de la queue et parcourir toute la liste. En utilisant simplement une adresse mémoire pour implanter une place, on observe qu'aucun calcul n'est à faire. Les noeuds stockent simplement l'adresse des noeuds suivant et précédent.

L'enregistrement d'un noeud en C est alors :

```
typedef struct ListNode ListNode;
typedef ListNode* Place;
struct ListNode {
    Element e;
    Place next;
    Place previous;
};
```

La dernière question reste dans l'enregistrement de la première place et de la dernière, pour manipuler la liste en tête et en queue. Il faut aussi choisir une convention qui ne soit pas sans queue ni tête pour les listes vides, qui n'ont ni tête ni queue. Des problèmes techniques peuvent aussi apparaître lorsqu'on a qu'un unique élément, puisque la tête et la queue sont à la même place. Libérer deux fois la même mémoire aboutit à une *segmentation fault*, ce qui n'est pas agréable.

Le choix le plus classique est d'utiliser un enregistrement comme suit :

```
typedef struct List {
    unsigned int length;
    Place head;
    Place tail;
} List;
```

La valeur `NULL` pour les places indique alors l'absence d'éléments. On peut traiter tous ces cas particuliers engendrés dans fonctions de manipulation par des conditions. Le risque d'erreurs est grand et les conditions ralentissent le code.<sup>4</sup>

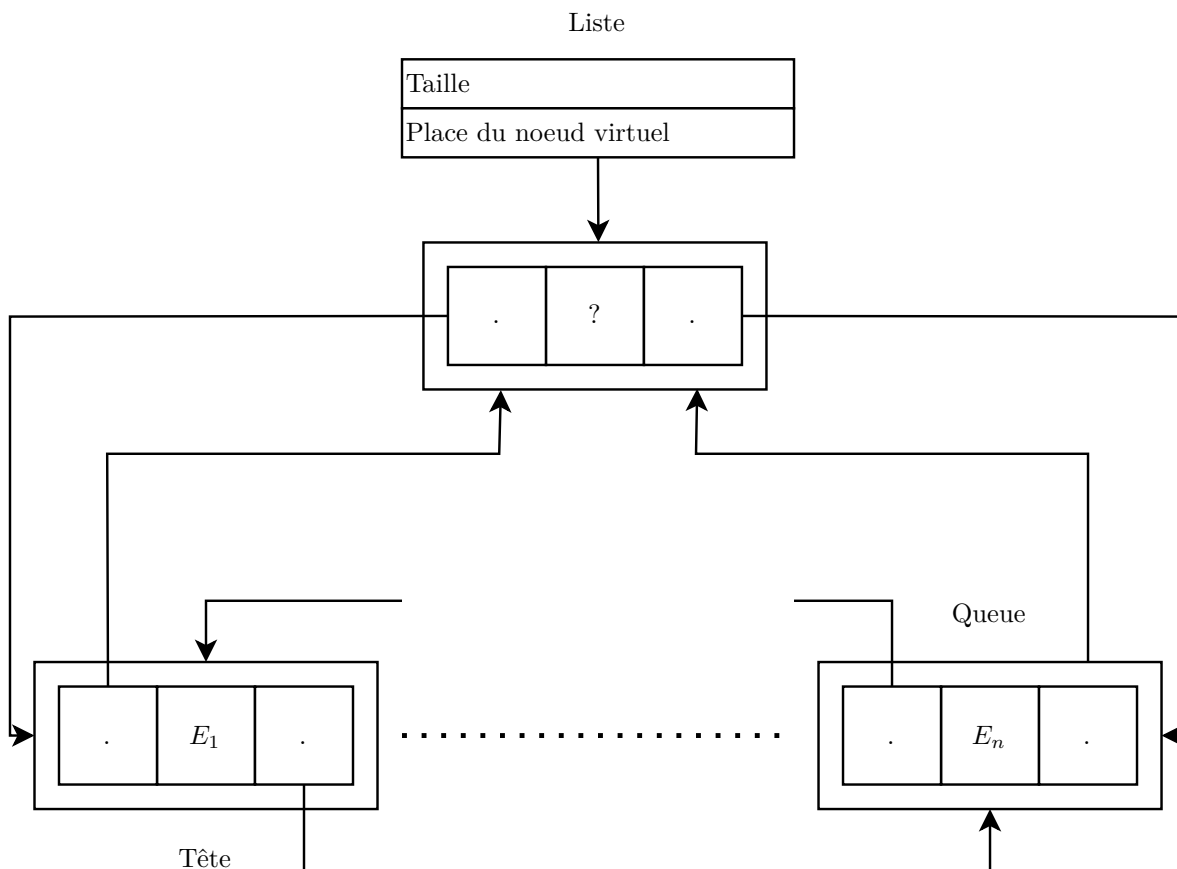
3. par malloc donc

4. Ça reste un bon exercice pratique sur les pointeurs

On présente ici une astuce d'implantation qui permet d'éviter de tels soucis, avec la structure de liste suivante :

```
typedef struct List {
    unsigned int length;
    Place virtual;
} List;
```

On ajoute un noeud virtuel, qui n'appartient pas à la liste au sens abstrait. Il sert uniquement à pointer vers les places de la tête et de la queue. La différence fondamentale est que la liste n'est jamais matériellement vide. Par ailleurs, quand on a un unique élément dans la liste, celui-ci ne pointe pas vers lui-même mais vers le noeud virtuel, ce qui évite des erreurs lors de la suppression de l'élément.



Il n'y a pratiquement plus aucun test à effectuer pour vérifier si les pointeurs de tête et queue sont `NULL` ou non, ce qui accélère les fonctions de manipulation.

Ainsi, l'ajout en queue de liste devient :

```
List list_push_on_tail(List l, Element e) {
    ListNode *new = malloc(sizeof(ListNode));
    new->next = l->virtual;
    l->virtual->prev->next = new;
    new->prev = l->virtual->prev;
```



```

l->virtual->prev = new;
new->e = e;
l->length++;
return l;
}

```

## 9.2.4 Exercices

**Exercice 61 (Listes virtuelles) [20].** Implanter les routines de la signature d’une liste en utilisant une liste chaînée à nœud virtuel. Ajouter une routine `void destroy(List l);` qui libère la mémoire de tous les nœuds de la liste<sup>5</sup>.

*Remarque :* on pourra considérer que les éléments sont des entiers, comme pour l’**Exercice 48**, pour pouvoir avoir un code fonctionnel et directement testable.

**Exercice 62 (Listes par tableaux) [23].** Implanter une liste avec un tableau, qui permette l’accès, l’insertion et la suppression en tête et queue. On veut que le coût dans le pire cas de l’accès et de la suppression soit fonction de  $O(1)$  et que le coût dans le pire cas des  $n$  premières insertions soit fonction de  $O(n)$ <sup>6</sup>. Quelle est le coût temporel des opérations de lecture, d’insertion et de suppression d’éléments à une place quelconque de la liste ? Comparer à l’implantation par chaînage.

**Exercice 63 (Axiomes algébriques) [25].**

Définir les axiomes algébriques qui régissent le comportement :

- d’une pile
- d’une file

*Indication :* on pourra remarquer que toute pile ou file peut être écrite sous forme canonique par ajouts d’éléments. Les opérations permettant cette écriture sont dites canoniques. Il est alors nécessaire et suffisant de décrire le comportement des opérations non canoniques selon les opérations canoniques ainsi que le comportement des opérations canoniques entre elles.

**Exercice 64 (Piles et files) [29].**

1. Construire :
  - une file en utilisant deux piles
  - une pile en utilisant deux files
2. Montrer en utilisant les axiomes de l’**Exercice 63** que les constructions sont correctes
3. Quelle est la complexité temporelle amortie des opérations de manipulation ?

**Exercice 65 (Optimizing greater than abstraction) [29].** Implanter une structure de liste *doublement* chaînée dont les noeuds ne possèdent qu’un seul pointeur qui contient l’information à la fois du pointeur avant et du pointeur arrière. Écrire alors une procédure de renversement qui inverse l’ordre de la liste avec un coût fonction de  $O(1)$

*Indication :* utiliser les propriétés du groupe commutatif  $(\mathcal{B}^N, \oplus)$  et analyser le titre de l’exercice.

**Exercice 66 (Fusion triée de listes) [17].** Écrire une fonction  $\text{fusion}(\text{Liste}, \text{Liste}) \rightarrow \text{Liste}$  qui renvoie la fusion de deux listes d’entiers triés. La liste résultante devra également être triée et sans doublons. Si les deux listes possèdent  $n$  et  $m$  éléments, le coût dans le pire cas doit être fonction de  $O(n + m)$ .

5. Cette routine est directement induite par l’implantation.

6. On dira que le coût *amorti* d’une seule insertion est fonction de  $O(1)$



## ARBRES BINAIRES

### 9.3.1 Motivation

Le problème des structures comme les tableaux ou les listes tient à leur linéarité. Si on veut manipuler certaines données spécifiques, on doit le plus généralement parcourir l'entièreté de toutes les données de la structure.

L'idée des *arbres* est de venir segmenter itérativement ces données, de la même façon qu'on trie et qu'on catalogue des données au format papier dans des archives par exemple. On commence par séparer l'ensemble des données en plusieurs segments (souvent étiquetés mais pas toujours). Chaque segment peut lui-même être à nouveau segmenté, etc. . . jusqu'à obtenir des segments atomiques<sup>7</sup>.

Les arbres constituent donc des structures *non linéaires*, c'est-à-dire que les éléments ne sont pas stockés les uns à la suite des autres séquentiellement. Il s'agit en fait d'un cas particulier des graphes<sup>8</sup>, comme cela sera vu plus tard.

Basiquement un arbre *réel* peut être vu comme un racine à laquelle sont accrochées des branches. À chaque branche on trouve des noeuds (segments étiquetés) qui donnent de nouvelles branches. Au bout de chaque branche il y a ou non une feuille (segment atomique/unitaire).

L'avantage d'une telle structure peut être intuité comme suit : en partant de la racine et avançant de branche en branche vers une feuille, on atteint la feuille sans avoir parcouru toutes les branches ni toutes les feuilles de l'arbre. Dans une archive papier, cela revient à checker d'abord les casiers, puis lorsqu'on a trouvé le bon casier, on cherche à l'intérieur le bon dossier, et dans ce dossier, le bon fichier. On a ainsi évité de chercher dans tous les casiers et dans tous les dossiers.

L'idée est de construire formellement une structure équivalente avec cette propriété extraordinaire qui permet de limiter grandement la complexité des opérations d'accès aux données.

### 9.3.2 Arbres quelconques

On appelle *arité* d'un arbre le nombre maximum de branches que peut avoir un noeud dans l'arbre.

#### Noeud et racine

On observe que la racine d'un arbre est un noeud comme les autres. Il a simplement la propriété d'être le noeud premier, qui n'a pas de noeud *parent*. Si la racine a une branche vers un noeud, on peut à l'inverse considérer ce noeud comme la racine d'un sous-arbre.

Ainsi, un arbre  $T$  d'arité  $n$  est la donnée d'un noeud racine  $r$  et d'au plus  $n$  sous-arbres  $t_1, \dots, t_n$  chacun d'arité  $n$ . On peut avoir exactement  $n$  sous-arbres à chaque fois si on complète par une *absence d'arbre*, qu'on appelle l'*arbre vide*  $E$ .

---

7. Au sens d'insécables.

8. Ce qui ne signifie pas que ce soit plus facile. Comme c'est simple mais qu'il faut faire des trucs intéressants avec, alors des gens un peu cinglés décident de complexifier à mort histoire qu'on y comprenne plus rien. Simple plaisir d'informaticien.

## Feuilles

Une feuille peut être vue comme un noeud... qui n'a pas de branches. C'est-à-dire que tous les sous-arbres sont absents. On définit donc une feuille comme un noeud dont tous les sous-arbres sont  $E$ .

**Remarque :** avec cette définition, une feuille forme toujours un arbre.

### 9.3.3 Principe

On s'arrête ici aux arbres binaires<sup>9</sup>, c'est-à-dire d'arité 2.

**Remarque :** Comme pour les listes, on va différencier un noeud  $n$  de l'arbre de son contenu enregistré  $x_n$ . Il sera ainsi possible de considérer des arbres binaires dont chaque noeud stockera toutes les informations que l'on veut puisqu'on peut choisir un ensemble arbitraire pour les  $x_i$

**Exemple :** Ci-dessous, un arbre<sup>10</sup> binaire à 6 noeuds :  $0, \dots, 5$ . 0 est la racine. Les arbres vides  $E$  ne sont pas montrés pour alléger le schéma.

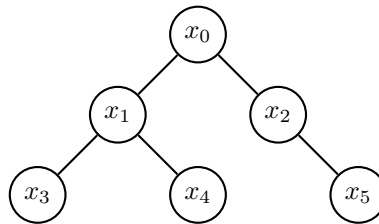


FIGURE 9.1 – Un arbre binaire

#### Détails :

- l'arbre de racine 3 a  $E$  comme sous-arbre à gauche comme à droite. On note cet arbre  $a_3 = (E, x_3, E)$
- *idem* pour l'arbre de racine 4, on a  $a_4 = (E, x_4, E)$
- *idem* pour l'arbre de racine 5, on a  $a_5 = (E, x_5, E)$
- l'arbre de racine 1 est donc  $a_1 = (a_3, x_1, a_4)$
- l'arbre de racine 2 est donc  $a_2 = (E, x_2, a_5)$
- la racine de l'arbre total est 0. L'arbre total est noté  $A = (a_1, x_0, a_2)$

On peut définir de la même façon n'importe quel arbre par induction<sup>11</sup> sur ses sous-arbres :

9. Y a déjà assez à faire comme ça...

10. Les informaticiens, ne sortant jamais de leur cave où ils restent le nez rivé sur leur écran, n'ont jamais vus ni vrai herbe, ni vrai arbre. Ça explique pourquoi leurs représentations d'arbres sont à l'envers...

11. Récurrence généralisée à n'importe quel type d'objet avec un ensemble de base défini par une règle  $\mathcal{B}$  et une induction  $\mathcal{I}$  sur l'ensemble

**Définition 56 : Arbres binaires**

On appelle  $E$  l'arbre vide.

On note  $A_X$  l'ensemble des arbres binaires définis sur un ensemble  $X$ .  $A_X$  est défini par induction par les règles :

$$\begin{cases} \mathcal{B} & : E \in A_X \\ \mathcal{I} & : \forall x \in X, \forall g, d \in A_X, (g, x, d) \in A_X \end{cases}$$

Pour tout  $a = (l, x, r) \in A_X \setminus \{E\}$ ,  $x$  est appelé la racine de  $a$ .  $l$  et  $r$  sont respectivement les enfants gauche et droit de  $a$ .

**Exemple :** On pose  $X = \mathbb{N}$ . On a  $E$  un arbre (l'arbre vide), donc  $(E, 5, E)$  est un arbre, donc  $((E, 5, E), 2, E)$  est aussi un arbre, etc. . .

**Définition 57 : Hauteur d'un arbre**

Soit  $A_X$  un ensemble d'arbres binaires définies sur un ensemble  $X$ . La hauteur d'un arbre  $a \in A_X$  est "son nombre d'étages". On définit alors la fonction hauteur  $h : A_X \rightarrow \mathbb{N} \cup \{-1\}$  par induction selon les règles :

$$\begin{cases} \mathcal{B} & : h(E) = -1 \\ \mathcal{I} & : \forall a = (g, x, d) \in A_X \setminus \{E\}, h(a) = \max(h(g), h(d)) + 1 \end{cases}$$

**Remarque :** On choisit une hauteur de  $E$  négative pour que la hauteur soit le nombre d'étages de branches. Un arbre constitué d'une racine seule  $(E, x, E)$  n'a pas d'étages. Sa hauteur est donc 0. On utilisera la même notation pour les démonstrations sur les partitions d'ensembles finis en section 9.7.

**Exemple :** sur la figure 9.1 :

- $h((g, x_0, d)) = 2$
- $h((g, x_1, d)) = h((g, x_2, d)) = 1$
- $h((E, x_3, E)) = h((E, x_4, E)) = h((E, x_5, E)) = 0$

**9.3.4 Signature et implantation**

La définition d'un arbre montre que tout arbre peut être construit par composition d'autres arbres. On en déduit la signature :

*ArbreBinaire*<*Element*> utilise *Booleen*, *Element*

- *arbrevide*()  $\rightarrow E$  renvoie la constante  $E$  d'arbre vide.
- *creer*( $x : \text{Element}$ )  $\rightarrow \text{ArbreBinaire}$  renvoie l'arbre  $(E, x, E)$
- *contenu*( $T : \text{ArbreBinaire}$ )  $\rightarrow \text{Element}$  où  $T = (g, x, d)$  renvoie  $x$ . Indéfini si  $T = E$
- *arbre\_droit*( $T : \text{ArbreBinaire}$ )  $\rightarrow \text{ArbreBinaire}$  où  $T = (g, x, d)$  renvoie  $d$ . Indéfini si  $T = E$
- *arbre\_gauche*( $T : \text{ArbreBinaire}$ )  $\rightarrow \text{ArbreBinaire}$  où  $T = (g, x, d)$  renvoie  $g$ . Indéfini si  $T = E$
- *construire*( $g : \text{ArbreBinaire}, x : \text{Element}, d : \text{ArbreBinaire}$ )  $\rightarrow \text{ArbreBinaire}$  renvoie l'arbre  $(g, x, d)$
- *fixer\_contenu*( $T : \text{ArbreBinaire}, x : \text{Element}$ )  $\rightarrow \text{ArbreBinaire}$  où  $T = (g, \dots, d)$  renvoie  $(g, x, d)$
- *fixer\_droit*( $T : \text{ArbreBinaire}, d : \text{ArbreBinaire}$ )  $\rightarrow \text{ArbreBinaire}$  où  $T = (g, x, \dots)$  renvoie  $(g, x, d)$

- $fixer\_gauche(T : ArbreBinaire, g : ArbreBinaire) \rightarrow ArbreBinaire$  où  $T = (\dots, x, d)$  renvoie  $(g, x, d)$
- $chercher(T : ArbreBinaire, x : Element) \rightarrow Booleen$  teste si  $x$  apparaît dans un des noeuds de l'arbre

En C, on peut, par exemple, utiliser des pointeurs d'arbres et poser par convention  $E = \text{NULL}$  :

```
#include "element.h"

typedef struct TreeNode TreeNode; // Tree est un type pointeur vers struct Tree
typedef TreeNode Tree;
struct TreeNode {
    Tree* left;
    Tree* right;
    TreeNode nd;
    Element x;

    // Plus des métadonnées sur le noeud comme :
    Tree* parent; // connaître le parent permet de remonter l'arbre depuis une feuille
    int h; // hauteur de l'arbre dont le noeud est la racine (utile plus tard pour l'équilibrage)
};

Tree* tree_empty() {
    return NULL; // = E
}

Tree* tree_create(Element x) {
    Tree* t = (Tree*)malloc(sizeof(Tree));
    t->left = NULL; // = E
    t->right = NULL; // = E
    t->x = x;

    t->parent = NULL; // la racine n'a pas de parent
    t->h = 0;
    return t;
}

void tree_destroy(Tree* t) {
    if (t != NULL) {
        tree_destroy(t->left);
        tree_destroy(t->right);
        free(t);
    }
}
```

Sans plus de propriétés, les arbres peuvent servir à la classification de données dont on connaît les chemins parmi les branches. C'est le cas par exemple des systèmes de fichiers.

**Arbres en peigne :** Les arbres binaires sont une généralisation des listes. En construisant un arbre avec seulement des enfants droits d'enfants droits d'enfants droits, etc... on obtient un arbre filiforme dit "en peigne"

$$(E, x_1, (E, x_2, \dots, (E, x_n, E) \dots))$$

dont on peut se convaincre aisément qu'il ne s'agit que d'une liste  $[x_1, \dots, x_n]$ . On cherche généralement à éviter ce cas de figure. En effet, l'accès à un élément devient fonction de  $O(n)$  dans le pire des cas,

où  $n$  est le nombre d'éléments de l'“arbre”. Tout l'intérêt d'utiliser un arbre s'évapore. On présentera dans la suite une méthode d'équilibrage de l'arbre.

### 9.3.5 Parcours d'arbres

Les arbres sont des structures par nature récursives, puisque définies par induction. Le parcours des éléments d'un arbre peut donc naturellement être traité récursivement.

On distingue alors 3 types fondamentaux de parcours récursifs<sup>12</sup> : préfixe, suffixe et infixe. La différence est seulement l'ordre dans lequel chacun des noeuds est parcouru. Cet ordre peut avoir une importance si l'arbre possède des propriétés particulières (voir la sous-section 9.3.6)

#### Parcours préfixe

Dans un parcours préfixe, la racine de l'arbre est parcourue/traitée *avant* de parcourir les sous-arbres gauche et droit :

---

##### Algorithme 10 : Parcours récursif *préfixe*

---

PARCOURSPRÉFIXE( $A : \text{ArbreBinaire}, p : \text{Procédure}$ )

---

```

1  suivant A faire
2  |  cas où  $A = E$  faire
3  |  |  // Assure que l'algorithme termine puisque les sous-arbres finissent
3  |  |  tous sur  $E$ 
4  |  cas où  $A = (g, x, d)$  faire
5  |  |   $p(x)$ ;
6  |  |  PARCOURSPRÉFIXE( $g, p$ );
7  |  |  PARCOURSPRÉFIXE( $d, p$ );
```

---

**Construction de la liste de parcours :** Un arbre  $A \neq E$  est la donnée d'un élément  $x$  et de deux sous-arbres  $g$  et  $d$ . Si on écrit le couple sous forme *préfixe*, c'est-à-dire sous la forme  $A = (x, g, d)$ , en écrivant explicitement  $A$  on obtient la liste  $L$ . Ainsi, en omettant les  $E$  :

$$A = (0, (1, (3), (4)), (2, (5)))$$

Ainsi, pour l'arbre 9.1, la liste des sommets parcourus est :

$$L = [0, 1, 3, 4, 2, 5]$$

**Remarque :** le parcours change si on écrit  $A = (x, d, g)$  où  $d$  est le sous-arbre droit de  $A$  et  $g$  le gauche.

**Interprétation :** on descend dans chaque sous-arbre en lisant un noeud dès qu'on le voit.

#### Parcours suffixe

Dans un parcours suffixe, la racine de l'arbre est parcourue/traitée *après* avoir parcouru les sous-arbres gauche et droit :

---

12. On peut effectuer n'importe quel des  $n!$  parcours en soi. Mais ceux-là sont généralisables à tout arbre et peuvent posséder des propriétés utiles de temps en temps (ce qui n'est pas dit de parcours arbitraires).

**Algorithme 11 : Parcours récursif *suffixe***


---

**PARCOURSUFFIXE**( $A : \text{ArbreBinaire}, p : \text{Procedure}$ )

---

```

1 suivant  $A$  faire
2   cas où  $A = E$  faire
3   cas où  $A = (g, x, d)$  faire
4     PARCOURSUFFIXE( $g, p$ );
5     PARCOURSUFFIXE( $d, p$ );
6      $p(x)$ ;

```

---

**Construction de la liste de parcours :** Un arbre  $A \neq E$  est la donnée d'un élément  $x$  et de deux sous-arbres  $g$  et  $d$ . Si on écrit le couple sous forme *préfixe*, c'est-à-dire sous la forme  $A = (g, d, x)$ , en écrivant explicitement  $A$  on obtient la liste  $L$ . Ainsi, en omettant les  $E$  :

$$A = (((3), (4), 1), ((5), 2), 0)$$

Ainsi, pour l'arbre 9.1, la liste des sommets parcourus est :

$$L = [3, 4, 1, 5, 2, 0]$$

**Remarque :** le parcours change si on écrit  $A = (d, g, x)$  où  $d$  est le sous-arbre droit de  $A$  et  $g$  le gauche.

**Interprétation :** on attend d'avoir vu tous les noeuds des sous-arbres avant de voir la racine.

**Exemple d'utilisation :** Par exemple, si une opération d'optimisation doit être effectuée sur un noeud qui dépend de l'opération effectuée sur chaque noeud des sous-arbres, le parcours suffixe permet de "rassembler" un maximum d'information pour avoir la meilleur opération d'optimisation.

### Parcours infixe

Dans un parcours infixe, on traite d'abord le sous-arbre gauche, puis la racine, puis le sous-arbre droit. C'est le parcours infixe qui va être le plus utile dans la suite par sa propriété de conserver un ordre dans la liste de parcours de l'arbre.

**Algorithme 12 : Parcours récursif *infixe***


---

**PARCOURSINFIXE**( $A : \text{ArbreBinaire}, p : \text{Procedure}$ )

---

```

1 suivant  $A$  faire
2   cas où  $A = E$  faire
3   cas où  $A = (g, x, d)$  faire
4     PARCOURSINFIXE( $g, p$ );
5      $p(x)$ ;
6     PARCOURSINFIXE( $d, p$ );

```

---

**Construction de la liste de parcours :** il suffit d'écrire un arbre  $A \neq E$  comme habituellement sous la forme  $A = (g, x, d)$  et de dérécursifier. Ainsi, pour l'arbre 9.1, la liste des noeuds parcourus est :

$$L = [3, 1, 4, 0, 2, 5]$$

### 9.3.6 Arbres binaires de recherche

Les Arbres Binaires de Recherche (ABR pour les intimes) forment l'application la plus commune des arbres binaires. Il s'agit d'un cas particulier dont les propriétés d'ordre sont très intéressantes. On peut par exemple utiliser les ABRs pour trier efficacement des tableaux d'éléments ou pour stocker efficacement des ensembles ordonnés.

L'ensemble  $X$  des éléments d'un ABR doit être ordonné par une relation d'ordre partielle<sup>13</sup>  $\leq_X$ . On peut choisir par exemple  $\mathbb{N}$ ,  $\mathbb{Z}$  ou  $\mathbb{R}$  munis de la relation d'ordre usuelle  $\leq$ .

**Notation :** pour définir formellement les arbres binaires de recherche, on note  $elts : A \rightarrow \mathcal{P}(\mathbb{Z})$  la fonction qui à tout arbre associe l'ensemble des éléments de cet arbre.

On définit formellement  $elts$  par induction :

$$\begin{cases} \mathcal{B} & : \quad elts(E) = \emptyset \\ \mathcal{I} & : \quad \forall a = (g, x, d) \in A_X \setminus \{E\}, elts(a) = elts(g) \cup \{x\} \cup elts(d) \end{cases}$$

#### Définition 58 : Arbre binaire de recherche

On note  $A_X^r$  l'ensemble des arbres binaires de recherche définis sur  $A_X$  par induction par :

$$\begin{cases} \mathcal{B} & : \quad E \in A_X^r \\ \mathcal{I} & : \quad \forall g, d \in A_X^r, \forall x \in X, [\forall (n_l, n_r) \in elts(g) \times elts(d), n_l \leq x \leq n_r] \Rightarrow (g, x, d) \in A_X^r \end{cases}$$

Ce qu'on peut traduire en français par : les éléments du sous-arbre droit sont tous supérieurs ou égaux à la racine de l'arbre qui est elle-même supérieure ou égale à tous les éléments du sous-arbre gauche.

**Remarque :** Un arbre est construit comme des sous-arbres imbriqués. La propriété doit être vraie pour tout sous-arbre  $(g, x, d)$  de l'arbre résultant de la racine primaire (c'est-à-dire celle sans parent)

**Exemple :** avec  $X = \mathbb{N}$  muni de la relation d'ordre usuelle :

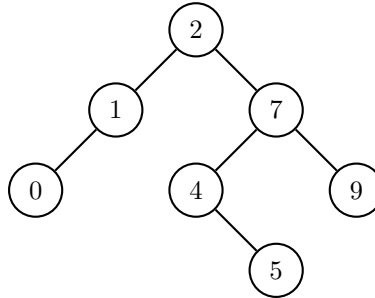


FIGURE 9.2 – Un ABR

#### Signature

La signature d'*ABR* (pour *ArbreBinaireDeRecherche*) est différente de celle d'un *ArbreBinaire* puisqu'il faut supprimer toutes les routines de manipulation qui ne tiennent pas compte des propriétés d'ordre. Il faut en ajouter de nouvelles qui permettent de construire des *ABR*.

13. Un ordre partiel est une contrainte suffisante, puisque le théorème de Szpilrajn affirme (démonstration rapide par lemme de Zorn) que toute relation d'ordre partielle peut être étendue en une relation d'ordre totale sur l'ensemble concerné.



$ABR \langle Element \rangle$  utilise *Booleen*, *Element* :

- $arbrevide() \rightarrow E$  renvoie la constante  $E$  d'arbre vide.
- $creer(x : Element) \rightarrow ABR$  renvoie l'arbre  $(E, x, E)$
- $contenu(A : ABR) \rightarrow Element$  où  $A = (g, x, d)$  renvoie  $x$ . Indéfini si  $A = E$
- $arbre_droit(A : ABR) \rightarrow ABR$  où  $A = (g, x, d)$  renvoie  $d$ . Indéfini si  $A = E$
- $arbre_gauche(A : ABR) \rightarrow ABR$  où  $A = (g, x, d)$  renvoie  $g$ . Indéfini si  $A = E$
- $chercher(A : ABR, x : Element) \rightarrow Boolean$  teste si  $x$  apparaît dans un des noeuds de l'arbre
- $insérer(A : ABR, x : Element) \rightarrow ABR$  renvoie un arbre binaire de recherche contenant  $elts(A) \cup \{x\}$
- $supprimer(A : ABR, x : Element) \rightarrow ABR$  renvoie un arbre binaire de recherche contenant  $elts(A) \setminus \{x\}$

Ces routines sont les seules à pouvoir être utilisés en externe. Les routines elles-même peuvent en interne utiliser des routines de manipulation d'*ArbreBinaire*. Pour rester rigoureux, on peut simplement considérer que les routines d'*ArbreBinaire* restent définis *si et seulement si* elles préservent la structure d'ABR.

### Recherche d'élément

Si on veut trouver  $x \in X$  dans un arbre  $A = (g, y, d)$  et que  $x \neq y$ , alors selon la comparaison d'ordre entre  $x$  et  $y$  on sait dans quel sous-arbre  $g$  ou  $d$  chercher  $x$  :

- si  $x < y$ ,  $x \notin d$  par propriété d'ordre des ABRs, donc on cherche dans  $g$
- si  $x > y$ ,  $x \notin g$  par propriété d'ordre des ABRs, donc on cherche dans  $d$

---

#### Algorithme 13 : Recherche d'un élément

---

**Entrées :**  $A : ABR \langle Element \rangle, x : Element$

```

1 tant que  $A \neq E \wedge contenu(A) \neq x$  faire
2   | si  $x \leq contenu(A)$  alors
3   |   |  $A \leftarrow arbre\_gauche(A);$ 
4   | sinon
5   |   |  $A \leftarrow arbre\_droit(A);$ 
6 retourner  $A \neq E$ 
```

---

**Maximum et minimum :** pour trouver le maximum de l'arbre, il suffit d'aller toujours à droite, pour trouver le minimum de l'arbre, il suffit d'aller toujours à gauche.

### Insertion d'élément

Pour insérer un élément, il suffit de chercher une place pour lui conserve la propriété d'arbre binaire de recherche. Or on sait comment chercher, il suffit d'effectuer des comparaisons successives avec les éléments rencontrés. Quand on descend dans l'arbre et qu'on rencontre un arbre vide, c'est là qu'il faut insérer l'élément.

**Remarque :** on peut dérécursifier l'algorithme assez facilement ou l'écrire en récursif terminal. Dans les deux cas, on retire la nécessité de remonter la pile d'exécution, mais on ajoute plusieurs conditions supplémentaires.

### Suppression d'élément

La suppression d'éléments dans un arbre est légèrement plus délicate pour s'assurer que l'arbre résultant est toujours un ABR. La recherche du noeud à supprimer est identique aux cas de recherche et

**Algorithme 14** : Insertion d'un élémentINSERERELEMENT( $A : ABR, x : Element$ )

```

1 suivant  $A$  faire
2   cas où  $A = E$  faire
3      $\lfloor$  retourner  $(E, x, E)$ 
4   cas où  $A = (g, y, d)$  faire
5     si  $x > y$  alors
6        $\lfloor A \leftarrow fixer\_droit(A, INSERERELEMENT(d, x))$ 
7     sinon
8        $\lfloor A \leftarrow fixer\_gauche(A, INSERERELEMENT(g, x));$ 
9    $\lfloor$  retourner  $A$ 
```

d'insertion. Lorsqu'on trouve le noeud à supprimer  $(g, x, d)$ , si  $d = E$  ou  $g = E$ , il suffit de remplacer le noeud par son seul sous-arbre différent de  $E$ . Dans le cas où chacun des deux sous-arbre est différent de  $E$ , il est très compliqué de supprimer le noeud tel quel. Par contre, on peut chercher à remplacer la valeur du noeud par celle d'un noeud du sous-arbre droit ou du sous-arbre gauche qui conserve l'ordre. Les deux seules valeurs possibles sont  $max(elts(g))$  et  $min(elts(d))$ . On choisit donc arbitrairement entre les deux et on supprime le noeud correspondant après remplacement de  $x$ .

**Tri ABR**

**Proposition 12 (Ordre du parcours infixe).** Le parcours infixe d'un arbre binaire de recherche  $a \in A_X^r$  résulte en une liste triée selon la relation d'ordre sur  $X$ , avec une complexité temporelle fonction de  $\Theta(n)$  où  $n$  est le nombre d'éléments de  $a$ .

**Démonstration :** Le parcours visite les  $n$  noeuds de l'arbre. La complexité est donc évidemment fonction de  $\Theta(n)$ . On montre le tri par induction, c'est-à-dire par *initialisation* pour chaque élément défini par la règle de base  $\mathcal{B}$  puis *hérédité* selon la règle  $\mathcal{I}$ , où  $\mathcal{B}$  et  $\mathcal{I}$  sont les règles qui définissent l'ensemble des arbres binaires de recherches  $A_X^r$ .

Initialisation :  $E$  est sans éléments. Tout parcours de  $E$  est vide donc trié.

Hérédité : Soient  $g, d \in A_X^r$  et  $x \in X$ . Supposons la proposition vraie pour  $g$  et  $d$ . On suppose que pour tout  $(a, b) \in elts(g) \times elts(d)$ ,  $a \leq x \leq b$ . Par la règle d'induction  $\mathcal{I}$ ,  $(g, x, d) \in A_X^r$ . Comme la liste  $L(g)$  résultant du parcours infixe de  $g$  et la liste  $L(d)$  résultant du parcours infixe de  $d$  sont triées, avec  $x$  majorant de  $elts(g)$  et minorant de  $elts(d)$ , alors la liste  $L(g) :: [x] :: L(d)$  est triée. Il s'agit de la liste construite par l'algorithme 12 de parcours infixe sur  $A_X^r$ .

**Remarque :** Si on arrive à construire efficacement un arbre binaire de recherche à partir d'une liste non triée d'éléments, on peut trier efficacement cette liste par l'algorithme 16

Intuitivement, le plus simple est de partir d'un arbre vide et d'ajouter chacun des éléments de la liste en conservant la structure d'arbre binaire de recherche. Cependant, il faut faire attention à ce que l'arbre résultant soit bien "équilibré", notion à définir.

**Arbres filiformes :** Comme on l'a observé précédemment, si l'arbre est *filiforme*, de la forme  $(E, e_1, (\dots, (E, e_n, E) \dots))$ , on a juste une liste. L'ajout du  $i^e$  élément a coûté le parcours de  $i-1$  éléments. La complexité temporelle du tri est alors fonction de  $O(n^2)$  par sommation des  $i$  de 1 à  $n$ . Il faut donc que l'arbre s'équilibre automatiquement au fur-et-à-mesure des insertions et des suppressions d'éléments.

---

**Algorithme 15** : Suppression d'un élément

---

**SUPPRIMERMAX**( $A : ABR, x : Element$ )1 **cas où**  $A = (g, y, E)$  **faire**2   | **retourner**  $g$ 3 **cas où**  $A = (g, y, d \neq E)$  **faire**4   | **retourner**  $(g, y, SUPPRIMERMAX(d, x))$ **SUPPRIMERELEMENT**( $A : ABR, x : Element$ )5 **suivant**  $A$  **faire**6   | **cas où**  $A = E$  **faire**7   |   | **retourner**  $E$ ;               // Element pas présent dans l'arbre : rien à supprimer8   | **cas où**  $A = (g, y, d)$  **faire**9   |   | **cas où**  $x > y$  **faire**10   |   |   |  $A \leftarrow fixer\_droit(A, SUPPRIMERELEMENT(d, x));$ 11   |   | **cas où**  $x < y$  **faire**12   |   |   |  $A \leftarrow fixer\_gauche(A, SUPPRIMERELEMENT(g, x));$ 13   |   | **cas où**  $x = y$  **faire**14   |   |   | **si**  $d = E$  **alors**15   |   |   |   | **retourner**  $g$ 16   |   |   | **si**  $g = E$  **alors**17   |   |   |   | **retourner**  $d$ 18   |   |   |  $Element : m \leftarrow max(g);$ 19   |   |   |  $g \leftarrow SUPPRIMERMAX(g);$ 20   |   |   | **retourner**  $(g, m, d)$ 21   | **retourner**  $A$ 

---

---

**Algorithme 16** : Tri ABR

---

**Entrées** :  $L : Liste<Element>$ 1  $abr : ABR<Element> \leftarrow construire(L);$ 2  $L \leftarrow PARCOURSINFIXE(abr);$ 3 **retourner**  $L$ ;

---

Deux types d'arbres binaire de recherche auto-équilibrés sont communs :

- les arbres *AVL*<sup>14</sup>
- les arbres *rouge-noir*<sup>15</sup>

Chacun est asymptotiquement équivalent en terme de classe de complexité temporelle. Les différences de performance se jouent principalement au niveau des constantes, qui diffèrent selon les opérations.

### 9.3.7 Arbres équilibrés

#### Définition 59 : Équilibre d'un arbre

On définit l'équilibrage d'un arbre par la fonction :

$$\begin{aligned} e : A_X &\rightarrow \mathbb{Z} \\ E &\mapsto 0 \\ (g, x, d) &\mapsto h(d) - h(g) \end{aligned}$$

On dit alors qu'un arbre  $T$  est *équilibré* si et seulement si  $|e(t)| \leq 1$  pour tout sous-arbre  $t$  de  $T$ . Il est dit *déséquilibré* sinon.

**Interprétation :** Si la hauteur d'un sous-arbre d'un noeud est beaucoup plus grande que l'autre, c'est que les noeuds sont mal répartis entre les deux sous-arbres. On borne donc la différence de hauteur au minimum possible pour assurer la répartition des noeuds dans l'arbre.

**Remarque :** on va s'intéresser dans la suite aux arbres qui sont à la fois des

**Exemples :** l'arbre de droite est déséquilibré car  $e(2) = -2$ . Tous les autres noeuds sont équilibrés.

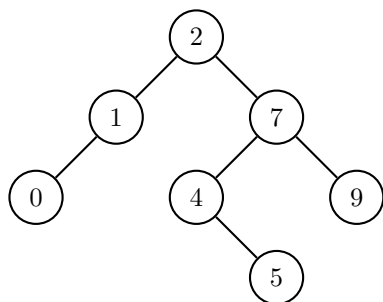


FIGURE 9.3 – ABR équilibré

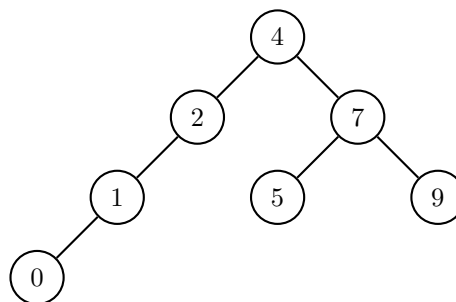


FIGURE 9.4 – ABR déséquilibré

14. Du nom de leur inventeurs : M. Adelson-Velsky et M. Landis. C'est triste que M. Landis ait écopé d'une lettre de moins...

15. Notamment utilisés dans la bibliothèque standard C++ pour les ensembles et les dictionnaires ordonnés.

**Définition 60 : Rotations gauche et droite**

Soit un arbre  $T = ((A, x, B), y, C)$ , avec  $x, y \in X$  et  $A, B, C \in A_X$ .

On appelle *rotation droite* l'opération :

$$R_d(((A, x, B), y, C)) = (A, x, (B, y, C))$$

On appelle *rotation gauche* l'opération  $R_g = R_d^{-1}$  inverse de  $R_d$  :

$$R_g((A, x, (B, y, C))) = ((A, x, B), y, C)$$

Il est directement visible que le parcours infixe de l'arbre est invariant par chacune des rotations. En particulier, un arbre binaire de recherche qui subit des rotations gauche ou droite reste un arbre binaire de recherche

**Visuellement :**

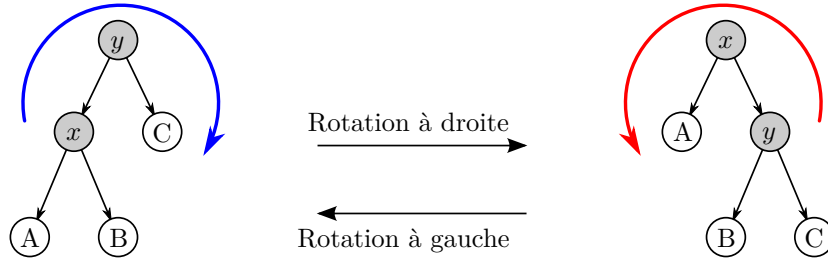


FIGURE 9.5 – Rotations (source : Wikipédia)

**Proposition 13 (Invariance de l'ordre par rotation).** La propriété d'arbre binaire de recherche est invariante par les rotations gauche et droite.

**Démonstration :** voir la définition des rotations<sup>16</sup>

**Proposition 14 (Équilibrage par rotations).** Soit  $T = (g, x, d)$  un arbre déséquilibré dont les sous-arbres sont équilibrés.

- si  $e(T) < -1$ 
  - ◆ si  $e(g) \leq 0$ ,  $R_d(T)$  améliore l'équilibrage de  $T$
  - ◆ si  $e(g) = 1$ ,  $R_d((R_g(g), x, d))$  améliore l'équilibrage de  $T$

C'est-à-dire que  $(e(T) < e(R_d((R_g(g), x, d))) \leq 1$  et  $R_g(g)$  équilibré) ou  $e(T) < R_d(T) \leq 1$
- si  $e(T) > 1$ 
  - ◆ si  $e(d) \geq 0$ ,  $R_g(T)$  améliore l'équilibrage de  $T$
  - ◆ si  $e(d) = -1$ ,  $R_g((g, x, R_d(d)))$  améliore l'équilibrage de  $T$

C'est-à-dire que  $(-1 \leq e(R_g((g, x, R_d(d)))) < e(T)$  et  $R_d(d)$  équilibré) ou  $-1 \leq e(R_g(T)) < e(T)$

**Démonstration** On ne traite que le cas  $e(T) < -1$ . Le second cas  $e(T) > 1$  est symétrique.

Soit  $T = ((A, x, B), y, C)$  un arbre tel que :

- $A$ ,  $B$  et  $C$  sont équilibrés.
- $(A, x, B)$  est équilibré, équivalent à  $|h(A) - h(B)| \leq 1$
- $e(T) < -1$

16. Flemme de poser l'induction immédiate.

- Si  $e(A, x, B) \leq 0$  :

Comme  $e(T) < -1$ , on a  $h((A, x, B)) = \max(h(A), h(B)) + 1 \geq h(C) + 2$ .

D'où  $\max(h(A), h(B)) \geq h(C) + 1$ . Dans tous les cas, on a  $h(A) \geq h(C)$  et  $h(B) \geq h(C)$  par l'équilibrage entre  $A$  et  $B$ .

$$\begin{aligned} e(R_d(T)) - e(T) &= \max(h(C), h(B)) + 1 - h(A) - h(C) + 1 + \max(h(A), h(B)) \\ &= h(B) - h(A) + 1 + \max(h(A), h(B)) + 1 - h(C) \\ &\geq 3 + \max(h(C), h(B)) - h(A) \\ &= 3 + h(B) - h(A) \\ &> 0 \end{aligned}$$

Donc  $e(T) < e(R_d(T))$ . Comme  $h(B) - h(A) \leq 0$  :

$$\begin{aligned} e(R_d(T)) &= \max(h(B), h(C)) + 1 - h(A) \\ &= h(B) + 1 - h(A) \\ &\leq 1 \end{aligned}$$

- Si  $e(A, x, B) = 1$ , on a  $B = (B_1, b, B_2)$ . On a  $h(B_1) = h(A)$  ou  $h(B_1) = h(A) - 1$ . *idem* pour  $h(B_2)$ . Après rotation gauche de  $(A, x, B)$ , on a l'arbre  $R_g(A, x, B) = ((A, x, B_1), b, B_2)$  qui n'est lui pas nécessairement équilibré. Par contre l'arbre  $R_d(R_g(A, x, B), y, C)$  est lui équilibré et ses sous-arbres le sont.

En effet,  $R_d(R_g(A, x, B), y, C) = ((A, x, B_1), b, (B_2, y, C))$ . Comme  $\max(h(B_1), h(B_2)) = h(A)$  et  $\min(h(B_1), h(B_2)) \geq h(A) - 1$ , alors  $(A, x, B_1)$  est équilibré de hauteur  $h(A)$ . On rappelle que  $h(B) + 1 \geq h(C) + 2$  donc  $h(B_2) \geq h(C)$ . Donc l'arbre  $R_d(R_g(A, x, B), y, C)$  est équilibré à l'équilibrage de  $(B_2, y, C)$  près. Si  $h(B) = h(C) + 2$ , l'arbre est équilibré. Si  $h(B) > h(C) + 2$ ,  $(B_2, y, C)$  est plus équilibré que l'arbre originel puisque  $h(B_2) < h(B)$ .

**Important :** on déduit de la démonstration que si  $e(g, x, d) = -2$ , soit  $R_d(g, x, d)$  est équilibré, soit  $R_d(R_g(g), x, d)$  est équilibré. *idem* pour  $e(g, x, d) = 2$ . C'est cette propriété qui est utilisé pour l'implantation les arbres AVLs (voir sous-section 9.3.8), notamment pour garantir l'équilibrage et par suite la complexité temporelle des opérations.

**Exemple :** La rotation droite du sous-arbre filiforme  $((((E, 0, E), 1, E), 2, E)$  de la figure 9.4 rééquilibre l'arbre :

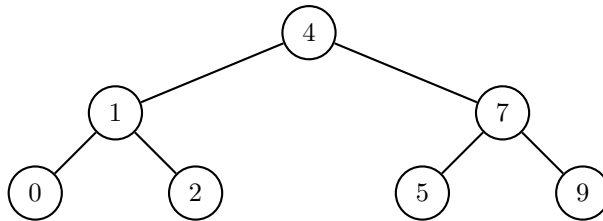


FIGURE 9.6 – ABR rééquilibré

**Proposition 15 (Hauteur d'un arbre strictement équilibré).** La hauteur d'un arbre équilibré de  $n$  éléments est fonction de  $O(\log_2(n))$

**Démonstration :** On note alors  $t(h)$  le nombre minimum d'élément d'un arbre équilibré  $T$  de hauteur  $h$ . Pour maximiser la hauteur par rapport au nombre d'éléments, il faut que cet arbre soit déséquilibré d'un côté ou de l'autre, c'est-à-dire  $e(T) \in \{-1, 1\}$ . Pour maximiser la hauteur totale de  $T$ , on suppose

que tout sous-arbre de  $T$  est déséquilibré ainsi. On a donc :

$$\begin{cases} t(0) &= 1 \\ t(1) &= 2 \\ t(h) &= \underset{\text{plus grand sous-arbre}}{t(h-1)} + \underset{\text{plus petit sous-arbre}}{t(h-2)} + \underset{\text{racine}}{1} \end{cases}$$

On pose pour  $h \in \mathbb{N}$ ,  $F(h) = t(h) + 1$   $t(h+2) = t(h+1) + t(h) + 1$  devient donc  $F(h+2) = F(h+1) + F(h)$ . L'équation caractéristique  $r^2 - r - 1 = 0$  admet les deux racines

$$r_1 = \phi = \frac{1 + \sqrt{5}}{2} \text{ et } r_2 = \frac{1 - \sqrt{5}}{2}$$

Il existe donc  $\alpha, \beta \in \mathbb{R}$  tels que :

$$F(h) = \alpha r_1^h + \beta r_2^h$$

On a  $F(0) = \alpha + \beta = 2$  et  $F(1) = \alpha \frac{1 + \sqrt{5}}{2} + \beta \frac{1 - \sqrt{5}}{2} = 3$

D'où  $\alpha = 1 + \frac{2}{\sqrt{5}}$  et  $\beta = 1 - \frac{2}{\sqrt{5}}$ .

Finalement,

$$t(h) = \left(1 + \frac{2}{\sqrt{5}}\right) \phi^h + \left(1 - \frac{2}{\sqrt{5}}\right) \left(\frac{1 - \sqrt{5}}{2}\right)^h - 1$$

Comme  $\left|\frac{1 - \sqrt{5}}{2}\right| < 1$ , il existe  $C_1, C_2 \in \mathbb{R}$  telle que  $C_2 \phi^h \leq t(h) \leq C_1 \phi^h$

Si un arbre équilibré a  $n$  élément, sa hauteur maximum  $h(n)$  est fonction de  $O(\log_\phi(n)) = O(\log_2(n))$

### 9.3.8 Arbres AVL

#### Définition 61 : Arbre AVL

Un arbre AVL est un arbre binaire de recherche strictement équilibré. <sup>a</sup>

<sup>a</sup>. Selon le Cormen, référence du domaine. Wikipédia dit qu'un arbre AVL est juste un arbre équilibré. Mais alors, pourquoi Knuth dit "balanced tree" et pas "AVL tree"? Parce-que l'article de Wikipédia a été écrit avec un manque de précision TERRIBLE, TERRIBLE je vous dis!

Les opérations d'insertion, de suppression et de recherche sont donc de complexité temporelle fonction de  $O(\log_2(n))$ , où  $n \in \mathbb{N}$  est le nombre d'éléments de l'arbre.

### 9.3.9 Exercices

**Exercice 67 (Type abstrait d'arbre binaire) []**. Planter en C la signature d'*ArbreBinaire* présenté dans le cours :

1. d'abord avec les enregistrements proposés dans le cours
2. ensuite avec un tableau qui contient tous les noeuds de l'arbre

Discuter les performances de chaque implantation.

**Exercice 68 (Arbres binaire de recherche) [23]**. Planter en C la signature d'*ArbreBinaireDeRecherche*.  
*Remarque : identique à la structure d'un arbre binaire. La seule différence est dans le corps des routines, qui doit préserver la structure d'ordre entre les étiquettes.*

**Exercice 69 (Arbres AVL) [31].**

Implanter des arbres AVLs en autoéquilibrant les ABRs de l' **Exercice 68** par des rotations durant les opérations d'insertion et de suppression.

*Notes : il faut faire attention à ne pas oublier de modifier le pointeur vers le parent, ou les prochaines rotations seront fausses. Utiliser un parcours récursif pour simplifier le code n'est pas absurde puisque la hauteur d'un arbre est logarithmique et que le surcoût des appels est donc relativement faible.*

**Exercice 70 (Arbres dimensionnels) [].****Exercice 71 (Les tries) [].****Exercice 72 (Codage de Huffman) [].****Exercice 73 (Arbres auto-ajustés) [].**





Il peut être intéressant de stocker des éléments selon un ordre et de vouloir très rapidement trouver le minimum ou le maximum de ces éléments selon l'ordre choisi. Quelques exemples sont donnés pour montrer l'intérêt d'une structure spécialisée pour cette opération.

**Exemple (*context-switching*) :** dans un système d'exploitation, les tâches sont exécutées en alternance pour simuler leur exécution parallèle. On veut sélectionner des tâches

- qui depuis longtemps n'ont pas été traités
- qui sont demandantes en ressources

On peut associer à chaque tâche une pondération de chaque critère qui correspond à sa *priorité*. On veut alors à chaque changement de tâche sélectionner celle de plus haute priorité. Comme le changement de tâche en lui-même doit consommer très peu de ressources, on veut une structure de données efficace, donc spécialisée pour ce but.

**Exemple (algorithmes gloutons) :** de nombreux algorithmes cherchent à optimiser numériquement des solutions en cherchant d'abord un optimum local dans l'espace des solutions. Si les différents choix sont organisés selon une file de priorité, la sélection de l'optimum local est très rapide et accélère grandement l'algorithme. C'est ce qu'on retrouve dans l'algorithme de *Dijkstra* pour la recherche d'un plus court chemin dans un graphe.

### 9.4.1 Signature

**Première remarque :** il suffit de définir une structure de donnée permettant de trouver rapidement le minimum des éléments. En effet, il suffit juste d'inverser la relation d'ordre pour obtenir une structure permettant de trouver rapidement le maximum des éléments.

*Element* désigne un type de donnée sur lequel est défini une relation d'ordre totale<sup>17</sup>

*FilePriorite*<*Element*> utilise *Booleen*, *Element*

- *file\_vide()* → *FilePriorite* renvoie une file de priorité sans éléments
- *est\_vide(f : FilePriorite)* → *Booleen* teste si la file *f* est vide.
- *minimum(f : FilePriorite)* → *Element* renvoie l'élément minimal de la file de priorité
- *supprimer\_min(f : FilePriorite)* → *FilePriorite* renvoie *f* dont on a retiré l'élément minimal
- *insérer(f : FilePriorite : f, x : Element)* → *FilePriorite* renvoie *f* à laquelle on a ajouté *x*

On construit la routine de commodité *extraire\_min*(mut *FilePriorite* : *f*) → *Element* qui renvoie l'élément minimal de *f* et le supprime de *f*.

On implanter les files de priorité par des arbres AVLs. Les complexités temporelles des routines *minimum* *supprimer\_min* et *insérer* sont alors des fonctions de  $O(\log_2(n))$ , où *n* est le nombre d'éléments dans la file de priorité.

On présente ici une autre implantation plus efficace et bien plus simple à mettre en oeuvre, le *tas minimum*.

<sup>17</sup>. Partielle suffit en soi d'après le théorème de Szpilrajn.

### 9.4.2 Arbres complets

Les tas minimums forment un cas particulier d'arbres presque complets. On définit donc d'abord cette notion et comment on peut implanter très facilement et très efficacement des arbres presque complets.

#### Principe

##### Définition 62 : Arbres presque complets

L'ensemble  $A_X^p$  des *arbres presque complets* sur un ensemble ordonné  $(X, \leq)$  est défini par induction par :

$$\begin{cases} \mathcal{B} & : E \in A_X^p \\ \mathcal{I} & : \forall g, d \in A_X^p, \forall x \in X, (0 \leq h(g) - h(d) \leq 1) \Rightarrow (g, x, d) \in A_X^p \end{cases}$$

En français : pour tout sous-arbre  $(g, x, d)$  d'un arbre,

- le sous-arbre gauche est toujours plus haut que le sous-arbre droit
- le sous-arbre est équilibré

**Exemple :** on représente ci-dessous les 9 premiers arbres presque complets :

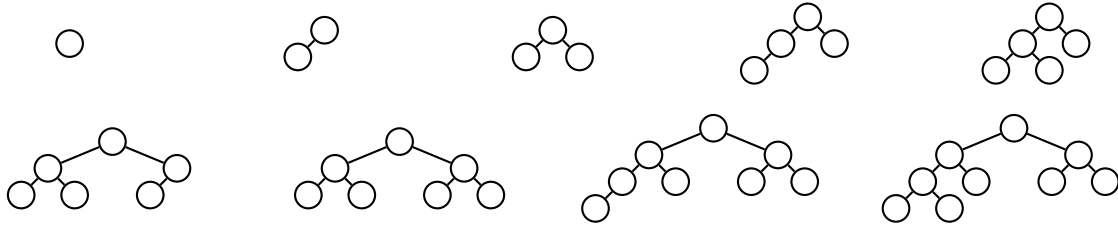


FIGURE 9.7 – les 9 premiers arbres presque complets

#### Implantation par tableau

On observe qu'un arbre presque complet à  $n$  étages possède  $n - 1$  étages complets et 1 étage rempli à partir de la gauche.

On peut compter le nombre de noeuds par étages :

- l'étage 0 a  $2^0$  noeuds
- l'étage 1 a  $2^1$  noeuds
- ...
- l'étage  $0 \leq i < n - 1$  a  $2^i$  noeuds
- ...

On veut faire correspondre à chacun des noeuds  $\{2^i - 1, \dots, 2^{i+1} - 2\}$  d'un étage ses deux sous-arbres gauche et droit. C'est-à-dire qu'on veut trouver une correspondance *injective*<sup>18</sup> de  $\{2^i - 1, \dots, 2^{i+1} - 2\}$  dans  $\{2^{i+1} - 1, \dots, 2^{i+2} - 2\} \times \{2^{i+1} - 1, \dots, 2^{i+2} - 2\}$

On remarque que :

$$\begin{array}{ccc} c & : & \{2^i - 1, \dots, 2^{i+1} - 2\} \rightarrow \{2^{i+1} - 1, \dots, 2^{i+2} - 2\}^2 \\ n & & \mapsto (2n + 1, 2n + 2) \end{array}$$

18. Une correspondance est une application à valeur dans un produit cartésien (i.e "une application qui renvoie plusieurs valeurs")

est une telle correspondance. Elle est représenté par le schéma suivant :

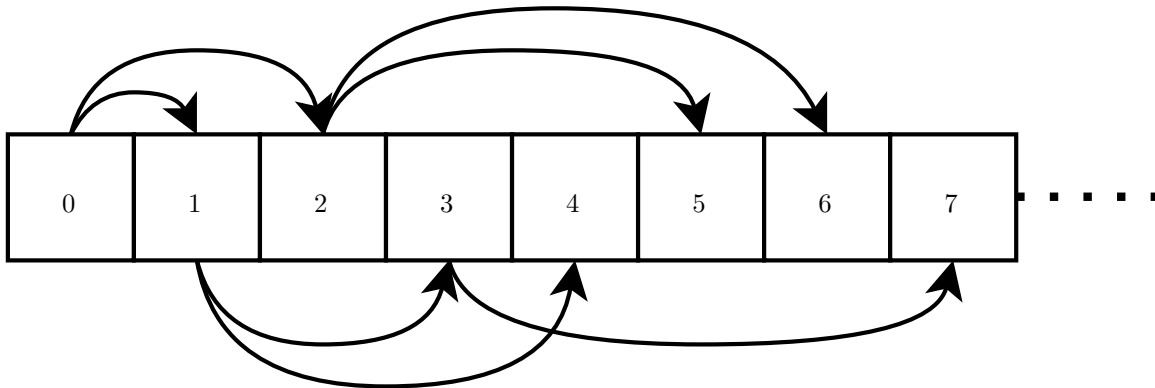


FIGURE 9.8 – Représentation par tableau d'un arbre presque complet

Comme les noeuds sont numérotés de gauche à droite, si la fin du tableau ne remplit pas un étage, c'est toujours un sous-arbre gauche qui est plus haut qu'un sous-arbre droit. Il s'agit donc bien d'un arbre presque complet.

```
typedef struct AlmostCompleteTree {
    // Éléments :
    Element *array;
    // nombre d'éléments
    unsigned int size;
    // taille total du tableau (i.e. nb max d'éléments avant redimensionnement) :
    unsigned int capacity;
} ACT;
```

### 9.4.3 Tas minimum

On décrit ici une implantation efficace des files de priorité grâce à la structure de donnée de *Tas minimum*.

#### Définition 63 : Arbre tournoi

L'ensemble  $A_X^t$  des *arbres tournois* sur un ensemble ordonné  $(X, \leq)$  est défini par induction par :

$$\begin{cases} \mathcal{B} & : E \in A_X^t \\ \mathcal{I} & : \forall g, d \in A_X^t, \forall x \in X, [(\forall n_l \in \text{elts}(g), x \leq n_l) \wedge (\forall n_r \in \text{elts}(d), x \leq n_r)] \Rightarrow (g, x, d) \in A_X^t \end{cases}$$

En français : la racine d'un arbre tournoi est plus petite que tous les éléments des sous-arbres gauche et droit.

**Exemple :** avec  $X = \mathbb{Z}$

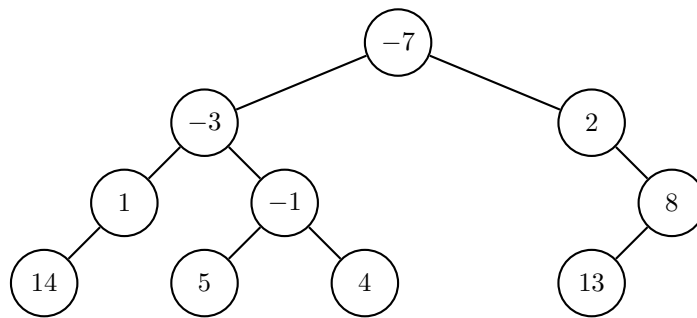


FIGURE 9.9 – Un arbre tournoi

**Proposition 16 (Croissance des chemins descendants).** Un arbre binaire est un arbre tournoi *si et seulement si* tous les chemins de la racine aux feuilles sont croissants selon les valeurs des noeuds.

**Démonstration :** par induction, découle immédiatement de la définition

#### Définition 64 : Tas

Un *tas* est un arbre de tournoi presque complet.

**Exemple :**

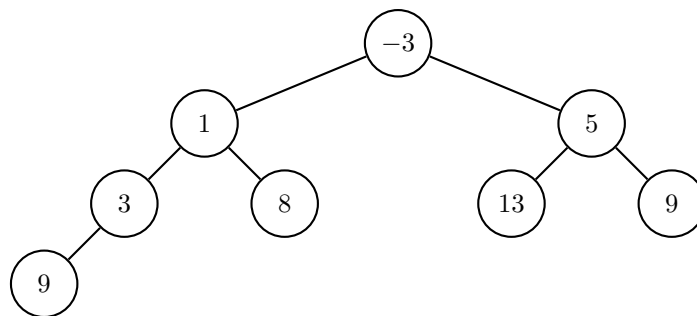


FIGURE 9.10 – Un tas minimum

On peut donc représenter un tas par arbre presque complet, lui-même représenté par un tableau :

```
| typedef ACT Heap;
```

Il faut simplement s'assurer lors de l'insertion ou la suppression d'éléments de conserver la structure d'arbre tournoi.

### 9.4.4 Mutateurs

On décrit dans cette sous-section les deux opérations d'insertion et de suppression dans un tas. En effet, il faut préciser la manière de conserver la structure de tas, donc à la fois d'arbre presque complet mais aussi et surtout de tournoi.

**Opérations de tamisage :** On désigne par :

- *tamisage bas* l'opération qui consiste à faire *descendre* un élément dans un tas jusqu'à ce qu'il soit plus grand que son élément parent dans l'arbre et plus petit que les éléments de ses sous-arbres

gauche et droit

- *tamissage haut* l'opération qui consiste à faire *monter* un élément dans un tas jusqu'à ce qu'il soit plus grand que son élément parent dans l'arbre et plus petit que les éléments de ses sous-arbres gauche et droit

La descente et la montée se font par échanges successifs d'éléments.

## Insertion

L'opération est assez simple : on ajoute un élément à la fin du tableau et par tamissage haut on conserve une structure de tas :

```

1  Heap heap_insert(Heap h, int x) {
2      h.size++;
3      h.array[h.size - 1] = x;
4      heap_top_sieve(h, h.size - 1); // tamissage haut de l'élément d'indice 'h.size - 1' = dernier
5      if (h.size >= h.capacity) {
6          Element *new = malloc(sizeof(Element) * (h.capacity << 1));
7          for (unsigned int i = 0; i < h.capacity; i++) {
8              new[i] = h.array[i];
9          }
10         free(h.array);
11         h.array = new;
12         h.capacity = h.capacity << 1;
13     }
14     return h;
15 }
```

## Suppression du minimum

Si on supprime simplement la racine du tas, ce n'est plus tas. Il faut donc remplacer la racine par un nouvel élément. Il faudrait alors choisir entre la racine du sous-arbre droit ou du sous-arbre gauche, etc... Il s'agit en quelque sorte d'un tamissage bas d'un *trou*. C'est étrange.

Le plus simple reste d'échanger la racine du tas avec le dernier élément du tableau, et d'effectuer un tamissage bas sur cette nouvelle racine :

```

Heap arrayheap_pop_min(Heap h) {
    Element x = h.array[0];
    array_switch(h.array, 0, h.size - 1); // échange avec ou sans effet de bord
    h.size--; // supprimer l'élément
    heap_bottom_sieve(h, 0); // tamissage bas de l'élément d'indice '0'
    return x;
}
```

### 9.4.5 Exercices

**Exercice 74 (Tamisages) [23].** On considère des tas d'entiers dans la suite.

1. Écrire une procédure `void heap_top_sieve(int *heap, int i);` qui effectue un tamissage haut de l'élément d'indice  $i$  dans un tas

2. Écrire une procédure `void heap_bottom_sieve(int *heap, int i);` qui effectue un tamisage bas de l'élément d'indice  $i$  dans un tas
3. Démontrer la correction de chacune des procédures

**Exercice 75 (Complexité de la signature) [15].** Quelle est la complexité temporelle de chaque routine de manipulation de *FilePriorite* implanté par un tas minimum ? Justifier. En particulier, justifier que le redimensionnement du tableau lors de l'insertion d'éléments n'affecte pas la complexité temporelle amortie de l'insertion, c'est-à-dire que pour tous  $n \in \mathbb{N}$ , le coût de  $n$  insertions dans une file vide est fonction de  $O(n \log_2(n))$  dans le pire des cas.

**Exercice 76 (Tableau en tas) [].** On s'intéresse à la construction d'un tas à partir d'un tableau de  $n$  éléments.

1. Si on part d'un tas vide et qu'on ajoute un à un par *insérer* chacun des éléments du tableau, quelle est le coût temporel dans le pire des cas de la construction, selon  $n$  ?
2. Proposer une seconde méthode permettant de construire un tas à partir des éléments d'un tableau dont le coût temporel dans le pire cas soit fonction de  $O(n)$

**Exercice 77 (Tri par tas) [12].** Proposer et implanter en C un algorithme de tri de tableaux qui utilise un tas et dont la complexité temporelle est dans le pire cas fonction de  $O(n \log_2(n))$ , où  $n$  désigne la longueur du tableau en entrée.



Sont à ajouter :

- les analyses de complexité
- le *load factor* et son seuil d'agrandissement d'une table
- correction des exercices

### 9.5.1 Introduction

Un problème important de calcul est la mise en correspondance d'éléments d'un premier ensemble avec des éléments d'un second ensemble. C'est-à-dire qu'on veut pouvoir *associer* à un premier élément de type fixé un second élément d'un autre type également fixé.

#### Définition 65 : Clefs et valeurs

On appelle donc dans la suite :

- *clef* : l'élément dont on cherche le correspondant
- *valeur* : le correspondant d'une clef

L'*espace des clefs* est l'ensemble des valeurs possibles que peuvent prendre les clefs.

**Remarque :** Les clefs et les valeurs peuvent être de nature différente. Il peut s'agir d'associer à une donnée simple comme une chaîne de caractères un objet complexe décrit par un enregistrement.

L'exemple le plus typique est celui des mots du dictionnaire auxquels on associe leur définition. Il faut pouvoir :

- ajouter/supprimer des mots efficacement
- lire la définition d'un mot efficacement

On appelle donc dans la suite un *dictionnaire* une structure de donnée permettant une telle correspondance, puisque toutes les mises en correspondance peuvent se ramener au cas du dictionnaire (ensemble fini de clefs différentes, taille fini de chaque définition, un type pour les clefs, un type pour les valeurs).

#### Exemples réels d'applications :

- les différents niveaux de la mémoire cache d'un processeur fonctionnent en construisant électroniquement un dictionnaire. Les clefs sont (grossièrement) des adresses mémoires de la RAM et les valeurs sont les données devant être stockées *in fine* à ces adresses
- associer à chaque utilisateur (la clef peut être une adresse mail) d'une application ses données personnelles (en 2018, 5% de la RAM des serveurs de Google étaient utilisée pour stocker des dictionnaires)

### Une tentative naïve

L'idée la plus simple semble être d'utiliser une liste chaînée de mots :

```

1 struct Word {
2     char* word;
3     char* definition;
4 };

```

```

5 |
6 | ... // Définition d'une liste chaînée WordList contenant des 'Word'
7 |
8 | WordList words;

```

On peut bien insérer et supprimer des couples (*mots, définition*) efficacement. Le problème est alors qu'il n'est pas possible de lire la définition d'un mot quelconque sans parcourir la liste jusqu'à le trouver. Les arbres ne servent pas plus puisqu'il est toujours nécessaire de parcourir chaque noeuds/feuilles de l'arbre pour comparer le mot.

Les structures vues jusqu'à présent sont des structures de stockage qui se prêtent très mal à la correspondance. Ce dont on rêve est une structure telle qu'il soit possible d'écrire :

```

1 | char* definition = structure_find(structure, "horloge");

```

sans avoir à parcourir l'entièreté des éléments de la structure. C'est-à-dire que les coûts de lecture/écriture soient en  $O(1)$

## Signature

Le type abstrait qui décrit un tel comportement est appelé un *dictionnaire*.

*Dictionnaire*<*Clef*, *Element*> utilise *Booléen*, *Entier*, *Clef* et *Element*

- *creer\_vide()* → *Dictionnaire* renvoie un dictionnaire vide, sans couple (*clef, valeur*)
- *est\_vide(Dictionnaire)* → *Booléen* teste si le dictionnaire contient au moins un couple
- *trouver(Dictionnaire, Clef)* → *Element* renvoie la valeur d'une clef dans le dictionnaire, indéfini si la clef n'appartient pas au dictionnaire
- *supprimer(Dictionnaire, k : Clef)* → *Dictionnaire* supprime le couple de clef *k* du dictionnaire
- *insertion(Dictionnaire, Clef, Element)* → *Dictionnaire* ajoute un couple (*clef, valeur*) au dictionnaire, indéfini si la clef appartient au dictionnaire
- *modifier(Dictionnaire, Clef, Element)* → *Dictionnaire* remplace dans le dictionnaire la valeur de la clef, indéfini si la clef n'appartient pas au dictionnaire
- *taille(Dictionnaire)* → *Entier* renvoie le nombre de couples (*clef, valeur*) stockés dans le dictionnaire.

On observe que les routines *insertion* et *modifier* sont complémentaires. On peut donc l'implanter avec une unique routine qui ajoute le couple si la clef n'existe pas déjà et le met à jour sinon.

**Remarque :** les tableaux sont l'implantation d'un cas particulier des dictionnaires où les indices du tableau sont les clefs et dont certaines cases ne sont simplement pas utilisées. En fait, si on arrive à transformer par un calcul les éléments d'un ensemble de clefs dans un ensemble borné d'entiers, on peut utiliser ce premier ensemble de clefs comme pour indiquer le tableau *via* la transformation, et on a une implantation générique d'un dictionnaire au prix seulement du calcul de la transformation.

C'est cette remarque qui guide l'implantation par *tables de hachages*.

### 9.5.2 Tables et fonctions de hachage

Il faut pouvoir *calculer* à partir de la clef la place de la valeur. Ce calcul doit être en temps constant, au moins en moyenne, et l'accès à la valeur depuis la place également. Le calcul de la place est



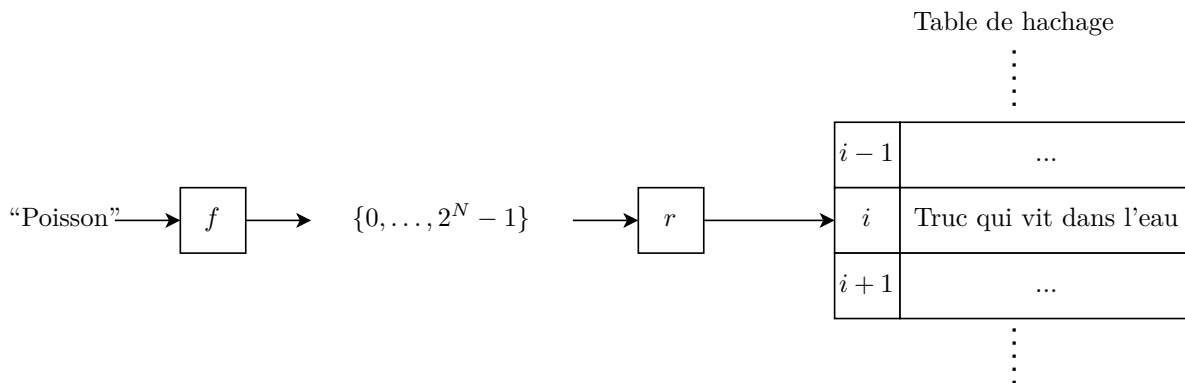
nécessairement effectué par une fonction<sup>19</sup>, qui prend en argument la clef et renvoie la place de la valeur. Comme l'ensemble des clefs possibles est *a priori* non borné<sup>20</sup> et qu'on veut par notre fonction calculer une place, qui est évidemment bornée, il va être nécessaire de supprimer des informations des clefs de départ. On appelle *fonction de hachage* la fonction qui "hache" (ou découpe) la donnée pour la faire "rentre" dans un ensemble borné. L'ensemble des clefs est potentiellement très grand, et est très peu rempli<sup>21</sup>. La fonction de hachage joue donc un rôle de repliement de l'espace de départ dans l'espace d'arrivée.

Comme la fonction de hachage est arbitraire, on peut choisir n'importe quel type pour représenter les places et les valeurs à ces places. Par souci de simplicité et d'optimisation, on considère donc un tableau de valeurs, dont les indices sont les places. Ce tableau est appelé une *table de hachage*.

On a finalement :

- une *table de hachage*  $T$  qui contient des valeurs accessibles par des indices  $i \in \llbracket 0; |T| - 1 \rrbracket$ .  $|T|$  désigne la capacité du tableau, et pas le nombre d'éléments stockés dans le dictionnaire.
- une *fonction de hachage*  $f$  qui permet de replier l'ensemble des clefs dans un ensemble borné (d'entiers ou en bijection avec des entiers). En général, il s'agit de l'anneau  $\frac{\mathbb{Z}}{2^N\mathbb{Z}}$  des entiers représentables sur  $N$  bits.
- une *fonction de réduction*  $r$  qui donne à partir du résultat de  $f$  une place dans la table de hachage.

La distinction entre la première fonction de hachage et la fonction de réduction rend la fonction de hachage choisie indépendante de la taille du tableau et de la représentation des places. On assure par exemple de pouvoir définir des fonctions de hachage sur plusieurs ensembles différents qui ne dépendent pas de la taille du tableau. C'est la réduction qui replie l'image de  $f$  en espace de places.

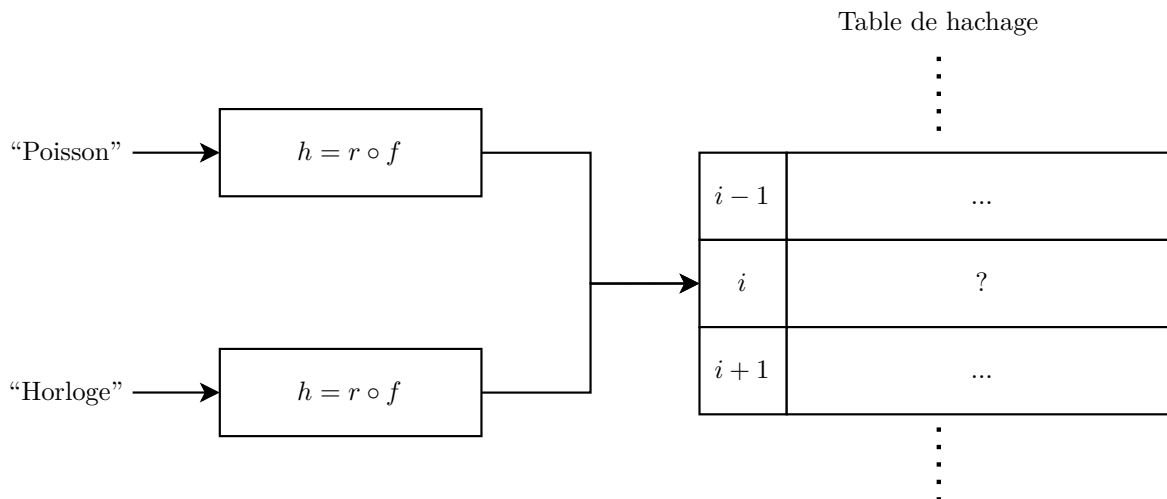


**Collisions :**  $f$  est indépendante de la taille du tableau et peut produire une sortie hors du tableau. Toutefois, l'image de la réduction est, elle, dans un ensemble fixe plus petit. Certaines clefs donneront donc le même indice dans la table de hachage, quelque soit la fonction choisie. On appelle ce phénomène une *collision*. Par exemple :  $r(f(\text{"Horloge"})) = r(f(\text{"Poisson"}))$

19. Puisqu'il s'agit d'un calcul.

20. Avec un clef textuelle, il y a une infinité de lettres et de permutations de lettres possibles

21. Par exemple, il y a très peu de mots dans le dictionnaire par rapport à toutes les possibilités de permutation de lettres.



Les questions sont alors :

- quelles fonctions  $f$  et  $r$  choisir pour calculer des indices dans la table en garantissant un minimum de collisions ?
- comment gérer efficacement les collisions pour ne pas dégénérer la complexité temporelle de la structure ?

Dans le cadre des tables de hachages, le plus important est la vitesse de calcul, donc :

- une répartition homogène de  $r$  dans l'intervalle des indices de la table
- un calcul (très) rapide des fonctions  $f$  et  $r$

De fait, la nécessité d'un calcul très rapide des fonctions de hachage et de réduction peut amener dans l'implantation à confondre les deux, c'est-à-dire à considérer *de fait* la donnée en entrée comme un entier<sup>22</sup>, à utiliser l'identité comme fonction de hachage, et à se concentrer sur la réduction qui occupe alors en partie le rôle de fonction de hachage.

### Bonne ou mauvaise fonction de hachage

*a.* Bon t'a la mauvaise fonction de hachage, elle voit un truc elle hache. Elle hache ! La bonne fonction de hachage, elle voit un truc... Bon elle hache mais euh... c'est une bonne fonction de hachage hein !

On ne s'intéresse pas à la sécurité engendré par la fonction de hachage. Ce ne sont pas ses propriétés cryptographiques qui intéressent mais seulement ses propriétés de répartition et de calculabilité. On veut une fonction *rapidement calculable* qui répartissent de manière "homogène" ses sorties sur  $\llbracket 0; |T| - 1 \rrbracket$

On peut mesurer la qualité d'une fonction de hachage cryptographique par le *critère strict d'avalanche* : chaque fois qu'un seul bit de l'entrée est modifié, chaque bit de sortie doit changer avec une probabilité 0.5. Ce critère assure l'homogénéité de la sortie et l'impossibilité de prédire le résultat de la fonction à cause de biais statistiques.

Dans la pratique... on s'en fout, tant que c'est empiriquement homogène et surtout très rapide à calculer.

<sup>22.</sup> Ce qui ne pose pas de problèmes puisque le mot binaire représentant la donnée peut être interprété comme un entier.

### 9.5.3 Fonction de hachage

Les fonctions qui ne génèrent pas de collisions, c'est-à-dire qui ne génèrent pas deux fois la même valeur avec deux clefs donnés, sont relativement rares. Un exemple clair est donné par le paradoxe des anniversaires. Avec plus de 23 personnes choisies aléatoirement, la probabilité que deux personnes parmi ces plus de 23 soient nées le même jour est très élevée. Ainsi, si on choisit aléatoirement une réduction, et qu'on veuille placer 23 couples (*cle*, *valeur*) dans un tableau de 365 cases, la probabilité que deux hash parmi ces 23 aient le même résultat par la réduction est de pratiquement 0.5.

Une fonction de hachage ne peut donc être choisie au hasard.

On présente là plusieurs fonctions simples qui suffisent amplement à la plupart des cas d'applications pratiques. Plus une fonction de hachage est complexe, plus elle est lente à calculer. Hors du cadre de la cryptographie, c'est la vitesse de calcul qui prime. Et la cryptographie sort très largement du cadre de ce livre.

Les fonctions de hachage présentées ci-dessous sont toutes à valeurs dans  $\{0, \dots, M-1\}$ , avec  $M \in \mathbb{N}^*$ . On considère  $K$  l'interprétation entière du mot binaire à hacher.

#### Hachage par division

Le hachage par modulo est le plus simple de tous. On pose alors :

$$f(K) = K \bmod M$$

où  $M$  doit être bien choisi. Le hachage est bien meilleur si  $M$  est premier puisque les restes par le modulo sont tous différents. D'autres valeurs amènent à un hachage particulièrement médiocre. Ainsi, si  $M = 2^p$ ,  $f$  est simplement la sélection des  $p$  bits de poids faibles de  $K$ .

On observe que le hachage par modulo n'est pas une bonne fonction de hachage selon le critère strict d'avalanche. Si on modifie les bits de poids faible de  $K$  avec  $K \bmod M < M$ , seuls les bits de poids faibles de  $f(K)$  seront modifiés car la "barre" du modulo ne sera pas dépassée et modifier  $K$  revient alors à modifier  $f(K)$ .

#### Hachage multiplicatif

On pose  $w$  le nombre de bits d'un mot machine (*a priori* 32 bits) et alors  $W = 2^w$

Le hachage multiplicatif est effectué par la fonction :

$$f_c(K) = \left\lfloor \frac{M}{W} (cK \bmod W) \right\rfloor$$

dont la qualité dépend du choix de  $c$ . On choisit alors  $c$  premier avec  $W$ .

**Justification du choix de  $c$  :** Pour une fonction de hachage fonctionnelle, on veut *a minima* assurer  $K_1 \neq K_2 \Rightarrow f(K_1) \neq f(K_2)$  pour  $0 \leq K_1, K_2 < W$ . Si  $c$  est premier avec  $W$ , il existe d'après le théorème de Bachet-Bézout  $A, B$  tels que  $Ac + BW = 1$ . On a alors :  $KcA = KBW + K$ .

Si  $0 \leq K < W$ , il s'ensuit :

$$\begin{aligned} K &= AKc \bmod W \\ &= A(Kc \bmod W) \bmod W \\ &= Af(K) \bmod W \end{aligned}$$

Supposons  $K_1 \neq K_2$ . Si  $f(K_1) = f(K_2)$ , on a  $K_1 = K_2$ , c'est absurde.

D'où  $K_1 \neq K_2 \Rightarrow f(K_1) \neq f(K_2)$ .

On remarque que le calcul de  $f$  ne nécessite aucune division. En effet, le modulo de  $2^w$  est effectué naturellement par le processeur et  $\frac{M}{W} = 2^{m-w}$ . D'où :

```

1 | #define C ...
2 | #define m ... // M = 2^m
3 | uint32_t f(uint32_t K) {
4 |     return (C * K) >> (m - sizeof(K))
5 | }
```

### Hachage par pliage

Le hachage par pliage consiste à découper le mot binaire à hacher en sous-mots de  $m$  bits, où  $M = 2^m$  et à superposer ces sous-mots les uns sur les autres. On note  $K = (K_{N-1} \dots, K_0)_2$ , avec  $N \in \mathbb{N}$ . Si  $N \bmod m \neq 0$ , il existe  $q$  tel que  $\lfloor \frac{N}{m} \rfloor = q$ , on étend  $K$  avec des 0 en posant  $K_N, \dots, K_{qm} = 0$

$$f(K) = ((K_0, \dots, K_{m-1})_2 \star \dots \star (K_{(q-1)m}, \dots, K_{qm-1})) \bmod M$$

Si  $\star$  est une opération bit-à-bit, le modulo n'a pas besoin d'être calculé. On peut choisir par exemple  $\star = \oplus$ . Finalement :

$$f(K) = \bigoplus_{i=1}^q (K_{(i-1)m}, \dots, K_{im-1})_2$$

Cette méthode ne peut être efficace que pour des clefs de taille très supérieure à  $M$ .

#### 9.5.4 Réduction

Supposons que la table  $T$  utilisé soit de capacité  $|T| \in \mathbb{N}$ . Pour toute clef  $k$ , il faut donc  $0 \leq r(f(k)) < |T|$ . En particulier,  $r$  dépend de  $T$ . On écrit donc dès à présent  $r(T, f(k))$ .

Idéalement, on veut exactement  $\frac{|Im(f)|}{|T|}$  clefs pour chaque indice dans l'intervalle  $\llbracket 0; |T| - 1 \rrbracket$ . Ce n'est possible que si  $|Im(f)|$  est divisible par  $|T|$ , ce qui n'est généralement pas le cas. Au mieux, on peut avoir soit  $\left\lfloor \frac{|Im(f)|}{|T|} \right\rfloor$  soit  $\left\lceil \frac{|Im(f)|}{|T|} \right\rceil$  valeurs pour chaque indice. Si  $|T|$  est petit par rapport à  $|Im(f)|$ , on peut considérer la réduction comme parfaite.

Le modulo est à nouveau la solution la plus simple :

$$r(T, f(K)) = f(K) \bmod |T|$$

**Remarque :** Dans le cas où on a  $f : K \mapsto K$ , la réduction sert également de hachage dans l'implantation. Il apparaît que certaines valeurs de  $|T|$  sont meilleures que d'autres par la même analyse que celle du hachage par modulo décrit précédemment.

On peut programmer facilement cette réduction :

```

1 | // a priori, on passe plutôt en paramètre l'enregistrement de la table
2 | // et on accède à sa capacité par accès au champ
3 | uint32_t reduction(uint32_t capa_T, uint32_t f_k) {
4 |     return f_k % capa_T;
5 | }
```

Le problème de cette fonction est sa lenteur. En effet, la division sur un ordinateur est une opération coûteuse, *a minima* deux fois plus lente qu’une multiplication.

Certaines hypothèses sur les taille de  $|T|$  et de  $f(K)$  permettent d’optimiser le calcul de la réduction.

Si  $|T| = 2^p$ , avec  $p \in \mathbb{N}$ , on peut calculer le modulo par quelques opérations logiques<sup>23</sup> :

```

1  uint32_t reduction(uint32_t f_k, uint32_t p) {
2      return f_k & (~((uint32_t)-1 << p));
3  }
```

Si  $|T|, f(K) \in \llbracket 0; 2^N - 1 \rrbracket$ , on peut accélérer la réduction en calculant  $r(f(K)) = \frac{f(K)|T|}{2^N}$ <sup>24</sup>. Avec  $N = 32$  bits :

```

1  uint32_t reduction(uint32_t f_k, uint32_t card_T) {
2      return ((uint64_t)f_k * (uint64_t)card_T) >> 32;
3  }
```

**Preuve :** En multipliant par  $|T|$ , on fait correspondre les valeurs  $\llbracket 0; 2^{32} \rrbracket$  aux multiples de  $|T|$  dans l’ensemble  $\llbracket 0; |T|2^{32} \rrbracket$ . La division par  $2^{32}$  fait correspondre tous les multiples de  $|T|$  de l’intervalle  $\llbracket k; (k+1)2^{32} \rrbracket$  à  $k$ . Le nombre de multiples de  $|T|$  dans un intervalle de  $2^{32}$  éléments est soit  $\left\lfloor \frac{2^{32}}{|T|} \right\rfloor$  soit  $\left\lceil \frac{2^{32}}{|T|} \right\rceil$ .

**Remarque :** Cette optimisation de la réduction n’est utile que si la fonction de hachage fournit des valeurs dans  $\{0, \dots, 2^{32} - 1\}$  de manière homogène

### 9.5.5 Gestion des collisions

On distingue deux méthodes principalement utilisées de gestion des collisions :

- par collisions séparées
- par adressage ouvert

#### Définition 66 : Taux d’occupation

Si la table de hachage de  $m$  cases contient  $n$  couples (*clef, valeur*), le *taux d’occupation* de la table<sup>a</sup> est le rapport  $\frac{n}{m}$ .

<sup>a</sup>. “Load Factor” en anglais

**Remarques :**

- pour une table par adressage ouvert, on a toujours  $\frac{n}{m} \leq 1$
- pour une table de listes chaînées, on peut avoir  $\frac{n}{m} > 1$  puisqu’il est possible de stocker plusieurs couples par case de la table

23. Si on fixe  $p$  et donc la taille de  $|T|$ , on peut précalculer le masque et n’avoir qu’un seul ET logique.

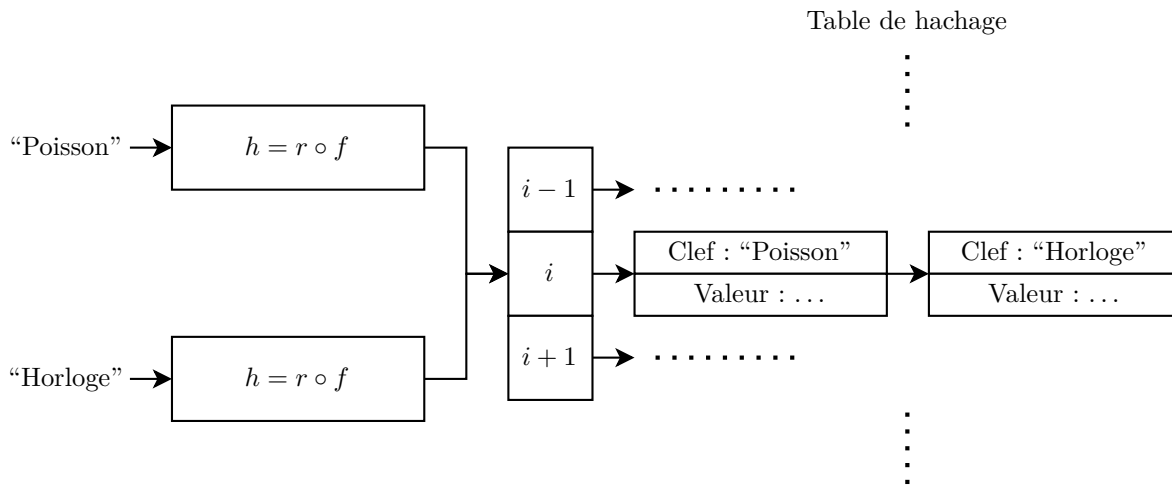
24. Il s’agit d’une optimisation proposée en 2016 par le chercheur Daniel Lemire de l’université du Québec et adoptée entre autre par *Stockfish* en 2017, le meilleur logiciel d’échecs du monde actuellement : <https://lemire.me/blog/2016/06/27/a-fast-alternative-to-the-modulo-reduction/>

## Collisions séparées par listes chaînées

La méthode par de collisions séparées apparaît comme la solution la plus “simple” : stocker toutes les collisions dans un/des espace(s) séparé(s) de la table. Exemples :

- une table de collisions supplémentaire, qui contient toutes les clefs dont la tentative d’ajout a généré une collision
- une table de hachage avec une liste chaînée pour chaque valeur de hachage possible. On a alors une table de listes chaînées

On s’intéresse à la deuxième possibilité : stocker les collisions dans des listes chaînées.



Lors de la recherche de la valeur de “Horloge”, il suffit de parcourir la liste de la cellule calculée par la fonction  $r \circ f$ , et pour chaque élément de comparer la clef avec celle recherchée.

**Remarque :** comme les fonctions de hachage et de réduction sont supposées répartir uniformément les clefs parmi les indices de la table, il n’y a que peu d’éléments en collisions en moyenne. L’implantation d’une liste par tableau dynamique peut être plus rapide car les données sont stockés dans la même localité mémoire et rentrent mieux dans le cache du processeur.

**Redimensionnement :** Si la table est très remplie ( $\alpha \gg 1$ ),

**Analyse du coût d’accès :**

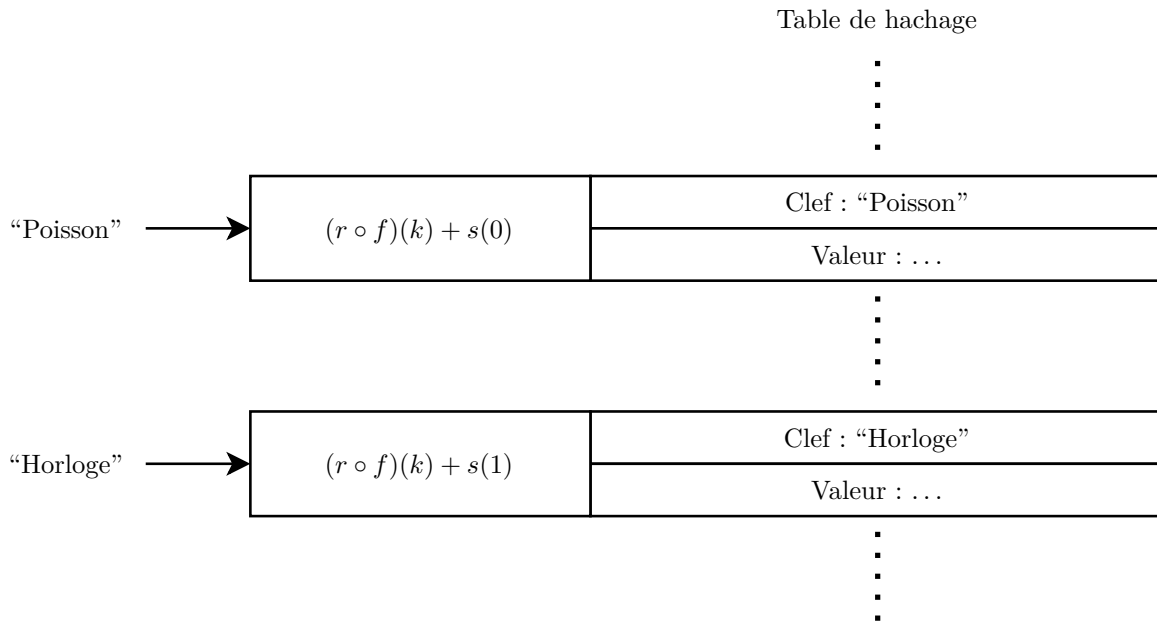
Pour  $0 \leq i < m$ , on note  $N_i$  la variable aléatoire dénotant le nombre d’éléments dans la liste chaînée d’indice  $i$  dans la table. Comme la fonction de hachage est supposée uniforme, on a pour tous  $0 \leq i, j, < m$ ,  $\mathbb{E}(N_i) = \mathbb{E}(N_j)$ . Par ailleurs, on a  $n$  éléments au total donc  $\sum_{i=0}^{m-1} \mathbb{E}(N_i) = n$ , c’est-à-dire que pour tout  $0 \leq i < m$ ,  $\mathbb{E}(N_i) = \frac{n}{m} = \alpha$ .

Donc le coût d’accès en moyenne est égal au taux d’occupation. Si la table est très remplie, et qu’on a  $\alpha \gg 1$ , il ne semble pas y avoir beaucoup d’avantage par rapport à une simple liste que l’on parcourt, sinon d’avoir accéléré par un facteur constant l’accès.

L’idée est de *doubler la taille du tableau* dès que  $\alpha$  s’approche de 1. Cela signifie qu’on copie tout le tableau, mais relativement rarement si le seuil pour  $\alpha$  est assez grand.

## Adressage ouvert par sondage linéaire

La méthode d'adressage ouvert consiste à stocker les paires (*clef*, *valeur*) en collision dans des cases différentes de la table de hachage. Pour calculer les cases vers lesquelles stocker les couples en collisions, on utilise une fonction de sondage  $s : \mathbb{N} \rightarrow \mathbb{N}$ . Le choix de la fonction de sondage est alors important pour garantir l'efficacité de la table. On ne présente ici que le sondage linéaire. D'autres choix sont présentés dans les exercices.



On peut supposer tous les calculs effectués *modulo*  $|T|$ . Pour un sondage linéaire, on a  $s(n) = n$ . On peut en fait choisir n'importe quelle fonction de  $\mathbb{N}$  dans  $\mathbb{Z}$ .

Pour la recherche d'un élément, on suit alors l'algorithme suivant :

---

**Algorithme 17 :** Recherche d'un élément par sondage

---

**Entrées :** *Table* :  $T$   
**Entrées :** *Clef* :  $K$

```

1  $hash \leftarrow r(T, f(K));$ 
2  $n \leftarrow 0;$ 
3  $place \leftarrow hash + s(n);$ 
4 tant que  $(clef(T, place) \neq K) \wedge (n < |T|)$  faire
5   |  $n \leftarrow n + 1;$ 
6   |  $place \leftarrow hash + s(n);$ 
7 si  $clef(T, place) = K$  alors
8   | retourner  $valeur(T, place);$ 
9 sinon
10  | Comportement indéfini... ;                                // La clef n'est pas trouvée
```

---

**Remarque :** le test  $n < |T|$  est optionnel si on suppose l'algorithme définit *ssi* la clef est présente dans le dictionnaire.

Pour l'insertion d'un élément et la suppression d'éléments, on peut utiliser un algorithme très similaire, en se munissant d'une fonction  $est\_utilise(Table : T, Place : p) \rightarrow Booléen$  qui renvoie *Vrai* si la place est utilisée pour un couple  $(clef, valeur)$  et *Faux* sinon. La condition de la boucle devient simplement  $est\_utilise(T, place) \wedge n < |T|$ .

### 9.5.6 Exercices

**Exercice 78 (Hachage de chaînes de caractères) [12].** Proposer une fonction de hachage permettant de hacher des chaînes de caractères.

**Exercice 79 (Dictionnaire en C) [20].** Les clefs et les valeurs considérés ici sont des chaînes de caractères.

1. Proposer un enregistrement *Couple* permettant de stocker un couple  $(clef, valeur)$  d'un dictionnaire
2. Implanter en C les fonctionnalités de dictionnaire proposées à la sous-section 9.5.1 de deux manières :
  - avec une gestion des collisions par chaînage
  - avec une gestion des collisions par adressage ouvert en sondage linéaire

La fonction de hachage et la fonction de réduction doivent pouvoir être spécifier pour chaque instance d'enregistrement de *Dictionary*<sup>25</sup>.

**Exercice 80 (Dictionnaire par ABR) [20].** On considère ici un type abstrait de dictionnaire dont les clefs sont des chaînes de caractères.

1. Donner une relation d'ordre sur les clefs
2. Implanter une structure de dictionnaire par arbre binaire de recherche, où les valeurs des noeuds de l'arbre sont les clefs des couples  $(clef, valeur)$  du dictionnaire.
3. Quelles sont les complexités temporelles de chaque opération de recherche, d'insertion et de suppression ?

**Exercice 81 (Sondage quadratique) [].**

**Exercice 82 (Double hachage) [].**

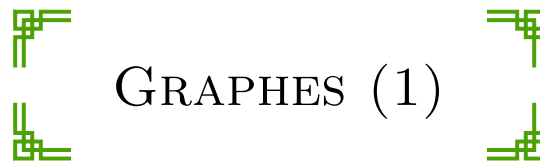
**Exercice 83 (Hachage de Fibonacci) [].**

**Exercice 84 (Filtre de Bloom) [].**

---

25. Les identifiants dans le code sont en anglais, toujours.





*Bis repetita placent.*<sup>26</sup>

– Inspiré d’Horace

La section ci-présente présente le même objet que celui de la section 1.1 mais sous un angle un peu différent. On peut considérer soit que les relations induisent la notion de graphe, soit que les graphes induisent la notion de relation. Cette question qui n’intéresse que les éleveurs de poules ne sera pas discutée ici.

Les graphes constituent un outil fondamental de modélisation d’ensembles structurés par des relations. Ils interviennent chaque fois qu’on étudie un ensemble de liaisons (orientés ou non) entre les éléments d’un ensemble d’objets. On trouve des graphes par exemple :

- dans la modélisation d’un réseau (Internet, relations social, etc...)
- dans la modélisation de problèmes d’ordonnancement de tâches en recherche opérationnelle
- la modélisation de systèmes complexes par la représentation des passages d’un état à un autre
- en théorie des jeux pour la modélisation d’arbres d’évaluation<sup>27</sup>
- en optimisation à la compilation en modélisant par un graphe les dépendances d’expressions d’un programme
- la représentation de modèles de théories
- etc...

Autant dire : un peu partout, puisque partout il y a des relations.

La recherche des algorithmes les plus efficaces pour réaliser les opérations de base sur les graphes constitue donc un objectif essentiel de l’algorithmique.

Les algorithmes décrits dans la suite ne sont pas exigés tels quels aux examens d’*Algo/Prog* de l’ISMIN<sup>28</sup>. Ils sont à connaître pour la culture. Il s’agit du minimum syndical des connaissances algorithmiques sur les graphes. Les connaître permet de se familiariser avec les bases pour avancer avec plus d’aisance à travers des propriétés moins simples.

**Remarque :** Il a été essayé de présenter chaque algorithme avec un maximum de rigueur sans pour autant être *trop* formel. Le but est de donner une intuition bien-fondée du fonctionnement et de la correction des algorithmes. On justifie donc principalement “avec les mains” les algorithmes et l’intuition que l’on peut avoir de leur fonctionnement. On peut trouver un cours bien plus formel et absolument rigoureux en bibliographie [11].

### 9.6.1 Définition et représentations

De nombreuses définitions sont nécessaires pour discuter efficacement des graphes. C’est un gros sujet, qu’on ne traite que pour une infime partie.

26. La répétition est chose plaisante.

27. Un arbre est un graphe particulier.

28. Puisque l’exigence tout court n’est pas encore arrivée à l’ISMIN. Ou alors, ça fait longtemps qu’elle est partie voir ailleurs...

**Définition 67 : Graphe orienté**

On définit un *graphe orienté*  $G$  (ou simplement *graphe*) comme un couple  $(S, A)$  où :

- $S$  est un ensemble fini de *sommets*<sup>a</sup> du graphe.  
On considère sans perte de généralité  $S = \{0, \dots, n-1\}$ , où  $n = |S| \in \mathbb{N}$  le nombre de sommets de  $G$  est appelé son *ordre*.
- $A \subseteq S \times S$  est l'ensemble des *arcs* du graphe, c'est-à-dire des liaisons orientées entre les sommets.

Soient  $u, v \in S$ . On note  $u \rightarrow v$  pour dire qu'il existe un arc de  $u$  vers  $v$  dans  $G$ , c'est-à-dire que  $(u, v) \in A$ . On dit alors que :

- $u$  est un *prédécesseur* de  $v$
- $v$  est un *successeur* de  $u$

<sup>a</sup>. C'est-à-dire en bijection avec un sous-ensemble de  $\mathbb{N}$

**Exemple :** Ci-dessous un dessin d'un graphe  $G = (S, A)$  d'ordre 5 avec :

- $S = \{0, 1, 2, 3, 4\}$
- $A = \{(0, 1), (1, 0), (1, 3), (2, 4), (2, 3), (3, 2), (4, 0), (4, 1), (4, 3)\}$

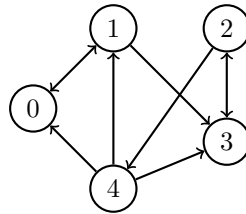


FIGURE 9.11 – Graphe orienté

**Définition 68 : Graphe non orienté**

Un graphe  $G = (S, A)$  est dit *non orienté* si :

$$\forall u, v \in S, (u, v) \in A \Rightarrow (v, u) \in A$$

On peut alors simplifier par  $A \subseteq \{\{u, v\} \mid u, v \in S\}$ . Les éléments  $\{u, v\} \in A$  sont appelés les *arêtes* de  $G$ .

Soient  $u, v \in S$ . On note  $u - v$  pour dire qu'il existe une arête entre  $u$  et  $v$  dans  $G$ , c'est-à-dire que  $\{u, v\} \in A$ . On dit alors que  $u$  et  $v$  sont *adjacents* ou *voisins*.

**Remarque :** on peut toujours voir un graphe non orienté  $G = (S, A)$  comme un graphe orienté  $G = (S, \{(u, v), (v, u) \mid \{u, v\} \in A\})$

**Exemple :** Ci-dessous un dessin d'un graphe  $G = (S, A)$  d'ordre 8 avec :

- $S = \{0, 1, 2, 3, 4, 5, 6, 7\}$
- $A = \{\{0, 4\}, \{0, 1\}, \{2, 3\}, \{3, 5\}, \{3, 6\}, \{5, 7\}, \{6, 7\}\}$

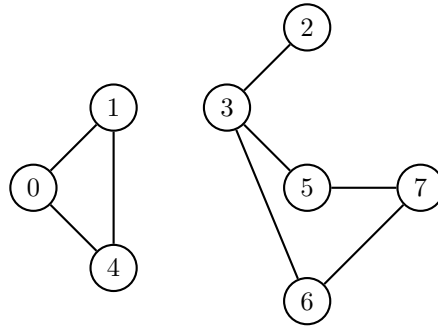


FIGURE 9.12 – Graphe non orienté

**Définition 69 : Degré entrant d'un sommet**

Soit  $G = (S, A)$  un graphe (orienté). Le degré *entrant* d'un sommet  $u \in S$  est :

$$d^-(u) = \text{Card}(\{(v, u) \in A \mid v \in S\})$$

C'est le nombre d'arcs qui arrivent à  $u$ , c'est-à-dire le nombre de prédécesseurs de  $u$  dans  $G$

**Définition 70 : Degré sortant d'un sommet**

Soit  $G = (S, A)$  un graphe (orienté). Le degré *sortant* d'un sommet  $u \in S$  est :

$$d^+(u) = \text{Card}(\{(u, v) \in A \mid v \in S\})$$

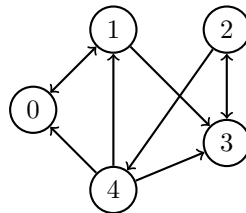
C'est le nombre d'arcs qui partent de  $u$ , c'est-à-dire le nombre de successeurs de  $u$  dans  $G$

**Définition 71 : Degré d'un sommet**

Soit  $G = (S, A)$  un graphe (orienté). Le degré d'un sommet  $u \in S$  est la somme du degré entrant et du degré sortant de  $u$ . On note :

$$d(u) = d^-(u) + d^+(u)$$

**Exemple :** Dans le graphe suivant :



On a  $d^+(3) = 1$  et  $d^-(3) = 3$  donc  $d(3) = d^+(3) + d^-(3) = 4$ .

### 9.6.2 Signature du type abstrait

Avant de présenter les représentations standards en mémoire des graphes, on présente une signature minimale des routines de manipulation des graphes. Ces routines permettront de mettre en évidence quelles sont les opérations fondamentales sur les graphes qui seront utiles pour la quasi-totalité des

algorithmes dans la suite.

*Graphe*<*Sommet*> utilise *Liste*<*Sommet*>, *Sommet*, *Entier*, *Booléen* :

- *creer*(*Graphe*,  $n : \text{Entier}$ )  $\rightarrow \text{Graphe}$  renvoie un graphe d'ordre  $n$  sans arêtes.
- *degre*(*Graphe*,  $s : \text{Sommet}$ )  $\rightarrow \text{Entier}$  renvoie  $d(s)$  le degré de  $s$
- *test\_arc*(( $E, V$ ) : *Graphe*,  $s_1 : \text{Sommet}$ ,  $s_2 : \text{Sommet}$ )  $\rightarrow \text{Booléen}$  renvoie *Vrai* si  $(s_1, s_2) \in E$ , *Faux* sinon
- *succ*(*Graphe*,  $s : \text{Sommet}$ )  $\rightarrow \text{Liste}<\text{Sommet}>$  renvoie la liste des successeurs de  $s$
- *prec*(*Graphe*,  $s : \text{Sommet}$ )  $\rightarrow \text{Liste}<\text{Sommet}>$  renvoie la liste des prédécesseurs de  $s$
- *ajouter\_sommet*( $G : \text{Graphe}$ )  $\rightarrow \text{Graphe}$  renvoie pour  $G = (S, A)$  originellement d'ordre  $n$  le graphe  $G' = (S \cup \{n\}, A)$  d'ordre  $(n + 1)$
- *supprimer\_sommet*( $G : \text{Graphe}$ ,  $s : \text{Sommet}$ )  $\rightarrow \text{Graphe}$  renvoie pour  $G = (S, A)$  originellement d'ordre  $n$  le graphe  $G' = (S \setminus \{s\}, A)$  d'ordre  $(n + 1)$ <sup>29</sup>
- *ajouter\_arc*( $G : \text{Graphe}$ ,  $u : \text{Sommet}$ ,  $v : \text{Sommet}$ )  $\rightarrow \text{Graphe}$  renvoie pour  $G = (S, A)$  le graphe  $G' = (S, A \cup \{(u, v)\})$  résultant de l'ajout de l'arc  $(u, v)$
- *supprimer\_arc*( $G : \text{Graphe}$ ,  $\text{Sommet} : u, v : \text{Sommet}$ )  $\rightarrow \text{Graphe}$  renvoie pour  $G = (S, A)$  le graphe  $G' = (S, A \setminus \{(u, v)\})$  résultant du retrait de l'arc  $(u, v)$

**Remarque 1 :** La signature ci-dessus ne préjuge en rien du type *Sommet*, bien qu'on puisse le plus souvent l'assimiler à *Entier* lors de l'implantation.

**Remarque 2 :** C'est l'efficacité des routines *succ* et *prec* qui garantit l'efficacité du parcours des sommets d'un graphe, et donc de la plupart des manipulations de celui-ci. La routine *succ* est prioritaire sur *prec* dans la plupart des cas d'utilisation.

### 9.6.3 Représentations

On peut représenter d'au moins trois façons standard un graphe en mémoire :

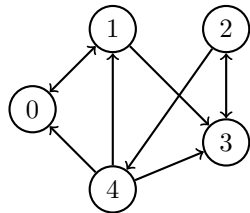
- par matrice d'adjacence
- par liste de successeurs
- (par matrice d'incidence, qu'on ne présentera pas ici de par son aspect particulièrement mineur dans la pratique même si utile parfois dans la théorie)

#### Matrice d'adjacence

On peut représenter un graphe  $G = (S, A)$  comme une matrice  $M \in \mathcal{M}_{|S|}(\mathcal{B})$ . On a alors :

$$\forall u, v \in S, M_{u,v} = \begin{cases} \text{Vrai} & \text{si } (u, v) \in A \\ \text{Faux} & \text{si } (u, v) \notin A \end{cases}$$

**Exemple :** avec  $\text{Vrai} \equiv 1$  et  $\text{Faux} \equiv 0$



$$\begin{pmatrix} 0 & 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 & 1 \\ 0 & 0 & 1 & 0 & 0 \\ 1 & 1 & 0 & 1 & 0 \end{pmatrix}$$

#### Complexité spatiale :

Cette représentation est de complexité spatiale en  $\Theta(|S|^2)$ .

29. On remarque que  $S = \{0, \dots, s-1, s+1, \dots, n\}$  puisque le sommet est quelconque dans  $S$ .

**Complexités temporelles des routines de test et de parcours :**

On peut déduire, pour tout  $u \in S$

- la liste des successeurs de  $u$  par le parcours de la  $u^e$  ligne de  $M$  (en  $\Theta(|S|)$ )
- la liste des prédécesseurs de  $u$  par le parcours de la  $u^e$  colonne de  $M$  (en  $\Theta(|S|)$ )

*idem* pour les calculs de  $d^+(u)$  et  $d^-(u)$

*test\_arc* est de complexité temporelle fonction de  $\Theta(|I|)$  dans le pire des cas puisqu'il suffit de lire la matrice en une case.

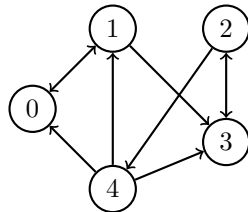
**VS les listes de successeurs :** On va voir avec la représentation par liste de successeurs que le calcul des successeurs est plus rapide mais que le calcul des prédécesseurs est substantiellement plus long (dans le pire cas, en  $O(|S|\log_2(|S|))$  si la liste est triée,  $O(|S|^2)$  sinon). De plus, le test de l'existence d'un arc est linéaire dans le pire des cas pour une liste d'adjacence.

**Liste de successeurs**

Si un graphe  $G = (S, A)$  contient peu d'arêtes, la matrice d'adjacence représentative de  $G$  est creuse. On observe d'abord que la mémoire est gaspillée par le stockage d'une grande quantité de 0 inutiles, d'autre part que le calcul du degré sortant/des successeurs d'un sommet de  $G$  n'est pas optimisé puisqu'il faut prendre en considération beaucoup d'absence d'arêtes, qu'on sait absentes si on sait la matrice creuse.

L'idée de la représentation d'un graphe par listes de successeurs est d'associer à chaque sommet  $u \in S$  la liste de ses successeurs  $\text{succ}(u)$  dans  $G$ . On peut facilement se convaincre que stocker ainsi les successeurs de tous les sommets permet bien de stocker toutes les arêtes, puisque chaque arête part d'un sommet pour *lui en faire succéder* un autre.

**Exemple :**



```

0 : [1]
1 : [0, 3]
2 : [4, 3]
3 : [2]
4 : [0, 3, 1]

```

**Remarque :** les listes de successeurs ne sont pas nécessairement triées. Toutefois, leur tri peut être une propriété appréciable pour l'efficacité de certaines routines de manipulation.

**9.6.4 Propriétés****Définition 72 : Chemin élémentaire**

Soit  $G = (S, A)$ . Soient deux sommets  $u, v \in S$ . Un chemin  $\gamma$  de  $u$  à  $v$  est élémentaire si tous les sommets de  $\gamma$  sont différents. Aucun ne se répète.

**Lemme 1 (König).** Soit  $G = (S, A)$ . Soient deux sommets  $u, v \in S$ . Il existe un chemin de longueur minimal entre  $u$  et  $v$ . Ce chemin est élémentaire.

**Démonstration (un peu avec les mains<sup>30</sup>) :** Un chemin  $\gamma$  de longueur minimal existe nécessairement

30. Mais à un concours de sortie de prépa, ça passe crème.

car le graphe est fini, donc son nombre de chemins entre  $u$  et  $v$  aussi et ces chemins sont ordonnés totalement par la relation “est plus court que”. Si un sommet  $w$  se répète dans  $\gamma$ , alors il existe un sous-chemin  $w \rightarrow \dots \rightarrow w$  dans  $\gamma$ . On peut le supprimer, et la répétition avec.

### 9.6.5 Parcours en profondeur et en largeur

#### Principe et exemples

##### Définition 73 : Parcours

Un parcours est une liste de tous les sommets d'un graphe  $G$   $L = [s_1, \dots, s_n]$  telle que pour tout  $j \in \{2, \dots, n\}$ , il existe  $i < j$  tel que  $s_i \rightarrow s_j$ .

En d'autres termes, lors d'un parcours de graphe, un sommet (autre que le premier) ne peut être visité que s'il est le successeur d'un sommet déjà visité. Sinon, ce n'est pas un parcours, c'est autre chose.

La différence entre un parcours en profondeur et un parcours en largeur tient donc dans le choix de l'ordre de visite.

##### Définition 74 : Parcours partiel

Un parcours partiel est une liste  $L = [s_1, \dots, s_k]$  avec  $k \leq n$  de sommets tous différents de  $G$  telle que pour tout  $j \in \{2, \dots, k\}$ , il existe  $i < j$  tel que  $s_i \rightarrow s_j$ .

En d'autres termes, c'est un parcours pas fini d'un graphe.  $L_n$  est un parcours complet (on a visité tous les  $n$  sommets).

**Remarque :** À tout moment d'un parcours de graphe, certains sommets sont visités et d'autres ne le sont pas. Si tous les sommets sont visités, c'est que le parcours est fini. En particulier, cela signifie que tant que le parcours n'est pas fini, certains sommets ont des successeurs non visités.

##### Définition 75 : Sommet ouvert

Un sommet d'un parcours partiel  $L_k$  est *ouvert* s'il a au moins un successeur qui n'apparaît pas dans  $L_k$ , c'est-à-dire qui n'est pas encore visité.

##### Définition 76 : Sommet fermé

Un sommet d'un parcours partiel  $L_k$  est *fermé* si tout ses successeurs sont dans  $L_k$ , c'est-à-dire qui sont tous déjà visités.

##### Définition 77 : Sommet découvert

On appelle *sommet découvert* par un parcours  $L_k$  un sommet non visité (= pas dans  $L_k$ ) qui est successeur d'un sommet visité (= dans  $L_k$ ).

On observe deux choses :

- le dernier sommet ouvert d'un parcours partiel est le plus récent sommet visité qui ait des successeurs non visités.
- le premier sommet ouvert d'un parcours partiel est le plus ancien sommet visité qui ait des successeurs non visités.

D'où deux choix qui apparaissent clairement :

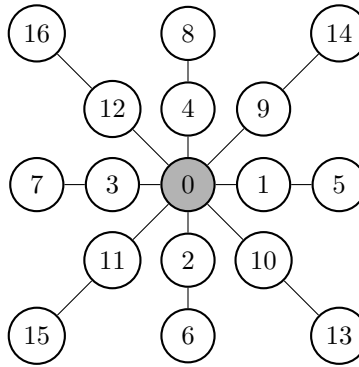
1. à chaque visite, choisir un successeur du plus récent sommet visité ouvert  $\Rightarrow$  parcours en profondeur
2. à chaque visite, choisir un successeur du plus ancien sommet visité ouvert  $\Rightarrow$  parcours en largeur

La définition de chacun des deux types de parcours est ainsi donné, en itérant le choix ci-dessus jusqu'à avoir parcouru tout le graphe. Maintenant, il s'agit de construire une intuition, c'est-à-dire de comprendre pourquoi ces parcours sont appelés *en profondeur* et *en largeur*. On s'occupera après de clarifier un algorithme pour chaque cas.

Dans les deux exemples qui suivent, on colorie :

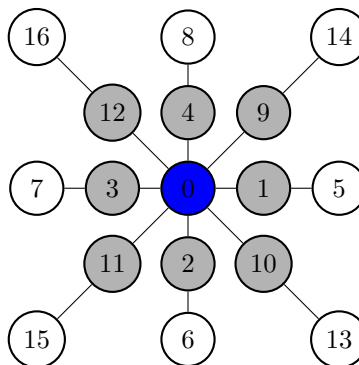
- en vert les sommets visités
- en gris les sommets découverts
- en bleu le sommet visité ouvert :
  - ◆ le plus récent dans le cas du parcours en profondeur
  - ◆ le plus ancien dans le cas du parcours en largeur

Dans les deux cas, on parcourt le graphe non orienté<sup>31</sup> suivant :



Le premier sommet est donc à chaque fois 0 puisqu'il s'agit du premier qu'on *marque* comme découvert initialement et qui est donc le seul à pouvoir être visité.

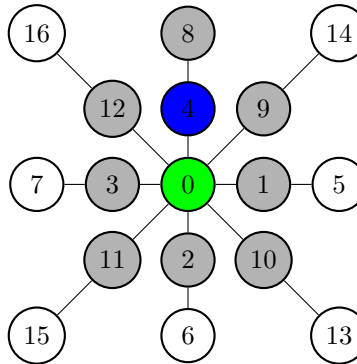
**Exemple 1 (parcours en profondeur) :** On visite 0. Donc  $L_1 = [0]$  Les sommets découverts du parcours partiel  $L_0$  sont les successeurs non visités des sommets de  $L_0$ , donc ici :



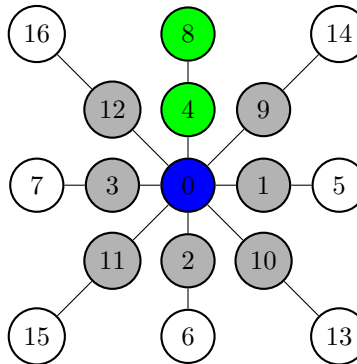
On visite ensuite un successeur du plus récent sommet visité ouvert, *i.e.* qui a encore des successeurs

31. Pour simplifier. Un graphe orienté, y a plein de cas bizarres de connexité, c'est plus chiant à lire.

non visités. Comme  $L_1$  ne contient que 0, il suffit de choisir arbitrairement un successeur de 0<sup>32</sup> comme 4 par exemple<sup>33</sup> :



On a alors le parcours partiel  $L_2 = [0, 4]$ . Comme 4 est ouvert, c'est le sommet ouvert le plus récent de  $L_2$ . On choisit un de ses successeurs *non visité* (donc pas 0), c'est-à-dire nécessairement 8 :



On a alors le parcours partiel  $L_3 = [0, 4, 8]$ . Comme 4 et 8 sont fermés, 0 redevient le sommet ouvert visité le plus récemment. On visite alors un des successeurs non visité de 0, donc un de ses successeurs différent de 4. L'exécution est alors similaire à la visite de 4 et 8. On itère jusqu'à avoir parcouru tous les sommets du graphe.

**Conclusion sur le parcours en profondeur :** le parcours va aller le plus loin possible puisqu'il choisi toujours les successeurs du sommet ouvert le plus récent. Il va *le plus profond possible*<sup>34</sup> jusqu'à ne plus pouvoir aller plus loin. Il revient alors à l'ex. plus ancien sommet ouvert par la fermeture des sommets plus récents.

Un parcours final peut alors être  $L = [0, 4, 8, 9, 14, 1, 5, 10, 13, 3, 7, 12, 16, 11, 15, 2, 6]$

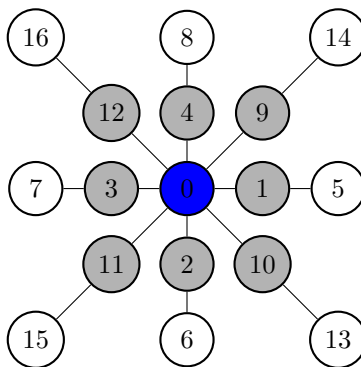
**Exemple 2 (parcours en largeur) :** on visite 0. Donc  $L_1 = [0]$  Les sommets découverts du parcours partiel  $L_0$  sont les successeurs non visités des sommets de  $L_0$ , donc ici :

32. On pourra choisir par exemple le premier découvert. Sachant que l'ordre de découverte est arbitraire (dépend de l'implantation), c'est pareil.

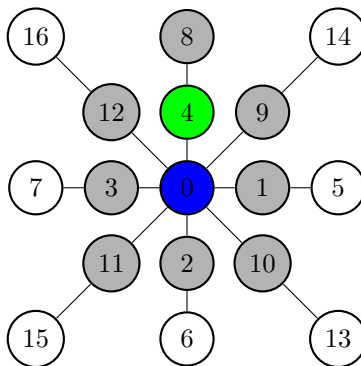
33. *Why not.*

34. ??? WTF

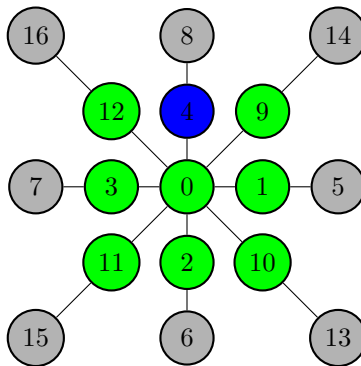




On visite ensuite un successeur du plus ancien sommet visité ouvert, *i.e.* qui a encore des successeurs non visités. Comme  $L_1$  ne contient que 0, il suffit de choisir arbitrairement un successeur de 0<sup>35</sup> comme 4 par exemple<sup>36</sup> :



On a alors le parcours partiel  $L_2 = [0, 4]$ . Ici, 0 est toujours ouvert et le plus ancien (donc toujours en bleu). C'est donc lui dont il faut continuer à visiter les successeurs. Cela va continuer jusqu'à avoir visité tous les successeurs de 0. On va arriver à la situation suivante :



On a un parcours partiel possible  $L_9 = [0, 4, 9, 1, 10, 2, 11, 3, 12]$ . Cette fois, 0 est fermé. On va donc choisir le sommet ouvert le plus ancien, qui est 4. Son unique successeur non visité est 8, qui sera donc le prochain sommet visité. Ce processus recommence jusqu'à visiter tous les sommets du graphe.

**Conclusion sur le parcours en largeur :** le parcours va visiter les sommets dans l'ordre de

35. Voir la note de l'exemple de parcours en profondeur.

36. *Why not (again).*

découverte. À l'inverse du parcours en profondeur, le parcours en largeur traite tous les voisins<sup>37</sup> d'un sommet avant de passer à chacun de ses voisins et de faire de même. On appelle aussi le parcours en largeur le parcours en “pelure d'oignon” car on parcourt les sommets par couche, comme les pelures d'un oignon.

## Algorithmes

L'entrée de chacun des algorithmes de parcours est :

- Un graphe  $G = (S, A)$
- Un sommet de départ  $s_0$  qui est le premier sommet du parcours  $L$

La sortie est simplement la liste de parcours.

La question la plus importante qui ressort pour chacun des algorithmes est la manière de stocker les sommets découverts et de choisir quel sommet découvert visiter. On suit les intuitions<sup>38</sup> suivantes :

- pour un parcours profondeur, le sommet choisi est celui découvert le plus récemment puisqu'il est successeur de l'ouvert le plus récent.
- pour un parcours largeur, le sommet choisi est celui découvert il y a le plus longtemps, puisqu'il est successeur de l'ouvert le plus ancien.

Il s'agit respectivement d'une structure *FIFO* (*First In First Out*) caractéristique du type abstrait *Pile* et d'une structure *LIFO* (*Last In First Out*) caractéristique du type abstrait *File*.

En stockant les sommets découverts dans un objet d'un de ces types, avec une même structure d'algorithme on obtiendra soit un parcours en profondeur (avec une *Pile*) soit un parcours en largeur (avec une *File*).

---

### Algorithme 18 : Parcours en profondeur

---

**Entrées :** *Graphe* :  $G = (S, A)$ , *Sommet* :  $s_0 \in S$

```

1 Tableau<Booleen> : sommets_visites [| $S$ |];
2 pour tous  $i \in S$  faire
3   | sommets_visites[ $i$ ] = Faux;
4 Liste<Sommet> : parcours  $\leftarrow$  creer_liste_vide();
5 Pile<Sommet> : decouverts  $\leftarrow$  creer_pile_vide();
6 decouverts  $\leftarrow$  empiler(decouverts,  $s_0$ );
7 tant que  $\neg$ est_vide(decouverts) faire
8   | Sommet :  $u \leftarrow$  depiler(decouverts);
9   | si sommets_visites[ $u$ ] = Faux alors
10    |   | sommets_visites[ $u$ ] = Vrai;
11    |   | parcours  $\leftarrow$  insérer_en_queue(parcours,  $u$ );
12    |   | pour tous  $v \in \text{succ}(G, u)$  faire
13    |   |   | decouverts  $\leftarrow$  empiler(decouverts,  $v$ );
14 retourner parcours;
```

---

Il s'agit simplement d'empiler/enfiler d'abord le sommet de départ  $s_0$  puis boucler le défilement/dépilement du sommet découvert choisi (selon la structure) et de ce sommet maintenant visité, d'empiler/enfiler ses successeurs (qui sont à présent découverts). Si un sommet a déjà été visité, on l'ignore simplement.

---

37. D'un coup *large*, disons...

38. Démonstrables formellement.

**Algorithme 19** : Parcours en largeur

---

**Entrées** : *Graphe* :  $G = (S, A)$ , *Sommet* :  $s_0 \in S$

```

1  Tableau<Booleen> : sommets_visites [|S|];
2  pour tous  $i \in S$  faire
3     $\lfloor$  sommets_visites[ $i$ ] = Faux;
4  Liste<Sommet> : parcours  $\leftarrow$  creer_liste_vide();
5  File<Sommet> : decouverts  $\leftarrow$  creer_file_vide();
6  decouverts  $\leftarrow$  enfiler(decouverts,  $s_0$ );
7  tant que  $\neg$ est_vide(decouverts) faire
8    Sommet :  $u \leftarrow$  defiler(decouverts);
9    si sommets_visites[ $u$ ] = Faux alors
10      $\lfloor$  sommets_visites[ $u$ ] = Vrai;
11     parcours  $\leftarrow$  insérer_en_queue(parcours,  $u$ );
12     pour tous  $v \in \text{succ}(G, u)$  faire
13        $\lfloor$  decouverts  $\leftarrow$  enfiler(decouverts,  $v$ );
14 retourner parcours;

```

---

### 9.6.6 Algorithme de Roy-Warshall

L'algorithme de Roy-Warshall<sup>39</sup> permet de calculer la matrice d'accessibilité d'un graphe, c'est-à-dire la matrice

$$M \in \mathcal{M}_{|S|}(\mathcal{B})$$

telle que :

$$\forall u, v \in S, M_{u,v} = \begin{cases} \text{Vrai} & \text{si } u \rightarrow^* v \quad (\text{il existe un chemin de } u \text{ à } v) \\ \text{Faux} & \text{si } u \not\rightarrow^* v \quad (\text{il n'existe pas de chemin de } u \text{ à } v) \end{cases}$$

C'est la matrice dénotant la clôture transitive<sup>40</sup>  $\rightarrow^*$ .

Cet algorithme est assez classique mais est en fait bien moins rapide *pour la simple détermination de la matrice d'accessibilité* que d'autres algorithmes comme l'algorithme de Tarjan<sup>41</sup>. Toutefois, l'algorithme de Roy-Warshall peut être facilement généralisé aux graphes valués sans perte d'efficacité. Il reste donc incontournable.

La complexité temporelle de l'algorithme est visiblement une fonction de  $\Theta(n^3)$ .

L'algorithme mérite quelques explications, pour justifier qu'il soit correct et ne soit pas simplement le résultat étrange d'une quelconque magie.

**Notation dans cette section** : Soit  $G = (S, A)$  un graphe avec  $S = \{1, \dots, n\}$ . Pour tout  $k \in S$ , on note  $G_k$  le graphe  $(S, A_k)$  où  $A_k$  est l'ensemble des couples  $(u, v) \in S^2$  telles qu'il existe  $u = x_0 \rightarrow x_1 \rightarrow \dots \rightarrow x_{l-1} \rightarrow x_l = v$ , où  $l \in \mathbb{N}$  et pour tout  $0 \leq i \leq l$ ,  $x_i \in \{1, \dots, k\}$ . L'intérieur du chemin est alors  $\{x_1, \dots, x_{l-1}\}$  si  $l \geq 2$ , et  $\emptyset$  si  $l = 0$ .

39. Comme pointé par Jean Goubault-Larrecq [11], *Roy* se prononce *roi*. Bernard Roy est français.

40. Réflexive si le graphe est non orienté puisque  $a \rightarrow a \Leftrightarrow a \leftarrow a$

41. L'algorithme de Tarjan sert à déterminer les composantes fortement connexes d'un graphe. Il est de même classe de complexité que l'algorithme de Kosaraju, mais ses constantes sont bien plus faibles. Les composantes fortement connexes d'un graphe induisent facilement sa matrice d'accessibilité. On ne présente par ici l'algorithme de Tarjan car il nécessite l'introduction de la notion de *points d'attache*, qui est technique et relativement difficile à expliquer clairement sans passer par 10 pages de démonstrations. Peut-être que ce sera ajouté, mais quand la motivation et une idée d'explication claire seront là.

**Algorithme 20** : Algorithme de Roy-Warshall

---

**Entrées :** *Matrice* :  $M_G$ , *Entier* :  $n$

```

1 Matrice :  $A \leftarrow \text{copie}(M_G)$ ;
2 pour tous  $i \in \{1, \dots, n\}$  faire  $A[i][i] = 1$  ;
3 pour tous  $k \in [1, \dots, n]$  faire
4   pour tous  $i \in [1, \dots, n]$  faire
5     pour tous  $j \in [1, \dots, n]$  faire
6        $A[i][j] \leftarrow A[i][j] \vee (A[i][k] \wedge A[k][j])$ ;
7 retourner  $A$ ;
```

---

Autrement dit, c'est le graphe  $G$  auquel on ajoute des arcs qui représentent l'existence d'un chemin fait de sommets de  $\{1, \dots, k\}$ . La matrice de  $G_n$  donne l'existence des chemins faits de sommets de  $S$ . C'est donc la matrice d'accessibilité de  $G$ .

On veut justifier pourquoi l'algorithme de Roy-Warshall calcule effectivement les matrices de  $G_1$  à  $G_n$ .

**Proposition 17 (Construction par récurrence).** Soit  $G = (S, A)$ . Pour tout  $(u, v) \in S^2$ , on a :

1.  $(u, v) \in A_0 \Leftrightarrow u = v \vee (u, v) \in A$
2.  $\forall k \in \{1, \dots, n\}, (u, v) \in A_k \Leftrightarrow (u, v) \in A_{k-1} \vee ((u, k) \in A_{k-1} \wedge (k, v) \in A_{k-1})$

**Signification de la récurrence :** Le deuxième point signifie littéralement : si deux sommets de  $G$  sont reliés par un chemin constitué de sommets (qu'on peut supposer tous différents si il n'y a pas de boucle) dans  $\{1, \dots, k\}$ , c'est que :

- si le sommet  $k$  n'est pas dans le chemin, ils sont reliés par un chemin de constitué de sommets dans  $\{1, \dots, k-1\}$
- si le sommet  $k$  est dans le chemin, on peut le décomposer en deux moitiés de chemin pour lesquelles  $k$  ne sera qu'à une extrémité mais pas au milieu

**Démonstration :**

1. Voir la définition
2. La réciproque est immédiate car  $A_{k-1} \subset A_k$ . Si  $(u, v) \in A_k$ , il existe donc un chemin d'intérieur inclu dans  $\{1, \dots, k\}$ . D'après le lemme de König, il existe un chemin minimal (donc élémentaire)  $\gamma$  entre  $u$  et  $v$ .

On procède ensuite par disjonction de cas selon que  $k$  est ou non dans  $\gamma$  :

- Supposons donc que  $k$  soit un sommet de  $\gamma$ . Le sous-chemin de  $u$  à  $k$  dans  $\gamma$  est, comme  $\gamma$ , à sommets dans  $\{1, \dots, k\}$ . Or  $\gamma$  est élémentaire, donc les sommets du sous-chemin sont dans  $\{1, \dots, k-1\}$ . Donc  $(u, k) \in A_{k-1}$ . Par le même raisonnement, on a  $(k, v) \in A_{k-1}$
- Si  $k$  n'est pas un sommet de  $\gamma$ , alors on a  $(u, v) \in A_{k-1}$

L'algorithme de Roy-Warshall trouve là un début de justification. Les lignes 1 et 2 calculent  $A_0$  et la triple boucle imbriquée applique la récurrence de la proposition précédente.

Un point chiffonne encore : dans la récurrence de la proposition ci-dessus,  $A_k$  est stocké indépendamment de  $A_{k-1}$ . Dans l'algorithme 20, il n'y a pas de copie de  $G_k$  (qui ralentirait considérablement l'algorithme). Tout est fait dans le même tableau, et certaines données pourraient donc s'entrecouper lors des calculs et fausser le résultat. Il faut s'assurer que cela n'arrive pas.

**Lemme 2.** On va avoir besoin pour démontrer la proposition suivante de deux points :

Pour tous  $i, j, k \in \{1, \dots, n\}$ ,

1.  $(i, k) \in A_{k-1} \Leftrightarrow (i, k) \in A_k$

2.  $(k, j) \in A_{k-1} \Leftrightarrow (k, j) \in A_k$

**Démonstration :**  $\Rightarrow$  est direct car  $A_{k-1} \subset A_k$ . Si  $(i, k) \in A_k$  (resp.  $(k, j)$ ), il existe un chemin minimal donc élémentaire (lemme de König) entre  $i$  et  $k$  (resp.  $j$  et  $k$ ) d'intérieur inclu dans  $\{1, \dots, k\}$  et par élémentarité,  $k$  n'apparaît pas, donc intérieur inclu dans  $\{1, \dots, k-1\}$ . D'où  $(i, k) \in A_{k-1}$  (resp.  $(k, j) \in A_{k-1}$ ).

**Proposition 18 (Correction de l'algorithme de Roy-Warshall).** L'algorithme de Roy-Warshall est correct, c'est-à-dire que  $A_0$  est correct et les invariants suivants sont vrais pour tout  $k \in \{1, \dots, n\}$  : Pour tous  $i, j \in \{1, \dots, n\}$

- pour tout  $(u, v) \in \{1, \dots, n\}^2$ ,  $(u, v) \leq_{lex} (i, j) \Rightarrow A[u][v] = (u, v) \in A_k$
- pour tout  $(u, v) \in \{1, \dots, n\}^2$ ,  $(u, v) >_{lex} (i, j) \Rightarrow A[u][v] = (u, v) \in A_{k-1}$

**Démonstration :** on procède par récurrence

Initialisation : pour  $k = 0$ , le programme est correct par le point 1 de la **Proposition 17**

Hérédité : soit  $k \in \{1, \dots, n\}$ , soient  $(i, j), (u, v) \in \{1, \dots, n\}^2$ . Supposons les invariants vrais.

- Si  $(u, v) \leq_{lex} (i, j)$ , par HR,  $A[u][v] = (u, v) \in A_{k-1}$ .  
D'après le lemme précédent,  $A[u][k]$  et  $A[k][v]$  sont également correctes car les couples  $(u, k)$  et  $(k, v)$  appartiennent soit à  $A_{k-1}$  soit à  $A_k$  (la valeur dans la matrice est identique par équivalence des deux appartenance).  
D'où, d'après la proposition ci-dessus,  $A[u][v] \vee (A[u][k] \wedge A[k][v]) = (u, v) \in A_k$
- Si  $(u, v) >_{lex} (i, j)$ , le raisonnement est le même.

Conclusion : L'hypothèse de récurrence est initialisée et héréditaire. Donc le programme calcule bien les  $G_k$ . À la fin de l'exécution,  $G_n$  est le graphe d'accessibilité de  $G$ .

## 9.6.7 Exercices

**Exercice 85 (Limite de la représentation matricielle) [3].** Supposons un cas d'application réel des graphes, comme par exemple la représentation des connexions entre les différents serveurs d'Internet, ou le réseau routier en France, qui relie les différents points d'intérêts. Dans chaque cas, on a *au moins* un ordre de grandeur de  $10^6$  serveurs/points d'intérêts. Que dire de la réalisabilité *pratique* d'une représentation par matrice d'adjacence d'un tel graphe ?

**Exercice 86 (Représentation et implantation) [28].**

1. Planter en C les enregistrements `struct MatrixGraph {...};` et `struct ListGraph {...};` contenant :
  - l'ordre du graphe
  - la représentation du graphe, soit par matrice d'adjacence (tableau de tableaux), soit par listes d'adjacence (tableau de listes)
2. Planter la signature donnée à la sous-section 9.6.2<sup>42</sup>
3. Justifier la complexité temporelle dans le pire cas de chaque routine de manipulation
4. Planter deux fonctions `struct MatrixGraph listgraph_to_matrixgraph(struct ListGraph g);` et `struct ListGraph matrixgraph_to_listgraph(struct MatrixGraph g);` qui effectuent une copie d'un graphe en modifiant la représentation utilisée. Quelle est la complexité temporelle de chaque fonction dans le pire cas ?

**Exercice 87 (Pour le plaisir de l'optimalité) [25].**

1. Proposer une représentation des graphes par tables de hachage d'adjacence telle que :

42. Les identifiants dans le code sont en anglais, toujours.

- le test de l'existence d'un arc
- le calcul de la liste des successeurs
- la suppression et l'ajout d'arcs

soient de complexité temporelle fonction de  $\Theta(|1|)$  en moyenne.

2. Implanter les graphes en C en utilisant cette représentation

**Exercice 88 (Aiguilles hors d'oignons sans profondeur) [6].** Finir l'exécution des algorithmes de parcours en profondeur et en largeur sur le graphe d'exemple donné.

**Exercice 89 (Les parcours) [23].**

1. Quelle est la complexité temporelle des deux parcours dans le pire cas si on choisit de représenter le graphe par matrice d'adjacence ?
2. Quelle est la complexité temporelle des deux parcours dans le pire cas si on choisit de représenter le graphe par listes d'adjacence ?
3. Implanter les parcours en profondeur et en largeur avec une représentation du graphe permettant une complexité temporelle optimale.
4. Soit  $G = (S, A)$  un graphe orienté. Démontrer que si un parcours est initié sur un sommet  $s \in S$ , alors les parcours en profondeur et largeur déterminent exactement les sommets accessibles depuis  $s$ , c'est-à-dire tous les sommets  $u \in S$  tels que  $s \xrightarrow{G} u$

**Exercice 90 (Bicoloriage) [15].** On dit qu'un graphe  $G = (S, A)$  non orienté est *bicoloriable* si il est possible de colorier par une fonction  $C : S \rightarrow \mathcal{B}$  chacun de ses sommets avec deux couleurs 0 et 1 de sorte que pour chaque arête  $\{a, b\} \in A$ ,  $C(a) \neq C(b)$ .

1. Justifier que  $G$  est bicoloriable *ssi* chacune de ses composantes connexes est bicoloriable
2. Montrer qu'un graphe  $G$  connexe non orienté est bicoloriable *ssi* pour tout sommet  $u$  visité lors d'un parcours, pour tout  $v \in \text{succ}(u)$ ,  $C(u) \neq C(v)$
3. En déduire une fonction `bool graph_is_2coloriable(Graph g);`  
*Remarque : On n'impose pas la représentation du graphe, qui est au choix du lecteur, mais influe sur la réponse à la question suivante.*
4. Quelle est sa classe de complexité temporelle ?

**Exercice 91 (Plus court chemin) [20].** Soit  $G = (S, A)$  un graphe orienté. Soit  $s \in S$  fixé.

1. Montrer que le parcours en largeur sur  $G$  depuis  $s$  détermine pour chaque sommet  $u \in S$  la longueur en nombre d'arcs du plus court chemin entre  $s$  et  $u$ . On note cette longueur  $d(s, u)$
2. Montrer également que pour tout  $d \in \mathbb{N}$ , les sommets de distance  $d(s, u) \leq d$  sont tous visités avant les sommets de distance  $d(s, u) > d$
3. En déduire et implanter en C un algorithme de parcours en largeur qui n'utilise pas de file mais alterne deux ensembles<sup>43</sup> de sommets  $E_d$  et  $E_{d+1}$  (représentant les sommets à distance  $d$  et  $d + 1$ ) en incrémentant  $d$  à chaque tour de boucle, c'est-à-dire lorsque tous les sommets de  $E_d$  sont visités.

**Exercice 92 (Arbres et graphes) [24].** On s'intéresse dans cet exercice à la relation entre les arbres et les graphes.

### Définition 78 : Arbre non orienté

En théorie des graphes, un *arbre non orienté* est un graphe non orienté connexe acyclique.

1. Justifier que cette définition contient celle donnée à la section 9.3.
2. Montrer que les propriétés suivantes sont toutes équivalentes pour  $G = (S, A)$  un graphe non orienté :

43. Implantés avec une structure quelconque.

- (a)  $G$  est un arbre non orienté
- (b)  $G$  est connexe et a  $(n - 1)$  arêtes
- (c)  $G$  est acyclique et a  $(n - 1)$  arêtes
- (d) Pour tous  $x, y \in S$ , il existe une unique chaîne entre  $x$  et  $y$
- (e)  $G$  est connexe minimal, c'est-à-dire que  $G$  est connexe et la suppression d'une arête de  $G$  détruit la connexité.
- (f)  $G$  est acyclique maximal, c'est-à-dire que  $G$  est acyclique et ajouter une arête à  $G$  crée un cycle.

On en déduit la définition :

**Définition 79 : Arbre couvrant**

Soit  $G = (S, A)$  un graphe connexe *non orienté*. Un *arbre couvrant* de  $G$  est un sous-graphe de  $G$  connexe minimal dont les sommets sont ceux de  $G$ .

3. Quel algorithme du cours permet de calculer un arbre couvrant de  $G$  ?



## PARTITIONS D'ENSEMBLES FINIS

On considère un ensemble fini  $E$ . Comme  $E$  est fini, il est en bijection avec  $\{0, \dots, n-1\}$  où  $n = |E|$ . On cherche ici un moyen efficace de stocker et de manipuler une partition de  $E$ , c'est-à-dire une partition de  $\{0, \dots, n-1\}$ .

La manipulation de partitions de  $E$  correspond exactement à la manipulation de classes d'équivalence de  $E$ . Les classes d'équivalences sont la conséquence d'une forme de clôture de relation entre objets (clôture réflexive-transitive-symétrique de la relation). Le lien direct avec les relations justifie théoriquement la manipulation des partitions d'ensembles par la manipulation de relations. L'efficacité remarquable des algorithmes qui en découlent en est une justification pratique.

### 9.7.1 Signature

L'objectif est principalement de travailler sur les classes d'équivalences d'une relation sur un ensemble, c'est-à-dire :

- construire des classes d'équivalence d'un ensemble par unions itérés de classes au départ unitaires (ne contenant qu'un seul élément)
- trouver la classe d'équivalence d'un élément quelconque de l'ensemble (*i.e.* son représentant)
- déterminer efficacement le nombre de classes d'équivalences dans l'ensemble par la relation

On en déduit la signature du type abstrait *Partition* :

*Partition* utilise *Booléen*, *Entier*

- *creer*( $n : \text{Entier}$ )  $\rightarrow \text{Partition}$  renvoie une partition d'un ensemble à  $n$  éléments en  $n$  classes d'équivalences où chaque élément de l'ensemble est représentant de sa propre classe d'équivalence d'un seul élément.
- *unir*(*Partition*,  $a : \text{Entier}$ ,  $b : \text{Entier}$ )  $\rightarrow \text{Partition}$  renvoie la partition de  $E$  où les classes d'équivalence de  $a$  et  $b$  sont unies en une seule classe d'équivalence
- *trouver*(*Partition*,  $i : \text{Entier}$ )  $\rightarrow \text{Entier}$  renvoie  $i$  le représentant de la classe d'équivalence de  $i$
- *nombre\_classes*(*Partition*)  $\rightarrow \text{Entier}$  renvoie le nombre de classes d'équivalences de la partition.

### 9.7.2 Principe sur un exemple naïf

Prenons l'ensemble  $E = \{0, 1, 2, 3, 4, 5\}$ . Sa partition la plus fine est constituée de 6 classes  $P = \{\{0\}, \dots, \{5\}\}$ . Si  $\rightarrow$  est la relation (homogène sur  $E$ ) associée à  $P$ , on a :

$$\rightarrow = \{(0, 0), (1, 1), (2, 2), (3, 3), (4, 4), (5, 5)\}$$

représentée graphiquement par :

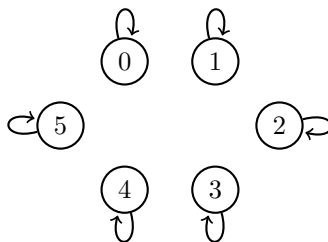




FIGURE 9.13 – Graphe initial de la relation

On peut faire subir quelques transformations à  $P$  selon les routines de manipulation décrites par la signature de *Partition* :

- $P \leftarrow \text{unir}(P, 0, 4) = \{\{0, 4\}, \{1\}, \{2\}, \{3\}, \{5\}\}$
- $P \leftarrow \text{unir}(P, 5, 1) = \{\{0, 4\}, \{1, 5\}, \{2\}, \{3\}\}$
- $\text{trouver}(P, 5) \rightarrow 1$
- $P \leftarrow \text{unir}(P, 5, 4) = \text{unir}(P, 1, 4) = \{\{0, 4, 1, 5\}, \{2\}, \{3\}\}$
- $\text{trouver}(P, 1) \rightarrow 4$

À la fin, on a  $0 \equiv 1 \equiv 4 \equiv 5$  dans  $P$ .

**Lien avec les relations et représentation graphique :** l'évolution de la partition peut être représentée par une relation  $\rightarrow$  telle que pour toute classe  $C$  de  $E$ ,  $e, f \in C \Leftrightarrow e \rightarrow^* f$ .

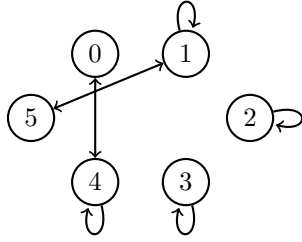


FIGURE 9.14 – Graphe après deux unions

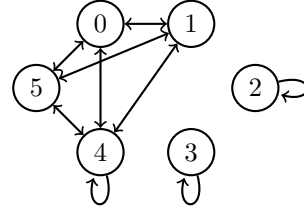


FIGURE 9.15 – Graphe final après une union

La partition représentée est simplement la clôture réflexive-transitive-symétrique du graphe final (après avoir subi toutes les unions).

**Problème de cette représentation naïve :** en deux points :

- Il est malheureusement très coûteux calculatoirement de *maintenir* une relation d'équivalence lors de la modification de celle-ci. Par exemple, l'union de deux classes d'équivalences requiert de mettre en relation chacun des éléments de la première classe avec chacun des éléments de la seconde. Ainsi, l'union de deux classes d'équivalences  $E_1$  et  $E_2$  d'un ensemble  $E$  nécessite  $\text{Card}(E_1) \times \text{Card}(E_2)$  opérations.
- Par ailleurs, les classes d'équivalences de la relation font que chaque élément d'une partie de  $E$  est représentant de cette partie. On ne peut pas utiliser ce critère de représentation pour vérifier que deux éléments  $e_1$  et  $e_2$  appartiennent à la même classe. Il faut à la place trouver ou non un chemin entre  $e_1$  et  $e_2$  dans le graphe, ce qui est également coûteux.

Finalement, cette représentation naïve est extrêmement coûteuse calculatoirement et ne permet pas d'implanter efficacement les deux opérations *unir* et *trouver*.

### 9.7.3 La clef vers l'optimisation

On observe que tous éléments d'une classe d'équivalence  $C$  représentent cette classe. Si on choisit en revanche un *unique* élément  $r_C \in C$  pour représenter  $C$ , il faut simplement pour trouver la classe d'un élément  $e \in E$  lui associer son représentant par un chemin d'arcs. Cette association fournit une nouvelle relation, noté  $\rightarrow_r$  telle que :

$$\forall C \in (E / \rightarrow), \forall e \in E, (e \in C \Leftrightarrow \exists ! e_1, \dots, e_{n-1} \in C \setminus \{r_C\}, e \rightarrow_r e_1 \rightarrow_r \dots \rightarrow_r e_{n-1} \rightarrow_r r_C)$$

**En français :** un élément  $e$  appartient à une classe d'équivalence défini par  $\rightarrow$  si et seulement si il existe un unique chemin de  $e$  vers  $r_C$  défini par  $\rightarrow_r$ .  $\rightarrow_r$  restreint donc  $\rightarrow$  en ne conservant qu'un seul chemin de tout élément d'une classe vers son représentant. C'est-à-dire que  $\rightarrow_r$  définit un arbre sur chaque classe  $C$  dont la racine est  $r_C$  (voir l' **Exercice 92** ).

**Remarques :**

- cette définition de  $\rightarrow_r$  n'assure absolument pas son unicité
- $\rightarrow$  est exactement la clôture réflexive-transitive-symétrique de  $\rightarrow_r$ , ce qui montre que  $\rightarrow_r$  désigne le même quotient (après clôture de la relation)
- la relation  $\rightarrow_r$  est fonctionnelle : chaque élément ne peut se déplacer que vers un unique autre élément dans le graphe (dans le cas contraire, le chemin vers  $r_C$  n'est pas unique). Autrement dit, pour tout  $s \in E$ ,  $d_{\rightarrow_r}^+(s) = 1$  ( $d_{\rightarrow_r}^+$  est le degré *sortant* de  $s$  dans le graphe représentatif de la relation  $\rightarrow_r$ ).

### Caractérisation relationnelle des opérations “unir” et “trouver”

De la définition et des remarques précédentes on déduit qu'étant donné un élément  $e \in E$ , on peut trouver sa classe d'équivalence par  $\rightarrow$  en appliquant la relation fonctionnelle  $\rightarrow_r$  jusqu'à tomber sur son représentant<sup>44</sup>. Il reste à caractériser le représentant d'une classe par la relation  $\rightarrow_r$  :

**Proposition 19 (Terminaison de l'opération “trouver”).**  $\forall e \in C, e \rightarrow_r e \Leftrightarrow e = r_C$ .

Il suffit donc d'appliquer  $\rightarrow_r$  jusqu'à avoir  $e \rightarrow_r e$  et on a trouvé le représentant de la classe.

**Proposition 20 (Union de deux classes).** Soient  $C_1, C_2 \in E/\rightarrow$  de représentants  $r_{C_1}$  et  $r_{C_2}$ .  $C_1 \cup C_2$  appartient à la clôture réflexive-transitive-symétrique de  $\rightarrow_r \cup \{(r_{C_1}, r_{C_2})\}$

**Exemple :** on reprend les unions sur le graphe 9.13 :

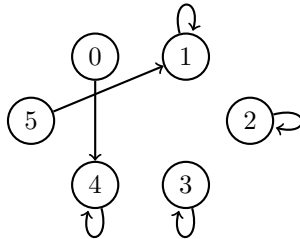


FIGURE 9.16 – Graphe après deux unions

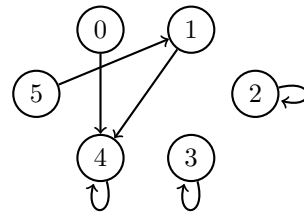


FIGURE 9.17 – Graphe final après une union

### 9.7.4 Implantation

On veut représenter la fonction  $\rightarrow_r$ , qui associe à chaque élément de  $\{0, \dots, n-1\}$  son parent dans l'arbre défini par la relation. C'est donc une fonction de  $\{0, \dots, n-1\}$  dans  $\{0, \dots, n-1\}$ . Un tableau d'entiers convient très bien :

44. On rappelle qu'étant donnée une relation fonctionnelle  $\mathcal{R}$  de  $E$  dans  $F$ , pour tous  $(e, f) \in E \times F$ ,  $e\mathcal{R}f \Leftrightarrow \mathcal{R}(e) = \{f\}$

| 0 | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|
| 4 | 4 | 2 | 3 | 4 | 1 |

FIGURE 9.18 –  $T[i] = j \Leftrightarrow i \xrightarrow[\text{graphe}]{} j$ 

```

struct UnionFind {
    unsigned int *parent;
    unsigned int cardinal;
    unsigned int classes_count;
};

struct UnionFind unionfind_create(unsigned int cardinal) {
    struct UnionFind uf;
    uf.repr = (unsigned int*)malloc(cardinal * sizeof(unsigned int));
    uf.cardinal = cardinal;
    for (unsigned int i = 0; i < cardinal; i++) {
        uf.parent[i] = i;
    }
    uf.classes_count = cardinal;
    return uf;
}

unsigned int unionfind_find(struct UnionFind uf, unsigned int e) {
    do {
        e = uf.parent[e];
    } while (e != uf.parent[e]); // tant que 'e' n'est pas le représentant
    return e;
}

struct UnionFind unionfind_union(struct UnionFind uf, unsigned r_a, unsigned r_b) {
    uf.parent[r_a] = r_b; // pourquoi pas... le choix est arbitraire
    uf.classes_count--;
    return uf;
}

```

**Remarque :** le type *Partition* est communément appelé *UnionFind* de par le nom des deux opérations élémentaires fondamentales du type.

#### Complexité temporelle :

- *trouver* : Dans le pire des cas, il faut itérer sur toute la classe d'équivalence qui peut au maximum contenir tout l'ensemble, c'est-à-dire  $n$  éléments. La complexité temporelle dans le pire cas est donc une fonction de  $O(n)$ .
- *unir* : fonction de  $O(1)$

**Reprise de l'exemple précédent :** on reprend l'ensemble  $E = \{0, 1, 2, 3, 4, 5\}$  de l'exemple précédent. Le tableau des parents est au départ :

$$T = [0, 1, 2, 3, 4, 5]$$

puisque chaque élément  $e$  est représentant de  $\{e\}$ .

- Après  $unir(P, 0, 4)$ , on a  $T = [4, 1, 2, 3, 4, 5]$
- Après  $unir(P, 5, 1)$ , on a  $T = [4, 1, 2, 3, 4, 1]$
- On a  $trouver(P, 5) = 1$  car  $T[5] = 1$  puis  $T[1] = 1$ , donc 1 est le représentant de 5
- Après  $unir(P, 5, 4) = unir(P, 1, 4)$ , on a  $T = [4, 4, 2, 3, 4, 1]$
- On a  $trouver(P, 5) = 4$  car  $T[5] = 1$  puis  $T[1] = 4$  puis  $T[4] = 4$  donc 4 est le représentant de 5

### 9.7.5 Amélioration de la complexité dans le pire cas

Les successeurs d'un représentant de classe dans le tableau forment une arborescence :

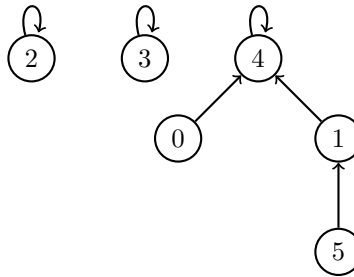


FIGURE 9.19 – Forêt représentative de la partition  $P$  de l'exemple

Si on doit parcourir toute la classe d'équivalence pour trouver le représentant, c'est que les successeurs de l'élément dont on cherche le représentant forment tout l'arbre. C'est donc une liste. L'objectif est d'éviter ce cas de figure. Si on parvient à éviter les arborescences "filiformes", on améliore considérablement le temps de parcours.

Dans le meilleur des cas, pour tout élément  $e \in E$ , on obtient le représentant de  $e$  directement par `uf.parent[e]`.

### Union pondéré

Prenons la partition représentée par la figure 9.19. Supposons qu'on veuille effectuer  $unir(P, 3, 4)$ . On a deux possibilités :

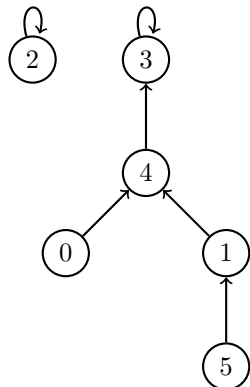


FIGURE 9.20 – Choix de 3 comme représentant

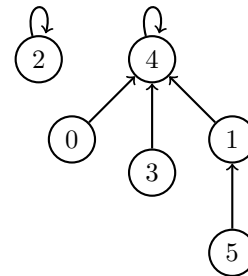


FIGURE 9.21 – Choix de 4 comme représentant

Le second choix est optimal puisque l'arbre est moins haut. Remonter l'arbre pour trouver le représentant est moins long. Donc le critère est de choisir comme enfant l'arbre le moins haut, pour minimiser la

hauteur du nouvel arbre. On ajoute donc dans `struct UnionFind` un tableau `highs`, rempli de 0 à la création et on modifie la routine d'union par :

```
#define MAX(X, Y) ((X) < (Y)) ? (Y) : (X)

struct UnionFind unionfind_union(struct UnionFind uf, unsigned r_a, unsigned r_b) {
    if (uf.highs[r_a] > uf.highs[r_b]) {
        uf.parent[r_b] = r_a;
    } else {
        uf.parent[r_a] = r_b;
        uf.highs[r_b] = MAX(1 + uf.highs[r_a], uf.highs[r_b]);
    }
    uf.classes_count--;
    return uf;
}
```

**Notation :** Après  $i$  opérations d'unions sur la partition, on note pour tous  $x \in E$  :

- $t_i(x)$  le nombre d'éléments de l'arbre de racine  $x$  après  $i$  unions sur la partition
- $h_i(x)$  la hauteur de l'arbre de racine  $x$  après  $i$  unions sur la partition

On observe que  $t_0(x) = 1$  et  $h_0(x) = 0$ .

$h_i(x)$  correspond à `uf.highs[x]` après  $i$  opérations d'union sur la partition.

**Proposition 21 (Borne sur la hauteur).** pour tout  $x \in E$ , pour tout  $i \in \mathbb{N}$ ,  $t_i(x) \geq 2^{h_i(x)}$ .

Il s'ensuit que pour  $n = \text{Card}(E)$ , on a  $\log_2(n) \geq \log_2(t_i(x)) \geq h_i(x)$ . La complexité temporelle dans le pire cas de *trouver* est donc en  $O(\log_2(n))$ .

**Démonstration :** On pose pour tout  $k \in \mathbb{N}$  la proposition  $P_k : \forall x \in E, t_k(x) \geq 2^{h_k(x)}$ .

Initialisation : pour  $k = 0$ ,  $t_0(x) = 2^{h_0(x)}$  car  $t_0(x) = 1$  et  $h_0(x) = 0$

Hérédité : Soit  $k \in \mathbb{N}$ . Soient  $x, y \in E$ . Supposons  $P_k$ .

On calcul  $\text{union}(P, x, y)$  :

- Si  $h_i(x) > h_i(y)$ , alors  $h_{i+1}(x) = h_i(x)$  et  $t_{i+1}(x) = t_i(x) + t_i(y) \geq t_i(x) \geq 2^{h_i(x)} = 2^{h_{i+1}(x)}$ . La hauteur et le nombre d'éléments de  $y$  sont inchangés. OK.
- Si  $h_i(x) \leq h_i(y)$ , on a déjà  $t_{i+1}(y) = t_i(y) + t_i(x)$ . Ensuite  $h_{i+1}(y) = \max(h_i(y), 1 + h_i(x))$ . Si  $h_{i+1}(y) = h_i(y)$ , tout est OK. Sinon, par l'inégalité supposée et puisque les hauteurs sont entières, on a  $h_{i+1}(y) = 1 + h_i(x) = 1 + h_i(y)$ . D'où  $t_{i+1}(y) \geq 2^{h_i(y)+1} \geq 2^{h_{i+1}(y)}$ . OK

Conclusion :  $(P_k)_{k \in \mathbb{N}}$  est initialisée et héréditaire, donc pour tout  $k \in \mathbb{N}$ ,  $P_k$  est vraie.

## Compression des chemins

La fonction *unir* est optimisée. On s'intéresse maintenant à optimiser *trouver*.

Lorsqu'on parcourt l'arbre pour trouver le représentant d'un élément, on peut au passage modifier cet arbre pour que tous les sommets visités par ce parcours soient "recollés" directement au représentant. En C, cela donne :

```

unsigned int unionfind_find(struct UnionFind uf, unsigned int e) {
    // Calcul du représentant
    unsigned r = e;
    do {
        r = uf.parent[r];
    } while (r != uf.parent[r]); // tant que 'r' n'est pas le représentant
    // Compression du chemin :
    // on parcourt à nouveau le chemin pour lier directement au représentant 'r'
    do {
        unsigned tmp = e;
        e = uf.parent[e];
        uf.parent[tmp] = r; // le sommet est lié directement au représentant
    } while (e != uf.parent[e]);
    return r;
}

```

La première fois qu'on cherche le représentant de  $e$ , on double le nombre d'opérations effectuées, pour effectuer la compression. Cependant, toutes les prochaines opérations *trouver* sur les éléments du chemin seront beaucoup plus rapide. C'est donc une optimisation au niveau global.

### 9.7.6 Exercices

**Exercice 93 (P'tite démo de caractérisation) [10].** Démontrer la proposition **Proposition 19**.

**Exercice 94 (Affichage d'une partition) [16].** Écrire une procédure :

```
void unionfind_display(struct UnionFind uf, char* buffer);
```

qui affiche la partition dans un tampon *buffer* grâce à la fonction `sprintf` du module *stdlib* de la bibliothèque standard.

*Note :* on pourra utiliser la commande “*man sprintf*” dans un terminal Linux pour lire la documentation de la fonction.

**Exercice 95 (Labyrinthe, monstres et trésors) [].** On s'intéresse à la génération et à l'exploration de labyrinthes parfaits selon différentes méthodes<sup>45</sup>.

45. Inspiré librement d'un sujet X-ENS d'informatique.



## GRAPHES (2) : VALUATION



On s'est intéressé dans la première partie sur les graphes aux graphes non valués, c'est-à-dire pour lesquelles on n'associait pas de distances ni aucun autre genre de valeurs aux arcs.

On étend la définition d'un graphe orienté en ajoutant une valuation. On étend les représentations de même. On décrit de nouveaux algorithmes qui tiennent compte de cette valuation. On s'intéresse en particulier :

- au calcul d'un arbre couvrant minimal d'un graphe
- au calcul du plus court chemin entre deux sommets d'un graphe

### 9.8.1 Définition et représentations

#### Définition 80 : Graphe valué

Un graphe valué sur  $\mathbb{R}$  est un triplet  $G = (S, A, c)$  où  $c : A \rightarrow \bar{\mathbb{R}}$  est une fonction de valuation qui à chaque arc associe une valeur.

Si  $(u, v) \notin A$ , on a par convention  $c((u, v)) = +\infty$

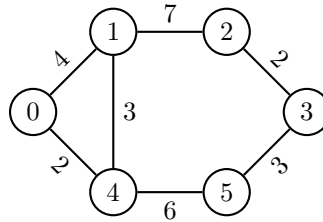
Le poids d'un chemin  $x_0 \rightarrow \dots \rightarrow x_k$  est  $c(x_0, x_1) + \dots + c(x_{k-1}, x_k)$ .

**Valuations autres :** Le graphe pourrait être valué dans d'autres monoïdes que  $(\mathbb{R}, +, 0)$ . Le poids n'a pas le même sens. Ci-dessous quelques exemples de monoïdes possibles :

- $(\mathbb{N}, +, 0)$  ou  $(\mathbb{R}, +, 0)$ , le poids d'un chemin  $\gamma = u \rightarrow^* v$  s'interprète comme le *coût* de  $\gamma$
- $([0, 1], \times, 1)$ , un poids s'interprète comme une probabilité de changement d'état entre deux sommets
- $(\mathcal{P}(\Sigma^*), \cdot, \{\varepsilon\})$ <sup>46</sup>, le poids d'un chemin est langage, et un graphe valué est un automate<sup>47</sup>.

**Exemple pour comprendre :** Le choix le plus commun de monoïde est  $(\mathbb{N}, +, 0)$ .

Le graphe suivant est valué sur ce monoïde :



Le chemin  $\gamma = 0 \rightarrow 4 \rightarrow 1 \rightarrow 2 \rightarrow 3$  est de poids  $2 + 3 + 7 + 2 = 14$ . On peut donc dire que le coût du chemin  $\gamma$  entre 0 et 3 est 14.

On observe que le plus court chemin entre 0 et 3 est  $0 \rightarrow 4 \rightarrow 5 \rightarrow 3$  (de poids 11) selon la relation d'ordre usuel sur  $\mathbb{N}$ . On appelle le coût de ce plus court chemin la *distance* entre 0 et 3.

Dans un graphe valué dans  $\mathbb{N}$ , on cherche généralement les chemins les plus courts, pour optimiser les déplacements. Le calcul du chemin le plus court se fait généralement en calculant son coût.

46. où  $\Sigma$  est un alphabet,  $\varepsilon$  est le mot vide, et où pour tous  $A, B \in \mathcal{P}(\Sigma^*)$ ,  $AB = \{ab \mid a \in A \wedge b \in B\}$

47. Automate généralisé, car d'ordinaire, le poids d'un arc est simplement un singleton  $\{a\} \subset \Sigma^1$

### 9.8.2 Signature du type abstrait

On désigne par *Monoïde* le type abstrait associé au monoïde  $(K, +, 0_K)$ . Dans la pratique, *Monoïde*  $\equiv$  *NombreRéal*

On étend la signature de graphe donnée dans la sous-section 9.6.2.

*Graphe*  $\langle$  *Sommet*, *Monoïde*  $\rangle$  utilise *Sommet*, *Liste*  $\langle$  *Sommet*  $\rangle$ , *Entier*, *Monoïde* :

- *poids*(*Graphe* : *G*, *Liste*  $\langle$  *Sommet*  $\rangle$  : *l*)  $\rightarrow$  *Monoïde* renvoie le poids du chemin  $l_1 \rightarrow \dots \rightarrow l_n$  formé par les sommets de la liste *l*

### 9.8.3 Représentations

On enrichit ici les représentations de graphe données dans la sous-section 9.6.3 en ajoutant la fonction de valuation du graphe évaluée sur chaque arc.

#### Matrice par liste de successeurs

On peut représenter un graphe valué  $G = (S, A, V)$  par listes d'adjacence en enrichissant la structure de noeud de liste. On utilise une liste *Liste*  $\langle$  *Sommet*, *Poids*  $\rangle$  qui stocke à la fois les successeurs d'un sommet mais également le poids de l'arête. C'est-à-dire que pour tout  $u \in S$ ,  $\text{succ}(u) := \{(v, c(u, v)) \mid u \rightarrow v\}$

#### Matrice d'adjacence

Dans le cas d'un graphe valué  $G = (S, A, c)$  valué dans  $\mathbb{R}$ , la matrice d'adjacence  $M \in \mathcal{M}_{|S|}(\mathbb{R})$  représentant  $G$  est telle que :

$$\forall (u, v) \in S \times S, M_{u,v} = \begin{cases} c((u, v)) & \text{si } (u, v) \in A \\ +\infty & \text{sinon} \end{cases}$$

**Remarque (l'infini en C) :** à partir de C99, l'existence de la constante INFINITY de type `double` est garantie par le standard dans le module *math.h*, donc quelque soit le compilateur. Il faut penser à ajouter le drapeau *-lm* pour compiler.

### 9.8.4 Premières propriétés

**Notation :** Soit  $G = (S, A, c)$  valué dans  $\mathbb{R}$  et  $x, y \in S$ . On note  $d(x, y)$  la borne inférieure des coûts des chemins de  $x$  à  $y$  dans  $G$ .

**Proposition 22 (Distance).** Soit  $G = (S, A, c)$  un graphe valué dans  $\mathbb{R}_+ \setminus \{0\}$  non orienté. La fonction  $d : S^2 \rightarrow \mathbb{R}_+$  telle que définie juste ci-dessus est une distance.

**Démonstration :** On vérifie les points suivants :

1. symétrie ( $\forall x, y \in S, d(x, y) = d(y, x)$ ) : OK puisque  $G$  n'est pas orienté. Les chemins de  $x$  à  $y$  sont les mêmes que ceux de  $y$  à  $x$
2. séparation ( $\forall x, y \in S, d(x, y) = 0 \Leftrightarrow x = y$ ) : OK car  $c$  est à valeur dans  $\mathbb{R}_+ \setminus \{0\}$
3. inégalité triangulaire ( $\forall x, y, z \in S, d(x, z) \leq d(x, y) + d(y, z)$ ) : OK par définition puisque  $d$  est une borne inférieure

**Important :** Si la valuation est à valeur dans  $\mathbb{R}$ , on peut annuler le coût d'une arête par d'autres arêtes de poids négatifs. Il est donc nécessaire pour avoir la séparation que la valuation soit à valeur dans  $\mathbb{R}_+ \setminus \{0\}$



Le résultat suivant est assez facile, mais s'avère utile pour justifier plusieurs algorithmes par la suite.

**Proposition 23 (Sous-optimalité).** Soit  $G = (S, A, c)$  un graphe valué dans  $\mathbb{R}$  et  $x, y \in S$ . Si  $\gamma : x = x_0 \rightarrow \dots \rightarrow x_n = y$  est un chemin de  $x$  à  $y$  de coût minimal, alors pour tous  $0 \leq i \leq j \leq n$ ,  $x_i \rightarrow \dots \rightarrow x_j$  est un chemin de coût minimal entre  $x_i$  et  $x_j$ .

**Démonstration :** Si un des sous-chemins n'était pas de coût minimal, on pourrait le remplacer par un sous-chemin de coût minimal, formant ainsi un nouveau chemin  $\gamma'$  de  $x$  à  $y$ .  $\gamma$  ne serait donc pas de coût minimal, puisqu'on aurait  $c(\gamma') < c(\gamma)$ .

### 9.8.5 Algorithmes de Kruskal et Prim

L'objectif des algorithmes de Kruskal et de Prim est de calculer l'arbre couvrant minimal d'un graphe connexe non orienté.

La définition d'un arbre couvrant est donnée dans l' **Exercice 92** .

#### Définition 81 : Arbre couvrant minimal

Soit  $G = (S, A, c)$  un graphe connexe non orienté valué dans  $\mathbb{R}$ . Un arbre couvrant minimal est un arbre couvrant de  $G$  dont la somme des poids des arêtes est minimal <sup>a</sup>.

<sup>a</sup>. Parmi les sommes des poids des arêtes des autres arbres couvrants.

#### Algorithme de Kruskal

L'algorithme de Kruskal regarde le graphe "de haut" et choisit les plus petites arêtes au fur-et-à-mesure pour coller les sommets. Il s'agit en fait de considérer chaque sommet une feuille. La liaison de deux sommets va former un "petit" arbre. Lorsqu'on relie deux petits arbres, on a un plus "gros" arbre. Lorsque toutes les feuilles sont reliés, on a un arbre couvrant.

On peut alors voir les sous-arbres comme une partition du graphe, et la liaison de deux sous-arbres comme une union. Si deux sommets  $x$  et  $y$  sont dans le même ensemble de la partition, c'est qu'ils sont déjà reliés dans un arbre. Il ne faut donc pas ajouter  $(x, y)$  à l'arbre couvrant en construction, sinon on crée un cycle.

On peut stocker chacune des arêtes du graphe dans une file de priorité, et simplement extraire de la file les arêtes jusqu'à avoir un arbre.

**Lemme 3.** Soit  $G = (S, A, c)$ . On note  $T$  le graphe construit par l'algorithme de Kruskal sur  $G$ . Pour tous  $x, y \in S$ ,  $x \rightarrow^* y$  dans  $T$  est équivalent à  $x \equiv y$  dans la partition.

**Démonstration :** On raisonne par récurrence sur la longueur  $k$  des chemins entre  $x$  et  $y$  dans  $T$ . Pour  $k = 0$ , on a  $x = y$  donc  $x \equiv y$ . Soit un chemin  $\gamma : x = x_0 \rightarrow \dots \rightarrow x_k = y$  dans  $T$  de longueur  $k > 0$ . Alors le chemin  $x_0 \rightarrow \dots \rightarrow x_{k-1} \neq 0$  est de longueur  $k - 1 \geq 0$ . Par HR,  $x = x_0 \equiv x_{k-1}$ . L'arête  $(x_{k-1}, y)$  a été ajouté en même temps que l'union de  $x_{k-1}$  et  $y$ . Donc  $x \equiv x_{k-1} \equiv y$ .

**Proposition 24 (Correction de l'algorithme de Kruskal).** L'algorithme de Kruskal termine et est correct. Il calcule bien un arbre couvrant minimal de  $G$ .

**Démonstration :** La terminaison est immédiate de par la suppression itérée des éléments de la file. Pour la correction, il faut vérifier deux points. D'abord que l'algorithme calcule effectivement un arbre couvrant (c'est-à-dire connexe et acyclique), ensuite que cet arbre couvrant est minimal.

1. Supposons que  $T$  n'est pas connexe. Il existe donc au moins deux composantes connexes  $T_1$  et  $T_2$ . Par connexité de  $G$ , il existe  $x \in T_1$  et  $y \in T_2$  tels que  $(x, y) \in A$ . Comme  $x \not\rightarrow^* y$  dans  $T$ ,  $x \not\equiv y$  d'après le lemme précédent. Or l'arête  $(x, y)$  a été extraite à un  $i^e$  tour de boucle quelconque de la file de priorité. Si  $x \not\equiv y$  au tour  $i$ , alors  $x \equiv y$  au tour  $(i + 1)$ . Ce qui est absurde puisque  $x \not\equiv y$  à la fin de l'exécution de l'algorithme. Donc  $T$  est connexe, donc couvrant.  
Supposons qu'il existe un cycle dans  $T$ . Il existe donc un chemin  $x \rightarrow^* y \rightarrow x$  dans  $T$  (avec  $x \neq y$ ). L'arête  $(y, x)$  a été ajoutée à  $T$  strictement avant ou strictement après la fermeture du chemin  $x \rightarrow^* y$ . Dans le premier cas, la fermeture du chemin  $x \rightarrow^* y$  n'est pas possible puisqu'on a déjà  $y \equiv x$ . idem dans l'autre cas.  
Donc  $T$  est connexe acyclique. Il est immédiatement couvrant puisque initialisé avec les sommets de  $T$ .
2. On procède par récurrence sur  $0 \leq i \leq |S| - 1$  pour montrer que  $T_i$  le graphe construit par l'algorithme après le  $i^e$  ajout d'arête est un graphe acyclique à  $i$  arêtes minimales. Pour  $i = 0$ , c'est le cas puisqu'il n'y a pas de coût possible. Supposons cette propriété vraie pour  $0 \leq i < |S| - 1$ . La  $(i + 1)^e$  arête ajoutée est celle de coût le plus petit possible parmi celles hors de  $T_i$ . En effet, si une arête  $(x, y)$  de coût encore plus petit avait été ajoutée, on aurait  $x \equiv y$  avant l'ajout, donc un cycle. Donc  $T_{i+1}$  est minimal par HR. On a  $T_n = T$ , donc  $T$  est minimal.

---

**Algorithme 21** : Algorithme de Kruskal

---

**Entrées** : Graphe :  $G = (S, A, c)$

```

1 Graphe :  $T = \text{creer}(|S|);$  // Arbre initialement vide
2 Partition :  $p = \text{creer}(|S|);$ 
3 FilePriorite<Sommet, Sommet, Coût> :  $\text{file} = \text{creer\_vide}();$  // La relation d'ordre de
   la file est sur le coût
4 pour tous  $x \in S$  faire
5   pour tous  $y \in \text{succ}(G, x)$  faire
6     insérer( $\text{file}, (x, y, c(x, y))$ );
7 tant que  $\neg \text{est\_vide}(\text{file})$  faire
8   Arc :  $(x, y) \leftarrow \text{extraire\_minimum}(\text{file});$  //  $\log_2 |A|$ .  $c(x, y)$  est ignoré car inutile
   autrement que pour l'ordre
9   si  $\text{trouver}(p, x) \neq \text{trouver}(p, y)$  alors
10      $p \leftarrow \text{unir}(p, x, y);$ 
11      $T \leftarrow \text{ajouter\_arc}(T, x, y);$ 
12      $T \leftarrow \text{ajouter\_arc}(T, y, x);$ 
13 retourner  $T;$ 
```

---

On effectue  $|S|$  opérations initialement pour construire l'arbre vide et la partition. Puis on fait  $|A|$  tours de boucle pour chacune des opérations suivantes :

1. insérer les arêtes dans la file de priorité (insertion de coût logarithmique)
2. construction de l'arbre (extraction de la file de coût logarithmique)

On note  $C_{\text{union}}$  et  $C_{\text{trouver}}$  les coûts des opérations *unir* et *trouver*.

Le coût temporel de l'algorithme de Kruskal est donc fonction de  $\Theta(|S| + |A|(\log_2 |A| + C_{\text{union}} + C_{\text{trouver}}))$ . Si on considère  $C_{\text{union}}$  et  $C_{\text{trouver}}$  en  $\Theta(1)$ , le coût temporel de l'algorithme de Kruskal est plus simplement fonction de  $\Theta(|S| + |A|\log_2 |A|)$ .

**Proposition 25 (Unicité de l'arbre couvrant minimal).** Si un graphe  $G = (S, A, c)$  connexe non orienté valué dans  $\mathbb{R}$  est tel que pour tous  $a_1, a_2 \in A$ ,  $c(a_1) \neq c(a_2)$  (valuation des arêtes deux à deux distincte sur  $A$ ), alors il existe un unique arbre couvrant minimal de  $G$ .

**Démonstration :** Si il existe deux arbres  $T_1$  et  $T_2$  couvrant minimaux distincts d'un même graphe  $G$ , alors il existe au moins une arête de  $G$  avec  $a \in T_1$  et  $a \notin T_2$ . On choisit celle de plus petit coût. On peut alors ordonner les arêtes de  $T_1$  selon leur coût par la liste  $[a_1, \dots, a_{i-1}, u_i, u_{i+1}, \dots, u_{|S|-1}]$  et les arêtes de  $T_2$  de même par la liste  $[a_1, \dots, a_{i-1}, v_i, v_{i+1}, \dots, v_{|S|-1}]$  où  $i$  est le premier indice pour lequel les deux suites diffèrent. On suppose sans perte de généralité que  $c(u_i) < c(v_i)$ . On ajoute  $u_i$  à l'arbre  $T_2$ . Comme  $T_2$  est acyclique maximal,  $T_2 \cup \{u_i\}$  possède un cycle. Comme  $T_1$  est un arbre, ce cycle n'est pas inclu dans  $T_1$ . Donc il existe une arête  $e$  de ce cycle qui n'appartient pas à  $T_1$ .  $e$  est donc une arête de  $\{v_i, \dots, v_{|S|-1}\}$ .  $T_3 = T_2 \cup \{u_i\} \setminus \{e\}$  est toujours connexe et a  $|S| - 1$  arêtes donc est un arbre couvrant. On a  $c(T_3) < c(T_2)$  car  $c(u_i) < c(e)$  donc ni  $T_1$  ni  $T_2$  ne sont minimaux, ce qui est absurde.

### Algorithme de Prim

L'algorithme de Prim effectue un parcours du graphe qui garantit la minimalité de l'arbre de parcours obtenu grâce à une file de priorité. L'algorithme de Prim voit donc le graphe "de l'intérieur", en se promenant dedans, contrairement à l'algorithme de Kruskal qui le voit "de haut".

De fait, on peut préciser en entrée l'algorithme le point de départ  $s_0$  du parcours. Comme le graphe est connexe, on peut aussi choisir un sommet de départ arbitraire (comme 0 si  $S = \{1, \dots, n\}$ ).

La structure des algorithmes de Prim et Dijkstra et le parcours effectué par chacun des algorithmes sont résolument identiques. La seule différence est que l'algorithme de Dijkstra effectue des calculs supplémentaires sur des distances.

---

#### Algorithme 22 : Algorithme de Prim

---

```

Entrées : Graphe :  $G = (S, A)$ , Sommet :  $s_0 \in S$ 
1 Tableau<Booleen> : sommets_visites [ $|S|$ ];
2 pour tous  $i \in S$  faire
3    $\text{sommets\_visites}[i] = \text{Faux}$ ;
4 Graphe :  $T = \text{creer}(|S|)$ ;
5 FilePriorite<Sommet, Coût> : decouverts  $\leftarrow \text{creer\_file\_vide}()$ ; // La relation d'ordre
   de la file est sur le coût
6 decouverts  $\leftarrow \text{enfiler}(\text{decouverts}, (s_0, 0))$ ;
7 tant que  $\neg \text{est\_vide}(\text{decouverts})$  faire
8   Sommet :  $x \leftarrow \text{defiler}(\text{decouverts})$ ; //  $\log_2 |A|$ .  $c(x, y)$  est ignoré car inutile
   autrement que pour l'ordre
9   si sommets_visites $[x] = \text{Faux}$  alors
10    sommets_visites $[x] = \text{Vrai}$ ;
11     $T \leftarrow \text{ajouter\_arc}(T, x, y)$ ;
12     $T \leftarrow \text{ajouter\_arc}(T, y, x)$ ;
13    pour tous  $y \in \text{succ}(G, x)$  faire
14       $\text{decouverts} \leftarrow \text{enfiler}(\text{decouverts}, (y, c(x, y)))$ ;
15 retourner  $T$ ;

```

---

**Définition 82 : Coupe**

Une coupe d'un graphe  $G = (S, A, c)$  est une partition  $C = \{S_1, S_2\}$  en deux sous-ensembles des sommets de  $G$ , c'est-à-dire que  $S = S_1 \sqcup S_2$  ( $S_1 \cap S_2 = \emptyset$ ).

On note  $A_C = (S_1 \times S_2) \cap A$  l'ensemble des arêtes de  $G$  qui relient directement les sommets de  $S_1$  aux sommets de  $S_2$ .

La coupe associée à un ensemble  $S' \subset S$  est  $C_{S'} = \{S', S \setminus S'\}$ .

**Définition 83 : Sommets incidents**

Soit  $G = (S, A)$  et  $A' \subseteq A$ . Les sommets incidents de  $A'$  est l'ensemble des sommets :

$$\{u \in S \mid \exists v \in S / \{u, v\} \in A'\}$$

Pour tous  $\{u, v\} \in A'$ , on a donc  $u$  et  $v$  des sommets incidents de  $A'$

**Proposition 26 (Propriété fondamentale des arbres couvrants minimaux).** Soit  $G = (S, A, c)$ . Soit  $T = (S, A_T)$  un arbre couvrant minimal de  $G$  et  $A' \subset A_T$ . On considère la coupe  $C = \{S', S \setminus S'\}$  associée aux sommets incidents de  $A'$ . On note  $a \in A_C$  l'arête de coût minimal de  $A_C$ . Alors il existe un arbre couvrant minimal qui contient  $A' \cup \{a\}$ .

**Démonstration :** Si  $a = \{x, y\}$  est une arête de  $T$ ,  $T$  est un arbre qui convient. Sinon, comme  $T$  est un arbre couvrant, il existe une chaîne  $x \rightarrow \dots \rightarrow y$  dans  $T$ . Il existe donc dans la chaîne une arête  $a' = \{x', y'\}$  telle que  $x' \in S'$  et  $y' \in S \setminus S'$ . On pose alors  $T' = T \cup \{a\} \setminus \{a'\}$ . Comme  $T'$  a  $|S| - 1$  arêtes et est sans cycle (on a supprimé le seul cycle ajouté par  $a$  en enlevant  $a'$ ), alors  $T'$  est un arbre couvrant de  $G$ . Par ailleurs,  $c(T') = c(T) + c(a) - c(a') \leq c(T)$  car  $a$  est de coût minimal dans  $A_C$ . Donc  $T'$  est un arbre couvrant minimal qui contient  $A' \cup \{a\}$ .

**Proposition 27 (Correction de l'algorithme de Prim).** L'algorithme de Prim termine et est correct. Il calcule bien un arbre couvrant minimal.

**Démonstration :** On visite chaque sommet du graphe une unique fois, donc l'algorithme effectue un parcours. En particulier, il termine et construit bien un arbre couvrant. Pour la minimalité on montre par récurrence sur  $0 \leq i < |S| - 1$  que le graphe  $T_i$  construit par l'algorithme de Prim après l'ajout de la  $i^e$  arête est toujours contenu dans un arbre couvrant minimal de  $G$ . En particulier, après l'ajout de  $|S| - 1$  arête, on a  $T_{|S|-1}$  un arbre couvrant minimal.

Initialisation : pour  $i = 0$ , aucune arête n'est ajoutée.  $G$  est non orienté connexe, donc il existe un arbre couvrant minimal de  $G$ . Cet arbre contient  $T_0$  qui est sans arêtes.

Hérédité : Soit  $0 \leq i < |S| - 1$ . On considère les sommets déjà reliés par des arêtes dans  $T_i$  (c'est-à-dire les sommets incidents aux arêtes de  $T_i$ ). Il s'agit des sommets déjà visités par l'algorithme. Les sommets découverts sont exactement ceux accessibles depuis ces sommets. Cela forme une coupe  $C$  entre les sommets visités et ceux non visités.  $A_C$  est l'ensemble des arêtes qui vont de sommets visités à des sommets non visités. L'algorithme de Prim ajoute  $a \in A_C$  l'arête de coût minimal de  $A_C$  pour former  $T_{i+1}$ . D'après la **Proposition 26**, il existe un arbre couvrant minimal de  $G$  qui contient  $T_{i+1}$ .

### 9.8.6 Bellman-Ford

On démontre les équations de Bellman pour justifier l'algorithme de Bellman-Ford, qui permet de calculer en temps fonction de  $O(|S|(|S| + |A|))$  la distance entre un sommet  $s$  et tous les autres

sommets du graphe. Cet algorithme peut être facilement adapté pour calculer le chemin de ce sommet à tous les autres sommets par un tableau de prédécesseurs.

**Notation :** Soit  $G = (S, A, c)$  valué dans  $\mathbb{R}$ . Pour tous  $s, x \in S$ ,  $k \in \mathbb{N}$ , on note  $d_k(s, x)$  la borne inférieure des coûts des chemins de  $s$  à  $x$  de longueur au plus  $k$  dans  $G$ . Si aucun chemin de longueur inférieure ou égale à  $k$  n'existe,  $d_k(s, x) = +\infty$ .

$d_k(s, x)$  peut être compris comme une “restriction” de  $d(s, x)$  aux chemins de longueur au plus  $k$ .

On commence par montrer par un premier lemme que  $d_{|S|}(s, x)$  est ce qu'on veut calculer. On décrit et démontre ensuite par un second lemme un moyen pratique de le calculer par récurrence sur  $k$ .

**Lemme 4.** Soit  $G = (S, A, c)$  un graphe valué dans  $\mathbb{R}$  sans circuit de coût strictement négatif accessible depuis  $s \in S$ , alors pour tout  $x \in S$ ,  $d(s, x) = d_{|S|}(s, x)$ .

**Démonstration :** On a immédiatement  $d(s, x) \leq d_{|S|}(s, x)$  puisque  $d(s, x)$  est une borne inférieure. Comme  $d_{|S|}(s, x)$  est optimal pour les chemins de longueur au plus  $|S|$ , on veut vérifier qu'il n'existe aucun chemin de longueur supérieure strictement à  $|S|$  qui soit plus court. Si un tel chemin  $\gamma$  existe, par principe des tiroirs il existe forcément un sommet  $y \in S$  qui apparaît deux fois dans  $\gamma$ . Le chemin de  $y$  à  $y$  dans  $\gamma$  est un cycle accessible depuis  $s$ . D'après l'hypothèse, le coût de ce cycle est positif. L'enlever diminue le coût total de  $\gamma$ . Au final, on peut enlever tous les doublons et avoir  $|\gamma| \leq |S|$ , donc  $d(s, x) \geq d_{|S|}(s, x)$ . On a double inégalité donc égalité.

**Lemme 5 (Équations de Bellman).** Soit  $G = (S, A, c)$  un graphe valué sur  $\mathbb{R}$ . Pour tous  $s, x \in S$  :

$$\begin{cases} d_0(s, x) &= \begin{cases} 0 & \text{si } s = x \\ +\infty & \text{sinon} \end{cases} \\ \forall k \in \mathbb{N}, \quad d_{k+1}(s, x) &= \min(d_k(s, x), \min\{d_k(s, y) + c(y, x) \mid (y, x) \in A\}) \end{cases}$$

**Démonstration :**

1. Il y a un seul chemin de longueur 0 de  $s$  à  $s$ . Il n'existe aucun autre chemin de longueur 0.
2. Soit  $k \in \mathbb{N}$ . On a d'abord l'inégalité :

$$d_{k+1}(s, x) \leq \min(d_k(s, x), \min\{d_k(s, y) + c(y, x) \mid (y, x) \in A\})$$

puisque les chemins considérés dans le membre de droite sont tous de longueur au plus  $(k+1)$ . Soit  $\gamma : s \rightarrow^* x$  de longueur  $|\gamma| \leq k+1$ . Si  $|\gamma| \leq k$ , c'est terminé. Si  $|\gamma| = k+1 > 0$ , on peut écrire  $\gamma = s \rightarrow^* y \rightarrow x$ . Donc  $d_{k+1}(s, x) \geq \min\{d_k(s, y) + c(y, x) \mid (y, x) \in A\}$ . On a les deux inégalités, donc égalité.

De cette récurrence on déduit un algorithme pour calculer les distances de  $s$  à tout  $x \in S$  par mémorisation des distances déjà calculés. C'est-à-dire qu'on garde en mémoire un tableau des distances aux  $|S|$  sommets qu'on maintient à jour au fur-et-à-mesure de l'algorithme pour ne pas avoir à le recalculer. Le tableau final donne alors les distances de  $s$  à tout  $x \in S$  d'après le premier lemme.

D'après les deux lemmes précédents, l'algorithme est correct. L'algorithme effectue une boucle de  $n$  tours sur les valeurs de  $k$ . Pour chacun de ces tours, il parcourt tous les sommets puis tous les successeurs de chaque sommet, c'est-à-dire toutes les arêtes. La complexité temporelle de l'algorithme est donc fonction de  $O(|S|(|S| + |A|))$  avec une représentation par listes d'adjacences si le calcul de  $c(x, y)$  est une opération élémentaire. Comme le coût d'une arête est stocké directement dans la liste d'adjacence, c'est bon.

**Calcul des chemins :** L'algorithme ne donne pour l'instant que les distances. On peut obtenir les chemins en eux-mêmes en ajoutant un tableau de prédécesseurs tel que  $predecesseur[x] = y$  si l'arête  $(y, x)$  est une arête de l'arbre de parcours minimal, avec  $predecesseur[s] = s$ . La mise à jour a lieu à la mise à jour de  $d_k(s, x)$ .

**Algorithme 23** : Algorithme de Bellman-Ford

---

**Entrées** : *Graphe* :  $G = (S, A, c)$ , *Sommet* :  $s$

```

1  Entier :  $n = |S|$ ;                                     // Pour alléger l'écriture ensuite
2  Matrice :  $d \in \mathcal{M}_{S, \{0, n\}}(\mathbb{R})$ ;             //  $d[x][k]$  donne  $d_k(s, x)$ 
3  pour tous  $x \in S \setminus \{s\}$  faire
4  |  $d[x][0] \leftarrow +\infty$ ;
5   $d[s][0] \leftarrow 0$ ;
6  pour  $k \in [1, \dots, n]$  faire
7  | pour tous  $x \in S$  faire
8  | |  $d[x][k] \leftarrow d[x][k-1]$ ;
9  | pour tous  $x \in S$  faire
10 | | | pour tous  $y \in \text{succ}(G, x)$  faire
11 | | | | si  $d[x][k-1] + c(x, y) < d[y][k]$  alors
12 | | | | |  $d[y][k] \leftarrow d[x][k-1] + c(x, y)$ ; //  $x$  permet d'arriver à  $y \Rightarrow x$  précède  $y$ 
12 | | | | | dans le parcours
13 retourner  $d$ ; // Dernière ligne (la  $(n+1)^e$ ) contient les distances de  $s$  à  $x$ 

```

---

**Simplification de l'algorithme**

Il est en vérité possible d'appliquer l'algorithme de Bellman-Ford *en place*. C'est-à-dire qu'au lieu d'avoir une matrice des  $d_k(s, x)$  avec  $k \in \{0, \dots, n\}$  et  $x \in S$ , on fait tous les calculs dans un seul tableau  $d[x]$  qui prendra successivement les valeurs  $d_0(s, x), \dots, d_n(s, x)$ . En fin d'algorithme, on aura  $d[x] = d_n(s, x)$  :

Que l'algorithme soit encore correct n'est pas évident au premier abord puisque les valeurs de  $d$  semblent dépendantes à l'intérieur de la boucle. Toutefois la dépendance ne fait que diminuer les valeurs prises tout en restant des coûts de chemins réels nécessairement au dessus de la valeur minimale. Donc l'algorithme calcule toujours la valeur minimale.

**Algorithme 24** : Algorithme de Bellman-Ford simplifié

---

**Entrées** : *Graphe* :  $G = (S, A, c)$ , *Sommet* :  $s$

```

1  Entier :  $n = |S|$ ;                                     // Pour alléger l'écriture ensuite
2  Tableau  $\langle \mathbb{R} \rangle$  :  $d$ ;                                 //  $d[x]$  donne  $d_k(s, x)$  après le  $k^e$  tour
3  pour tous  $x \in S \setminus \{s\}$  faire
4  |  $d[x] \leftarrow +\infty$ ;
5   $d[s] \leftarrow 0$ ;
6  pour  $k \in [1, \dots, n]$  faire
7  | pour tous  $x \in S$  faire
8  | | pour tous  $y \in \text{succ}(G, x)$  faire
9  | | | si  $d[x] + c(x, y) < d[y]$  alors
10 | | | |  $d[y] \leftarrow d[x] + c(x, y)$ ; //  $x$  permet d'arriver à  $y \Rightarrow x$  précède  $y$  dans le
10 | | | | | parcours
11 retourner  $d$ ; // Dernière ligne (la  $(n+1)^e$ ) contient les distances de  $s$  à  $x$ 

```

---

**Démonstration (avec les mains)** : On montre que pour tout  $k \in \{1, \dots, n\}$ , on a  $d[x] = d_k(s, x)$  après la boucle de la ligne 7. On procède par récurrence. C'est vrai au début de l'algorithme, après

l'initialisation des lignes 3 à 5. Supposons que cela soit vrai pour  $k \in \{1, \dots, n-1\}$ . Si  $d[x]$  n'a pas été modifié depuis le début de la  $k^e$  boucle, l'assignation à  $d[y]$  est équivalente à celle de l'algorithme non simplifié. Sinon, comme  $d[x] \leq d_k(s, x)$ ,  $d[y]$  est inférieure ou égale à sa valeur dans l'algorithme non simplifié, c'est-à-dire  $d[y] \leq d_{k+1}(s, y)$  à la fin du parcours des arêtes. Par ailleurs, la valeur correspond toujours au coût d'un chemin de longueur au plus  $k+1$  vers  $y$  donc  $d[y] \geq d_{k+1}(s, y)$  puisque  $d_{k+1}(s, y)$  est minimal. Donc pour tout  $y \in S$ ,  $d[y] = d_{k+1}(s, y)$ .

### Détection des cycles à coût strictement négatif

L'algorithme de Bellman-Ford peut être étendu pour détecter les cycles de coût strictement négatif accessibles depuis le sommet de départ. Dans le cas où de tels cycles existent, la distance entre le sommet de départ et un autre sommet est toujours  $-\infty$  puisque le cycle peut être emprunté une infinité de fois.

Le lemme suivant permet de détecter les cycles à coût strictement négatifs :

**Lemme 6 (Cycles à coût strictement négatifs).** Soit  $G = (S, A, c)$  un graphe non orienté valué dans  $\mathbb{R}$ . Soit  $s \in S$ . Il existe un cycle de coût strictement négatif  $z \rightarrow \dots \rightarrow z$  si et seulement si il existe  $y \in S$  tel que  $d_{n+1}(s, y) < d_n(s, y)$ .

Il suffit alors d'effectuer un tour de boucle supplémentaire dans l'23 pour calculer  $d_{n+1}$  et ajouter une boucle en fin d'algorithme pour vérifier que la condition de cycle négatif n'est pas vérifiée.

#### Démonstration :

1. ( $\Leftarrow$ ) Si il existe  $y \in S$  tel que  $d_{n+1}(s, y) < d_n(s, y)$ , alors il existe un chemin  $\gamma : s \rightarrow^* y$  de longueur  $(n+1)$  de  $s$  à  $y$  de coût strictement inférieur à tous les chemins de  $s$  à  $y$  de longueur au plus  $n$ . Par principe des tiroirs il existe un sommet qui se répète dans  $\gamma$ . Si le cycle déduit est de coût positif ou nul, on peut le supprimer et diminuer le coût de  $\gamma$ , qui est alors au plus de longueur  $n$ , ce qui n'est pas possible. Donc le cycle déduit est de coût strictement négatif.
2. ( $\Rightarrow$ ) Puisqu'on peut supprimer les cycles des chemins de longueur  $n+1$ , on a toujours  $d_{n+1}(s, y) \leq d_n(s, y)$ . On a immédiatement  $d_{n+1}(s, y) \geq d_n(s, y)$ . Donc  $d_{n+1}(s, y) = d_n(s, y)$ . Par le même raisonnement, on a pour tout  $k \in \mathbb{N}$   $d_{n+k}(s, y) = d_n(s, y)$ . Supposons qu'il existe un cycle  $z \rightarrow \dots \rightarrow z$  de longueur  $k$  de coût strictement négatif accessible depuis  $s$ . On choisit  $\gamma$  un chemin de longueur au plus  $n$  qui relie  $s$  à  $z$ . On peut concaténer à  $\gamma$  le cycle strictement négatif. On a alors  $d_{n+k}(s, z) < d_n(s, z)$ , ce qui est absurde.

### 9.8.7 Algorithme de Dijkstra

En restreignant encore les hypothèses et en supposant le graphe valué sur  $\mathbb{R}_+ \setminus \{0\}$  au lieu de  $\mathbb{R}$ , on peut obtenir l'algorithme de Dijkstra de complexité temporelle fonction de  $O(|E| + |S| \log_2 |S|)$ . La restriction est plus forte puisqu'au lieu d'interdire les circuits à coût strictement négatifs on interdit les arcs à coût strictement négatifs.

### 9.8.8 Algorithme de Johnson

L'algorithme "classique" pour calculer le plus court chemin entre chaque paire de sommets d'un graphe est l'algorithme de Floyd(-Warshall), qui est une généralisation de l'algorithme de Roy-Warshall. Cependant, cet algorithme est assez lent puisqu'il effectue le calcul en  $O(|S|^3)$ .

On montre ici l'algorithme de Johnson, qui effectue ces calculs en  $O(|S||A| + |S|^2 \log_2 |S|)$ . Il construit un nouveau graphe à partir de  $G$  sur lequel il applique d'abord Bellman-Ford puis itère l'algorithme de Dijkstra.

**Algorithme 25** : Algorithme de Bellman-Ford avec détection des cycles négatifs

---

**Entrées** : *Graphe* :  $G = (S, A, c)$ , *Sommet* :  $s$

```

1  Entier :  $n = |S|$ ;                                     // Pour alléger l'écriture ensuite
2  Matrice :  $d \in \mathcal{M}_{S, \{0, n\}}(\mathbb{R})$ ;             //  $d[x][k]$  donne  $d_k(s, x)$ 
3  pour tous  $x \in S \setminus \{s\}$  faire
4  |  $d[x][0] \leftarrow +\infty$ ;
5   $d[s][0] \leftarrow 0$ ;
6  pour  $k \in [1, \dots, n+1]$  faire
7  | pour tous  $x \in S$  faire
8  | |  $d[x][k] \leftarrow d[x][k-1]$ ;
9  | pour tous  $x \in S$  faire
10 | | pour tous  $y \in \text{succ}(G, x)$  faire
11 | | | si  $d[x][k-1] + c(x, y) < d[y][k]$  alors
12 | | | |  $d[y][k] \leftarrow d[x][k-1] + c(x, y)$ ; //  $x$  permet d'arriver à  $y \Rightarrow x$  précède  $y$ 
12 | | | | dans le parcours
13 pour tous  $y \in S$  faire
14 | si  $d[y][n+1] < d[y][n]$  alors
15 | | retourner Erreur : présence de cycle de coût strictement négatif;
16 retourner  $d$ ; // Avant-dernière ligne  $((n+1)^e)$  contient les distances de  $s$  à  $x$ 

```

---

**Algorithme 26** : Algorithme de Dijkstra

---

**Entrées** : *Graphe* :  $G = (S, A, c)$ , *Sommet* :  $s_0 \in S$

```

1  Tableau<Coût> :  $d[|S|]$ ;                                     // Tableau des distances
2  Tableau<Sommet> :  $\text{pred}[|S|]$ ;                               // Tableau/arbre des prédécesseurs
3  Tableau<Booleen> :  $\text{sommets\_visites}[|S|]$ ;
4  pour tous  $x \in S$  faire
5  |  $\text{sommets\_visites}[x] = \text{Faux}$ ;
6  |  $d[x] = +\infty$ ;
7  |  $\text{pred}[x] = \text{Indéfini}$ ;                                     // -1 par exemple
8   $d[s_0] = 0$ ;
9  FilePriorite<Sommet, Coût> :  $\text{file} = \text{creer\_vide}()$ ; // La relation d'ordre de la file
9  | est sur le coût
10  $\text{insérer}(\text{file}, (s_0, 0))$ ;
11 tant que  $\exists s \in S / \text{sommets\_visites}[s] = \text{Faux}$  faire
12 | Sommet :  $x = \text{extraire\_minimum}(\text{file})$ ; //  $d[x]$  est ignoré car inutile autrement
12 | | que pour l'ordre
13 | |  $\text{sommets\_visites}[x] \leftarrow \text{Vrai}$ ;
14 | | pour tous  $y \in \text{succ}(G, x)$  faire
15 | | | si  $d[y] > d[x] + c(x, y)$  alors
16 | | | |  $d[y] \leftarrow d[x] + c(x, y)$ ;
17 | | | |  $\text{pred}[y] \leftarrow x$ ;
18 | | | si  $\neg \text{sommets\_visites}[y]$  alors
19 | | | |  $\text{insérer}(\text{file}, (y, d[y]))$ ;
20 retourner  $d, \text{pred}$ ;

```

---



### 9.8.9 Exercices



## GRAPHERS (3) : FLOTS



Partie IV

# LE VRAI MONDE DE LA RÉALITÉ RÉELLE



# INTRODUCTION



Cette quatrième partie du cours d'informatique prend le contrepied de la troisième partie. Son contenu de fond appartient à un cadre beaucoup plus pratique de la programmation.

Si le lecteur assidu et consciencieux des trois parties précédentes doit maintenant maîtriser assez bien le langage C<sup>48</sup> d'un point de vue des connaissances syntaxiques et sémantiques, et doit également posséder suffisamment de connaissances théoriques pour comprendre les fondements de l'informatique, cela ne suffit pas à en faire un développeur compétent. Il lui manque encore certains points fondamentaux qui agissent plus au niveau professionnel que personnel. On fait ainsi la distinction entre la programmation pure et le développement d'application. Il y a dans le mot *développement* l'idée de faire grandir un programme, de le faire évoluer dans une certaine direction. Cet agrandissement se doit d'être contrôlé de la même manière qu'on fait grandir un arbre. Si l'arbre grandit de travers, il n'y a plus moyen par la suite de le redresser sans d'immenses difficultés. Il devient presque nécessaire de planter un nouvel arbre et de reprendre de zéro.

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>49</sup> et impossible à modifier en profondeur sans tout reprendre de rien. Cette quatrième partie se veut apporter quelques outils pour conserver une stabilité dans le développement d'applications complexes, et permettre de programmer dans un cadre véritablement concret. Elle vise en particulier :

- la recherche autonome de ressources d'apprentissage de l'informatique
- le minimum des outils nécessaires pour le développement d'un projet informatique :
  - ◆ *make* : un premier outil relativement simple et puissant pour l'automatisation de construction logiciel
  - ◆ *gdb* : LE débogueur
  - ◆ *Git* : LE gestionnaire de versions
  - ◆ le langage *Markdown*, pour l'écriture de notes et de *READMEs*
- le développement "propre" de projets par :
  - ◆ la prévention et la minimisation des erreurs incontrôlées *via* les tests globaux et les tests unitaires
  - ◆ l'utilisation d'un style d'écriture clair
  - ◆ l'internationalisation du code<sup>50</sup>
  - ◆ l'utilisation efficiente des commentaires
  - ◆ l'écriture de documentation, dont on discutera de l'automatisation par intelligence artificielle

L'objectif est de rendre le lecteur relativement autonome en discutant des fondamentaux du développement d'application, que l'on distingue de la programmation pure qui n'est finalement que la partie centrale, le cœur, du développement, mais non son tout.

---

48. Ainsi que nombre de notions gravitant autour comme les représentations binaires des nombres ou quelques principes de la programmation système

49. 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.

50. déjà évoqué en fin de première partie et début de deuxième

# OUTILS POUR LA PROGRAMMATION



*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. D'autres logiciels comme *CMake* sont aussi utilisables et utilisés.

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.

## 10.1.1 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*<sup>1</sup>.

## 10.1.2 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

---

1. Sous-entendu : il faut placer un fichier *Makefile* dans le répertoire où il y a le code, sinon ça fonctionne pas.

- 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>2</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>
```

2. Mais toujours pas fou...

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.

### 10.1.3 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" 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
```

### 10.1.4 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>3</sup> :

| Symbole             | Signification                                                                                   |
|---------------------|-------------------------------------------------------------------------------------------------|
| <code>\$@</code>    | Nom de la cible                                                                                 |
| <code>\$&lt;</code> | Nom de la première ressource                                                                    |
| <code>\$?</code>    | La liste des noms, séparés par des espaces, de toutes les ressources plus récentes que la cible |
| <code>\$^</code>    | 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
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

```

### 10.1.5 Réduire le nombre de règles avec la *stem*

Le symbole `%` désigne le “radical” *quelconque* d'un mot (en anglais *stem*, littéralement *tige*<sup>4</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

3. 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. La traduction est de moi. Je ne sais pas comment traduire autrement



ensuite pour compiler qu'à écrire *make*. Il n'y a donc plus à réécrire la liste des fichiers sources de module pour compiler.

### 10.1.6 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>5</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
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 $@
```

5. Voir <https://www.grymoire.com/Unix/Bourne.html#uh-4> pour les autres.

### 10.1.7 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 finissant cette fois-ci par *.o*”

Ce qui résout 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 !

### 10.1.8 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 sera 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>
```

### 10.1.9 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 *build* 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 (c’est-à-dire utiliser un répertoire pour les sources<sup>6</sup> et un second répertoire pour les entêtes<sup>7</sup>)
- 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’extensions “.a” ou “.so”)



## UTILISER UN DÉBUGUEUR



Le déboguage est une composante fondamentale du développement logiciel. En effet, il est crucial 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 les plus basiques qu’il propose.

6. Le nom standard est *src*

7. Le nom standard est *include*




La prise de notes efficace est une nécessité dans beaucoup de cadres. En particulier, la mise en forme de ces notes est généralement rébarbative et un frein dans l'écriture. Le langage Markdown se veut une solution à ce problème. Sa syntaxe est minimale pour écrire le plus vite possible.

Le Markdown est très utilisé dans le développement pour écrire des documentations rapides, pour écrire des *READMEs*, fichiers servant à indiquer les informations générales sur le contenu d'un dossier. Il peut aussi servir à écrire des rapports courts, à prendre des notes en réunions, écrire des idées, etc. . .

**Logiciel :** on peut utiliser [Obsidian](#), gratuit et probablement le meilleur sur le marché, bien que non libre. . . Il effectue la mise en page automatiquement.

Le formatage utilise les règles de base suivantes, selon la spécification [CommonMark](#) :

|                                                  |                                                                                     |
|--------------------------------------------------|-------------------------------------------------------------------------------------|
| <code>*texte en italique*</code>                 | <i>texte en italique</i>                                                            |
| <code>**texte en gras**</code>                   | <b>texte en gras</b>                                                                |
| <code># Titre 1</code>                           | Titre 1                                                                             |
| <code>## Titre 2</code>                          | Titre 2                                                                             |
| <code>### Titre 3</code>                         | Titre 3                                                                             |
| <code>[Link to example](example.com)</code>      | <a href="#">Link to example</a>                                                     |
| <code>![Link to image](chemin/smiley.png)</code> |  |

**Remarque 1 :** il est possible de combiner les symboles. Ainsi, `***Salut!***` donne ***Salut !***.

**Remarque 2 :** les liens d'image comme de "site" peuvent être des chemins internet ou locaux au système de fichier.

On peut y ajouter les listes :

- element de liste
- element de liste
- element de liste

- element de liste
- element de liste
- element de liste

qui peuvent être numérotés :

1. element de liste
2. element de liste
3. element de liste

1. element de liste
2. element de liste
3. element de liste

On peut aussi insérer des lignes de séparation par “- -” qui donne :

et insérer du code :

```
```LANGAGE (par exemple C ou Python)
du code, plein de code
sur pleins
de lignes
```
```

```
1 | printf("Salut les gens");
2 | // et d'autre code
```

## Mathématiques en Markdown

L'écriture de notes scientifiques induit la nécessité d'écrire des mathématiques. Markdown n'a pas de standard pour écrire des mathématiques.

Heureusement, la plupart des logiciels d'éditions Markdown, dont Obsidian, possèdent un interpréteur  $\text{\LaTeX}$  pour les formules mathématiques. Il suffit, comme en  $\text{\LaTeX}$ , d'écrire les formules entre dollars simples  $\$formule\$$  ou doubles  $\$\$formule\$\$$ .

# Théorème de Pythagore

Soit  $ABC$  un triangle.

$ABC$  est rectangle en  $\widehat{BAC}$  ssi :

$$AB^2 + AC^2 = BC^2$$

## Théorème de Pythagore

Soit  $ABC$  un triangle.

$ABC$  est rectangle en  $\widehat{BAC}$  ssi :

$$AB^2 + AC^2 = BC^2$$

On peut trouver l'intégralité des symboles mathématiques  $\text{\LaTeX}$  en deux versions :

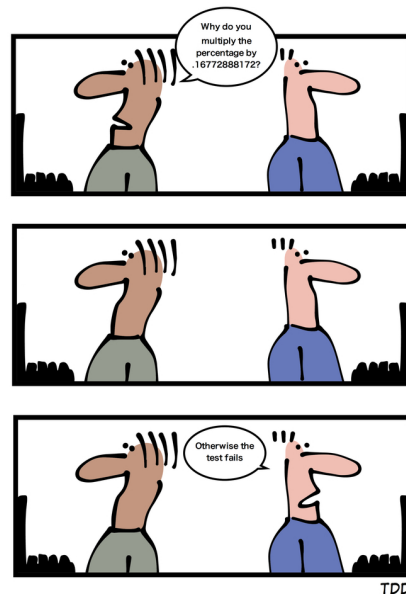
- une version légère avec seulement les symboles mathématiques basiques [https://www3.nd.edu/~nmark/UsefulFacts/LaTeX\\_symbols.pdf](https://www3.nd.edu/~nmark/UsefulFacts/LaTeX_symbols.pdf)
- une version complète contenant l'intégralité des symboles de  $\text{\LaTeX}$  <https://ctan.tetaneutral.net/info/symbols/comprehensive/symbols-a4.pdf>

# PROGRAMMER PROPREMENT

À PROPOS DU STYLE

LA GESTION DES ERREURS

“Le test de programmes peut être une façon très efficace de montrer la présence de bugs mais est désespérément inadéquat pour prouver leur absence” (Edsger Dijkstra)



La gestion d'erreurs en programmation est absolument fondamentale :

- un ordinateur physique n'est pas théorique, et un programme subit tous les défauts de l'implantation physique de l'ordinateur. On peut par exemple penser à la limitation de la mémoire, qui n'est pas infini. Une allocation dynamique peut donc échouer dans le cas où on arrive "au bout"
- les programmes développés sur un ordinateur ont été écrits par des humains faillibles. Ainsi, il n'est jamais sûr à 100% qu'un programme écrit soit correct, bien qu'il soit démontré correct en théorie.
- etc. . .

Dans le cas de "petits" programmes, ne pas gérer les erreurs n'a qu'une incidence minime sur le développement. Dans le cas de plus grands projets, des erreurs humaines ont de fortes chances de se glisser dans les petits programmes qui constituent certaines étapes du développement et les tester et les gérer est extrêmement important pour éviter *au moins pire* des bugs incontrôlés et surtout non localisés.

Par ailleurs, les manquements à la gestion des erreurs et aux tests sont la source première des failles de sécurité des systèmes informatiques.<sup>1</sup>

### 11.2.1 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.

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 et je sais de quoi je parle.

2. Ah bon ?

Avec un malloc cela ressemblerait à :

```
int *x = NULL; // TOUJOURS initialiser à 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 limitées. 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.

### 11.2.2 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 *fd = fopen(filename, "mode");
if (!fd) {
    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 bugs : une routine bugué est plus facilement corrigeable 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



## BENCHMARKS



Partie V

# ANNEXES

# CORRECTION DES 95 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 deux solutions (optimalité, lisibilité, rigueur, etc...). Il n'est ni garanti que les programmes proposés soient optimaux, ni que les démonstrations proposées soient les plus élégantes.

Toute proposition de démonstration ou de code peut être envoyée au contact donné en préface dans l'optique d'amélioration des futures éditions.

**Note quant aux codes :** certains codes ne sont ni annotés ni commentés. Pour ne pas surcharger le document, on indique simplement le chemin dans le répertoire des solutions du fichier de code qui contient la correction.



## PREMIÈRE PARTIE



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

### 12.1.1 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 :

- $\vee : (a, b) \mapsto a + b - ab$
- $\oplus : (a, b) \mapsto |a + b - 2ab| (= |a - b|)$
- $\wedge : (a, b) \mapsto a \times b$
- $\neg : a \mapsto 1 - a$

**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>1</sup> :

$$*(e) = 1 \Leftrightarrow \bigvee_{i=1}^{\text{Card}(T)} (e = t_i) \quad (12.1)$$

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 :

$$\begin{aligned} e = t_i &\Leftrightarrow \bigwedge_{k \in K_1(t_i)} (e_k = t_{i,k}) \bigwedge_{k \in K_0(t_i)} (e_k = t_{i,k}) \\ &\Leftrightarrow \bigwedge_{k \in K_1(t_i)} (e_k = 1) \bigwedge_{k \in K_0(t_i)} (e_k = 0) \\ &\Leftrightarrow \bigwedge_{k \in K_1(t_i)} e_k \bigwedge_{k \in K_0(t_i)} \neg e_k \end{aligned}$$

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>2</sup>

1. On explicite dans la suite l'expression de  $e = t_i$ .

2. 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* car  $(\mathcal{B}, \vee, \wedge)$  est un semi-anneau. On en retrouve des conséquences dans 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 :  $(01000 \ll 2) \gg 2 = 00000 \neq 01000$

### 12.1.2 Opérations arithmétiques

**Solution 7 (Du décimal au binaire).**

- |                  |                  |
|------------------|------------------|
| • $(01001100)_2$ | • $(11010011)_2$ |
| • $(10111100)_2$ | • $(00000100)_2$ |
| • $(00100001)_2$ | • $(11101110)_2$ |
| • $(01101101)_2$ | • $(10100001)_2$ |
| • $(01011100)_2$ | • $(01111110)_2$ |

**Solution 8 (P'tites démos).**

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

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

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

Proposition 3 :

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

Proposition 4 :

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

Proposition 5 : 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 : Soit  $v = v_{N-1} \dots v_0 \in \mathcal{B}^N$ .

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

### 12.1.3 Opérations sur les nombres à virgules

**Solution 9 (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.

**Solution 10 (Racine carrée inverse rapide (1)).**

On note  $P = x^2 + y^2 + z^2$ . Pour simplifier les notations, on note pour  $x \in \mathbb{R}$  :

- $itof(x) = btof(btoi^{-1}(x))$
- $ftoi(x) = btoi_s(ftob(x))$

On utilise ici trois résultats :

$$\log_2 \left( \frac{1}{\sqrt{P}} \right) = -\frac{1}{2} \log_2(P) \quad (12.2)$$

$$\forall x \in \mathbb{R}_{f32}, x = itof(A^{-1}(\log_2(x) - B)) \quad (12.3)$$

$$\forall x \in \mathbb{R}_{f32}, \log_2(x) = Aftoi(x) + B \quad (12.4)$$

En utilisant l'égalité (12.2) dans (12.3) en  $x = \frac{1}{\sqrt{P}}$ , on obtient :

$$\frac{1}{\sqrt{P}} = itof \left( -\frac{1}{2A} (\log_2(P) + 2B) \right)$$

En injectant (12.4), on a alors :

$$\frac{1}{\sqrt{P}} = itof \left( -\frac{1}{2} ftoi(P) - \frac{3B}{2A} \right)$$

### 12.1.4 Représentation hexadécimale

**Solution 11 (Conversion binaire-hexadécimale).**

- $713705 = (1010\ 1110\ 0011\ 1110\ 1001)_2 = 0xAE3E9$
- $8.8 = (1.00011001001001001001)_2 \times 2^3 = 0100\ 0001\ 1000\ 1100\ 1001\ 0010\ 0100\ 1001 = 0x418C9249$
- $42 = (0001\ 1010)_2 = 0x1A$
- $-1.1 = -(1.00010010010010010010010)_2 \times 2^1 = 1100\ 0000\ 0000\ 1001\ 0010\ 0100\ 1001\ 0010 = 0xC0092492$
- $101 = (0110\ 0101)_2 = 0x65$

### 12.1.5 Caractères ASCII

**Solution 12 (Traduction ASCII 1).** La chaîne de caractères décrite est :

“Minitel = GOAT”

**Solution 13 (Traduction ASCII 2).** Le mot binaire qui décrit cette chaîne est :

0x4F6820210A5175692065732D74752021083F



DEUXIÈME PARTIE



### 12.2.1 Bases du langage

#### Variables

**Solution 14 (Intervention de variables par effet de bord).**

**Code :** voir solutions/part2/chapter2/inter\_var\_side\_effect.c

#### Formatage de chaînes de caractères

**Solution 15 (Taille des types).**

**Code :** voir solutions/part2/chapter2/taille\_types.c

#### Opérateurs sur les variables

**Solution 16 (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 17 (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 18 (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 19 (Interversion sans effet de bord (1)).**

```

1 | #include <stdio.h>
2 | #include <stdlib.h>
3 |
4 | int main() {
5 |     int a = ...; // valeur quelconque
6 |     int b = ...; // valeur quelconque
7 |
8 |     // l'échange :
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'interversion, et  $a_s$  et  $b_s$  les valeurs de  $a$  et  $b$  succédant à l'interversion. 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 20 (Interversion 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 21 (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 22 (Valeur absolue (1)).** 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; // valeur quelconque positive ou négative
6      unsigned int abs_x = x;
7      unsigned int positive = (abs_x >> 31) - 1; // -1 si positif, 0 si négative
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 23** (Quelques évaluations entières).

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 24.**

```

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 25** (Question d'âge).

**Code :** voir solutions/part2/chapter2/question\_d\_age.c

**Solution 26** (Suite de Fibonacci).

**Code :** voir solutions/part2/chapter2/suite\_fibonacci\_1.c

**Solution 27** (Test de primalité).

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>3</sup> est donc le suivant :

```

1  #include <stdio.h>
2  #include <stdlib.h>
3
4  #define N ...
```

---

3. 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).

```

5
6 int main() {
7     char is_primal = N == 2 || (N > 2 && N % 2 != 0); // nécessaire
8     for (int i = 3; i < N-1 && is_primal; i += 2) {
9         is_primal = (N % i == 0) ? 0 : is_primal;
10    }
11
12    if (is_primal) {
13        printf("Premier !\n");
14    } else {
15        printf("Pas premier...\n");
16    }
17
18    return EXIT_SUCCESS;
19 }

```

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  $\sqrt{n}$ .

Il suffit donc de vérifier la divisibilité de  $n$  par les entiers impairs  $m \leq \sqrt{n}$  :

```

10 for (int i = 3; i*i <= N && is_primal; i += 2) {
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 (il ne s'agit pas d'une constante).

## Routines

### Solution 28 (Encore Fibonacci).

**Code :** voir solutions/part2/chapter2/suite\_fibonacci\_2.c

### Solution 29 (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 30 (Quelques procédures inutiles pour devenir un bot efficace).**

**Code :** solutions/part2/chapter2/become\_bot.c

**Solution 31 (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 32 (Distance de Manhattan).** Le type `double` est sur  $8 = 2 \times 4$  octets. Il permet donc de stocker deux nombres de type `float`.

Si  $x$  est une variable de type `double`,  $\&x$  est aussi l'adresse des premiers 4 octets, et  $\&x + 4$  est *mathématiquement* l'adresse des seconds 4 octets. Il faut toutefois faire attention. En C,  $\&x$  est de type `double`. Si on veut lui ajouter 4, il faut le projeter en un pointeur de type sur 4 octets, et ajouter 1 à ce pointeur :

```

1 double x; // 8 octets
2 float *x_left = (float*)&x; // pointeur sur 4 octets de gauche
3 float *x_right = (float*)&x + 1; // pointeur sur 4 octets de droite
```

**Code complet :** voir solutions/part2/chapter2/manhattan\_dist.c

**Solution 33 (Valeur absolue (2)).** On veut simplement modifier le bit de signe. Toutefois, les opérations bit-à-bits sont interdites sur les nombres à virgules.

Les projections classiques `float x = (float)n`; et `int n = (int)x`; modifient la représentation binaire, ce qui n'est pas souhaité. On utilise à la place une idée semblable à celle utilisée pour l' **Exercice 32** :

```

1 float abs(float x) {
2     int int_x = *((int*)&x); // réinterprétation entière
3     int_x ^= 0x7FFFFFFF;
4     float abs_x = *((float*)&int_x); // réinterprétation flottante
5     return abs_x;
6 }
```

**Solution 34 (Racine carrée inverse rapide (2)).** La première étape est de déterminer la projection de type à effectuer pour calculer les fonctions :

- $itof(x) = btof(btoi^{-1}(x))$
- $ftoi(x) = btoi_s(ftob(x))$

On utilise la même technique que pour l' **Exercice 33** :

```

1 | float fast_inverse_square_root(float x) {
2 |     int n = *((int*)&x);
3 |     // transformation
4 |     float y = *((float*)&n);
5 |     return y;
6 | }
```

On a  $A = 0x340003ce$  et  $B = 0xc2fe000f$  interprétés comme des nombres flottants, d'où  $-\frac{3B}{2A} = 0x4ebe7a62$  selon la norme *IEEE 754*. La difficulté est donc de trouver le mot binaire correspondant à la projection entière de  $-\frac{3B}{2A}$  :

```

1 | int binary = 0x4ebe7a62;
2 | float float_interpretation = *((float*)&binary);
3 | int projection = (int)(float_interpretation);
4 | printf("%x\n", projection);
```

donne le mot binaire  $0x5f3d3100$ . Finalement :

```

1 | float fast_inverse_square_root(float x) {
2 |     int n = *((int*)&x);
3 |     n = 0x5f3d3100 - (n >> 1);
4 |     float y = *((float*)&n);
5 |     return y;
6 | }
```

On obtient  $\frac{1}{\sqrt{0.01}} = 10 \approx 10.70$ . L'erreur est assez élevé. En effet, la projection entière de  $0x4ebe7a62$  comporte une imprécision supplémentaire qui s'ajoute à celle de  $\log_2$ .

On peut améliorer la précision en balayant les projections entières adjacentes. Avec  $0x5f31eb85$ <sup>4</sup>, on obtient une erreur de moins de 0.1%. Cette précision peut encore être améliorée en utilisant la [méthode de Newton](#).

## Interagir avec les flux standards

### Solution 35 (Distance Euclidienne).

**Code** : solutions/part2/chapter2/euclidian\_dist.c

### Solution 36 (Somme d'entiers).

**Code** : solutions/part2/chapter2/int\_sum.c

### Solution 37 (Jeu du plus ou moins).

**Code** : solutions/part2/chapter2/plus\_ou\_moins.c

La stratégie optimale est identique à celle de l' **Exercice 40** pour les mêmes raisons. Il faudra au maximum  $\lceil \log_2(N) \rceil$  tentatives pour trouver le nombre mystère.

4. Le jeu *Quake III* utilise la constante  $0x5f3759df$ , mais utilise la méthode Newton. La constante doit aussi s'adapter à ça pour de meilleurs résultats.

## Tableaux statiques

**Solution 38 (Les routines, direction la seconde classe!).** Une fonction ne peut pas être construite pendant l'exécution du programme en C<sup>5</sup>.

**Solution 39 (Routines classiques de manipulation de tableaux).**

**Code :** solutions/part2/chapter2/tab\_select\_sort.c

On pose pour  $i \in \llbracket 0; l(T) \rrbracket$  l'hypothèse de récurrence  $P(i)$  : « après le  $i^e$  tour de boucle, on a les invariants suivants :

- $T[0 : i]$  est trié dans l'ordre croissant
- $T[0 : i]$  contient les  $i$  plus petits éléments de  $T$ .

»

Initialisation : le tableau  $T[0 : 0]$  est vide. Les deux propositions constitutives de  $P(0)$  sont immédiates car le tableau est vide.

Hérédité : Soit  $i \in \llbracket 0; l(T) - 1 \rrbracket$ . Supposons  $P(i)$ . Au  $(i + 1)^e$  tour de boucle, on a  $i_{min}$  l'indice de l'élément minimal dans  $T[i : l(A)]$ . Par HR, tous les éléments de  $T[0 : i]$  sont inférieurs à ceux de  $T[i : l(A)]$  et donc à  $T[i + i_{min}]$ . Après le  $(i + 1)^e$  tour de boucle, on a échangé  $T[i]$  et  $T[i + i_{min}]$  donc  $T[0 : i + 1]$  contient des éléments tous inférieurs à ceux de  $T[i + 1 : l(A)]$  et  $T[i]$  majore  $T[0 : i + 1]$  qui est donc trié par ordre croissant.

Donc les deux propositions de  $P(i + 1)$  sont vraies.

Conclusion : L'hypothèse de récurrence est initialisée et héréditaire. Donc par principe de raisonnement par récurrence, pour tout  $i \in \llbracket 0; l(T) \rrbracket$ ,  $P(i)$  est vraie.

En particulier, pour  $i = l(T)$ , d'après le premier invariant, le tableau  $T[0 : l(T)] = T$  est trié à la fin de l'exécution de l'algorithme.

**Solution 40 (Recherche dichotomique).**

**Question 1 :** La démonstration est assez immédiate. Supposons que  $x \in T$ . Il existe donc  $i_x \in \llbracket 0; l(T) - 1 \rrbracket$  tel que  $T[i_x] = x$ .

- Supposons  $x \leq T[i]$ . Comme  $T$  est trié par ordre croissant,  $i_x \leq i$
- Supposons  $x > T[i]$ . Comme  $T$  est trié par ordre croissant,  $i_x > i$

**Question 2 :** À chaque tour de boucle, on pose  $i_m = \left\lfloor \frac{l(T)}{2} \right\rfloor$ , et alors :

- si  $l(T) = 0$ , renvoyer *Faux*
- si  $l(T) = 1$  et  $T[0] = x$ , renvoyer *Vrai*
- si  $x \leq T[i_m]$ ,  $T \leftarrow T[0; i_m]$  et boucler
- si  $x > T[i_m]$ ,  $T \leftarrow T[i_m : l(T)]$  et boucler

Grâce au choix de  $i_m$  effectué, on divise par deux la taille du tableau. Après au maximum  $\lceil \log_2(l(T)) \rceil$  étapes, on a  $l(T) \leq 1$  et le programme termine.

```

1  #include <stdio.h>
2  #include <stdlib.h>
3
4  char is_in(int x, int array[], unsigned int length) {
5      unsigned int i_left = 0;
```

5. Enfin, en assembleur on peut, mais là n'est pas le sujet, ce n'est plus du C.

```

6   unsigned int i_right = length;
7   unsigned int i_mid = i_left + (i_right - i_left) / 2;
8   while (i_mid > i_left) {
9       if (array[i_mid] > x) { // x à gauche
10          i_right = i_mid;
11      } else if (array[i_mid] < x) {
12          i_left = i_mid;
13      } else {
14          return 1;
15      }
16      i_mid = i_left + (i_right - i_left) / 2;
17  }
18  return array[i_mid] == x; // traite i_mid == i_left
19  }

```

### Solution 41 (Liste des nombres premiers).

Dans la suite, on note<sup>6</sup> :

- $O(f) = \{g : \mathbb{N} \rightarrow \mathbb{R} \mid \exists c \in \mathbb{R}_+, g \leq cf\}$ .  
C'est l'ensemble des fonctions dominées linéairement par  $f$
- $\Omega(f) = \{g : \mathbb{N} \rightarrow \mathbb{R} \mid \exists c \in \mathbb{R}_+, f \leq cg\}$ .  
C'est l'ensemble des fonctions qui dominent linéairement  $f$ .
- $\Theta(f) = O(f) \cap \Omega(f)$  est l'ensemble des fonctions linéairement équivalentes à  $f$

On écrit  $a(n) \approx b(n)$ , ou  $a(n) \in \Theta(b(n))$ , de manière équivalente à la phrase  $\exists C_1, C_2 \in \mathbb{R}_+, \forall n \in \mathbb{R}, C_1 b(n) \leq a(n) \leq C_2 b(n)$ .

On utilise les résultats suivants découlant du théorème des nombres premiers :

- $\pi(i) \approx \frac{i}{\ln(i)}$
- $p_i \approx i \cdot \ln(i)$ , où  $p_i$  désigne le  $i^e$  nombre premier

### Questions 1 et 2 :

La structure du code est la même dans les deux cas :

```

1   #include <stdio.h>
2   #include <stdlib.h>
3
4   #define N ...
5
6   char primal_test(unsigned int n, unsigned int primes[], unsigned int counter);
7
8   int main() {
9       unsigned int primes[N];
10      unsigned int i = 3;
11      unsigned int counter = 0;
12      primes[0] = 2;
13      while (i <= N) {
14          if (primal_test(i, primes, counter)) {
15              primes[++counter] = i;
16              printf("%u;", primes[counter]);

```

6. Voir section 8.2 pour plus de détails. Ces notations ne sont pas *nécessaires* pour la démonstration. Elles allègent l'écriture, c'est tout.



```

17     }
18     i += 2;
19 }
20 unsigned nb_primes = counter + 1; // + 1 car 2 est aussi premier
21 return EXIT_SUCCESS;
22 }

```

Si on note pour  $3 \leq i \leq N$ ,  $c_{test}(i)$  le nombre de tour de boucle effectué par *primal\_test*(*i*), le cout total  $C(N)$  est de :

$$C(N) = \sum_{i=3}^N c_{test}(i)$$

Les paramètres *primes* et *counter* sont inutiles pour la première question :

```

1 char primal_test(unsigned int n, unsigned int primes[], unsigned int counter) {
2     char is_primal = n == 2 || (n > 2 && n % 2 != 0); // nécessaire
3     for (int i = 3; i*i <= n && is_primal; i += 2) {
4         is_primal = (n % i == 0) ? 0 : is_primal;
5     }
6     return is_primal;
7 }

```

On a  $c_{test}(i) = 0.5\sqrt{i} - 3$ , d'où  $C(N) = O(N^{\frac{3}{2}})$ .

Pour la seconde :

```

1 char primal_test(unsigned int n, unsigned int primes[], unsigned int counter) {
2     char is_primal = 1; // premier par défaut
3     for (unsigned int k = 0; k <= counter && is_primal && primes[k] * primes[k] <= n; k++) {
4         is_primal = (n % primes[k] == 0 ? 0 : is_primal);
5     }
6     return is_primal;
7 }

```

Si *n* est premier, il faut tester tous les premiers inférieurs à  $\sqrt{n}$ . Le test fait donc au plus  $\pi(\sqrt{n})$ .

### Question 3 :

L'algorithme décrit ne suit pas la même structure. Cette structure semble seulement plus complexe à écrire pour aboutir au même résultat. On montre à la dernière question que tel n'est pas le cas et que le crible d'Eratosthène est plus rapide que l'algorithme "naïf" précédent.

```

1 #include <stdio.h>
2 #include <stdlib.h>
3
4 #define N ...
5
6 int ceiling_divide(int a, int b) {
7     return a/b + (a % b != 0);
8 }
9

```

```

10 int main() {
11     unsigned int nb_primes = 0;
12     unsigned int P[N];
13     P[0] = 0;
14     P[1] = 0;
15     for (unsigned int i = 2; i < N; i++) {
16         P[i] = 1;
17     }
18     unsigned int p = 2;
19     while (p*p <= N) {
20         for (unsigned int f = p; f < ceiling_divide(N, p); f++) {
21             P[p*f] = 0;
22         }
23         do {
24             p++;
25         } while (p < N && !P[p]);
26     }
27     for (unsigned int i = 0; i < N; i++) {
28         nb_primes += P[i];
29     }
30     printf("Nb primes : %u\n", nb_primes);
31     return EXIT_SUCCESS;
32 }

```

#### Question 4 :

##### Analyse du crible d’Eratosthène :

La boucle *for* fait  $\left\lceil \frac{N}{p_i} \right\rceil - p_i$  tours de boucle et la boucle *do-while* en fait  $p_{i+1} - p_i$  puisqu’elle part de  $p_i$  pour s’arrêter au premier juste suivant, c’est-à-dire  $p_{i+1}$ .

D’où :

$$C_e(N) = \sum_{i=1}^{\pi(\sqrt{N})} \left\lceil \frac{N}{p_i} \right\rceil + p_{i+1} - 2p_i$$

Par ailleurs :

$$\begin{aligned}
 p_{i+1} - 2p_i &\approx (i+1)\ln(i+1) - 2i\ln(i) \\
 &= i(\ln(1 + \frac{1}{i})) + \ln(i+1) - i\ln(i) \\
 &\leq i\ln(2) + \ln(i+1) - i\ln(i) \\
 &\leq 0
 \end{aligned}
 \quad \text{pour } i \geq 4$$

D’où :

$$\begin{aligned}
 C_e(N) &\approx \sum_{i=4}^{\pi(\sqrt{N})} \left\lceil \frac{N}{p_i} \right\rceil + p_{i+1} - 2p_i \\
 &\leq \sum_{i=4}^{\pi(\sqrt{N})} \frac{N}{p_i}
 \end{aligned}$$

On conclut une comparaison série-intégrale :

$$\begin{aligned}
 C_e(N) &\approx N \sum_{i=4}^{2\frac{\sqrt{N}}{\ln(N)}} \frac{1}{i \cdot \ln(i)} \\
 &\leq N \int_3^{2\frac{\sqrt{N}}{\ln(N)}} \frac{1}{x \cdot \ln(x)} dx \\
 &\approx N \cdot \ln \left( \frac{2\sqrt{N}}{\ln(N)} \right) \\
 C_e(N) &\in O(N \cdot \ln(\ln(N)))
 \end{aligned}$$

### Analyse de l'algorithme par test de primalité :

On cherche d'abord un majorant, pour se donner une idée du nombre d'opérations effectués par l'algorithme. On a environ  $\frac{N}{p_i}$  multiples de  $p_i$  inférieurs à  $N$ . Pour être certain de disqualifier chacun de ces multiples, il faut vérifier sa divisibilité par les  $i$  premiers nombres premiers. Ainsi, les multiples de  $p_1 = 2$  doivent être disqualifiés par une division par 2. Les multiples de  $p_2 = 3$  doivent être disqualifiés par une division par 2 et une division par 3, etc... On compte clairement trop d'opérations, puisque certains multiples de  $p_i$  sont aussi multiples de  $p_j < p_i$  et nécessiterons moins de  $i$  opérations pour être disqualifiés.

Cela donne :

$$\begin{aligned}
 C_{pt}(N) &\leq \sum_{i=1}^{\pi(\sqrt{N})} i \frac{N}{p_i} \\
 &\approx \frac{N}{2} + 2\frac{N}{3} + N \sum_{i=3}^{2\frac{\sqrt{N}}{\ln(N)}} \frac{1}{\ln(i)} \\
 &\leq 7\frac{N}{6} + N \int_2^{2\frac{\sqrt{N}}{\ln(N)}} \frac{1}{\ln(x)} dx \quad \text{comparaison série-intégrale} \\
 &= 7\frac{N}{6} + N \times Ei \left( \ln \left( 2\frac{\sqrt{N}}{\ln(N)} \right) \right) \quad \text{Exponentielle intégrale en posant } u = \ln(x)
 \end{aligned}$$

On utilise le développement en série entière :

$$\begin{aligned}
 Ei(x) &= \gamma + \ln|x| + \sum_{k=1}^{+\infty} \frac{x^k}{k \cdot k!} \\
 &\leq \gamma + \ln|x| + e^x - 1
 \end{aligned}$$

Alors :

$$\begin{aligned}
 C_{pt}(N) &\leq \left( \frac{1}{6} + \gamma \right) N + N \left[ \ln \left| \ln \left( 2\frac{\sqrt{N}}{\ln(N)} \right) \right| + 2\frac{\sqrt{N}}{\ln(N)} \right] \\
 &\in O \left( \frac{N\sqrt{N}}{\ln(N)} \right)
 \end{aligned}$$

On cherche ensuite un minorant en supposant que l'algorithme de la **Question 2** ne teste que les

nombre premiers. Il s'agit d'une sous-estimation du nombre d'opérations effectués :

$$\begin{aligned}
 C_{pt}(N) &\geq \sum_{i=3}^{\pi(N)} \pi(\sqrt{p_i}) \\
 &\approx \sum_{i=3}^{\frac{N}{\ln(N)}} \frac{\sqrt{i \cdot \ln(i)}}{\ln(\sqrt{i \cdot \ln(i)})} \\
 &\approx 2 \int_3^{\frac{N}{\ln(N)}} \frac{\sqrt{i \cdot \ln(i)}}{\ln(i \cdot \ln(i))} di \quad \text{par comparaison série-intégrale}
 \end{aligned}$$

On ne va pas s'amuser à essayer d'intégrer cette merde<sup>7</sup>.

Par contre, on peut utiliser le résultat de majoration. On a :

$$\left( \frac{x\sqrt{x}}{\ln(x)} \right)' = \frac{\sqrt{x}(\frac{3}{2}\ln(x) - 1)}{\ln^2(x)}$$

On va donc comparer  $A(x) = \frac{\sqrt{x \cdot \ln(x)}}{\ln(x \cdot \ln(x))}$  et  $B(x) = \frac{\sqrt{x}(\frac{3}{2}\ln(x) - 1)}{\ln^2(x)}$  :

$$\frac{A(x)}{B(x)} = \frac{\ln^{2.5}(x)}{(\ln(x) + \ln(\ln(x)))\left(\frac{3}{2}\ln(x) - 1\right)}$$

On a  $\frac{A(x)}{B(x)} \xrightarrow{x \rightarrow +\infty} +\infty$ . En particulier, il existe  $r \in \mathbb{R}_+^*$  tel que pour tout  $x \geq r$ ,  $A(x) \geq B(x)$ .  $A$  et  $B$  sont positifs. Donc :

$$\begin{aligned}
 \int_r^{\frac{N}{\ln(N)}} A(x) dx &\geq \int_r^{\frac{N}{\ln(N)}} B(x) dx \\
 &= \frac{\frac{N}{\ln(N)} \sqrt{\frac{N}{\ln(N)}}}{\ln\left(\frac{N}{\ln(N)}\right)} + C \\
 &\in \Omega\left(\frac{N\sqrt{N}}{\ln^{1.5}(N)\ln\left(\frac{N}{\ln(N)}\right)}\right)
 \end{aligned}$$

L'algorithme de la **Question 2** est donc asymptotiquement plus lent que le crible d'Eratosthène<sup>8</sup>

## Tableaux dynamiques

**Solution 42 (Conversion en binaire).**

**Code :** solutions/part2/chapter2/nat2bin.c

## Tableaux multidimensionnels

**Solution 43 (Afficher un tableau 2D).**

**Code :** solutions/part2/chapter2/tab\_display\_2d.c

**Solution 44 (Affichage du triangle).**

Version généralisée pour `N_LINES` quelconque.

<sup>7</sup>. Btw j'ai essayé... ça s'est pas bien passé...

<sup>8</sup>. Le résultat est moins précis que celui de M. O'Neill, professeur au département d'informatique de l'université de Harvey Mudd, qui trouve que l'algorithme de la **Question 2** est en  $\Theta\left(\frac{N\sqrt{N}}{(\ln(N))^2}\right)$ .

**Code :** solutions/part2/chapter2/triangle.c

#### Solution 45 (Matrices (1)).

Soit  $m$  une matrice implantée en C. Il faut d'abord déterminer ce que signifie  $mat[i] \equiv ((char*)m + i \times sizeof(double*))$ . Le  $double*$  est le type d'une ligne. Donc  $l_i = mat[i]$  est le pointeur vers la  $i^e$  ligne. On a alors  $l_i[j]$  le  $j^e$  élément de la  $i^e$  ligne. En remplaçant  $l_i$  par son expression, on a :

$$mat[i][j] \equiv *((mat + i) + j) \equiv (((char*)mat + i \times sizeof(double*)) + j \times sizeof(double))$$

**Code :** solutions/part2/chapter2/matrix.c

### Structures

#### Solution 46 (Matrices (2)).

Il suffit d'utiliser l'interface suivante, les modifications des routines sont assez triviales :

```

1  #ifndef MATRIX_H_INCLUDED
2  #define MATRIX_H_INCLUDED
3
4  struct Matrix {
5      double** mat;
6      unsigned int n;
7      unsigned int m;
8  };
9
10 struct Matrix matrix_new(unsigned int n, unsigned int m);
11 void matrix_destroy(struct Matrix m);
12 void matrix_display(struct Matrix m);
13 struct Matrix matrix_mul(struct Matrix a, struct Matrix b);
14
15 #endif

```

**Solution 47 (Optimisation d'alignement).** Avec un double à la place d'un int :

- `sizeof(struct ExampleUnaligned) = 24`
- `sizeof(struct ExampleAligned) = 16`

#### Solution 48 (Listes chaînées).

**Code :** solutions/part2/chapter2/linkedlist\_1.c

### Modulation et entêtes

#### Solution 49 (Un module de listes chaînées).

On écrit le code des fonctions de l'exercice précédent dans un fichier "*linkedlist.c*"<sup>9</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>

```

9. Dans le répertoire de travail du fichier "*main.c*"

```

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 *tail;
18     unsigned int length;
19 };
20
21 struct LinkedList* linkedlist_new();
22 void linkedlist_destroy(struct LinkedList* l);
23 char linkedlist_error_access(struct LinkedList *l);
24 char linkedlist_is_empty(struct LinkedList* l);
25 void linkedlist_push_on_head(struct LinkedList *l, int value);
26 void linkedlist_push_on_tail(struct LinkedList *l, int value);
27 int linkedlist_pop_from_head(struct LinkedList *l);
28 int linkedlist_pop_from_tail(struct LinkedList *l);
29 void linkedlist_display(struct LinkedList *l);
30
31 #endif

```

Il conserve la même fonction main du fichier “main.c” avec une inclusion du module :

```

1  // stdio est non nécessaire ici
2  #include <stdlib.h> // juste pour 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_tail(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>

```

**Solution 50 (Calculatrice).**

**Code :** solutions/part2/chapter2/calculatrice.c

**Solution 51 (Atoi).**

**Code :** solutions/part2/chapter2/atoi.c

## Interagir avec les flux de fichiers

### 12.2.2 Concepts avancés

#### Passage d'arguments au programme

**Solution 52 (Liste des arguments).**

**Code :** solutions/part2/chapter3/list\_args.c

**Solution 53 (Un cat minimaliste).**

**Code :** solutions/part2/chapter3/cat.c

#### Pointeurs de routines

**Solution 54 (Mapping).**

**Code :** solutions/part2/chapter3/mapping.c

#### Directives du préprocesseur (2)

**Solution 55 (Tout dépend du système).**

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  int main() {
5      #if defined(__linux__)
6      printf("Chouette !\n");
7      #elif defined(_WIN32)
8      printf("Beeerk !\n");
9      #endif
10     return EXIT_SUCCESS;
11 }
```

**Solution 56 (Libération sécurisée).**

```
#define safe_free(p) do {free((void*)p);p = NULL;} while(0)
```

#### Routines variadiques

**Solution 57 (À un doigt du zéro).**

**Code :** solutions/part2/chapter3/un\_doigt\_du\_zero.c

**Solution 58 (Fonctions à paramètres par défaut).**

**Code :** solutions/part2/chapter3/default\_parameters.c



## TROISIÈME PARTIE



### 12.3.1 Types abstraits

**Solution 59 (Tableaux dynamiques).**

On considère la signature de la fonction équivalente de la procédure de redimensionnement :

$$\text{redimensionner}(\text{Tableau}, \text{Entier}) \rightarrow \text{Tableau}$$

On étend la spécification par les équations suivantes, définies pour tout tableau  $t$ , et pour tous entiers  $i$  et  $n$  :

- $\text{taille}(\text{redimensionner}(t, n)) = n$
- $\text{taille}(t) \leq i < n \Rightarrow \text{init}(\text{redimensionner}(t, n), i) = \text{Faux}$  (ok même si  $n < \text{taille}(t)$ )
- $0 \leq i < \min(n, \text{taille}(t)) \Rightarrow \text{init}(\text{redimensionner}(t, n), i) = \text{init}(t, i)$
- $0 \leq i < \min(n, \text{taille}(t)) \wedge \text{init}(\text{redimensionner}(t, n), i) = \text{Vrai} \Rightarrow \text{lire}(\text{redimensionner}(t, n), i) = \text{lire}(t, i)$

**Code :** solutions/part3/chapter2/tableaux\_dynamiques

**Solution 60 (Entiers relatifs et rationnels).**

Malgré la formulation de la question, la difficulté n'est pas théorique mais technique. D'ailleurs, le code proposé pour l'implantation de  $(\mathbb{Z}, +, \times)$  est *très largement* optimisable.

Il est surtout fondamental de comprendre la différence entre le type abstrait algébrique  $(\mathbb{Z}, +, \times)$ , qui est implanté dans cet exercice, et les types abstraits  $\frac{\mathbb{Z}}{2^N\mathbb{Z}}$  où  $N$  est le nombre de bits maximal, qui sont les types donnés nativement par le langage C (`char`, `short int`, `int`, etc...).

Le type  $(\mathbb{Z}, +, \times)$  est beaucoup plus lourd à implanter en termes de performances car il est plus difficile d'avoir une accélération matérielle (au niveau du processeur) pour le gérer. Par ailleurs, la plupart des cas d'utilisations n'en ont pas l'utilité, ce qui justifie l'obligation de l'implanter logiciellement.

**Code :** solutions/part3/chapter2/adt\_entiers\_et\_rationnels

### 12.3.2 Listes

**Solution 61 (Listes virtuelles).**

**Code :** solutions/part3/chapter2/listes\_virtuelles

**Solution 62 (Listes par tableaux).** C'est l'implantation des listes de Python.

**Code :** solutions/part3/chapter2/listes\_tableaux



**Coûts en tête et queue de liste :**

Les coûts d'accès et de suppression sont visiblement fonctions de  $O(1)$ . Pour l'insertion en tête ou en queue de liste, c'est le redimensionnement du tableau qui pose problème. On note  $c(k)$  le coût de la  $k^e$  insertion. Initialement, la liste est de longueur 0. L'insertion de l'élément numéro 1 =  $2^0$  amène à doubler sa taille. Par récurrence, on obtient que l'insertion  $2^i$  dans la liste coûte  $2^{i+1}$  opérations tandis que les autres insertions coûtent toutes 1 opération à un facteur indépendant de  $k$  près (qui ne change pas le résultat final puisque la domination linéaire "écrase" le facteur). Autrement dit :

$$c(k) \approx \begin{cases} k + 1 & \text{si } k = 2^i \\ 1 & \text{sinon} \end{cases}$$

**Remarque :** On approxime  $2k \approx k + 1$  car les deux ne diffèrent qu'à une transformation linéaire indépendante de  $k$  près.

On observe que  $\sum_{k=2^i}^{2^{i+1}-1} c(k) = 2^{i+1} = \sum_{k=2^i}^{2^{i+1}-1} 2$ . Le coût est linéaire.

Soit  $n \in \mathbb{N}$ . On note  $e = n = \lfloor \log_2(n) \rfloor$ . On a  $n = 2^e + r$ , où  $0 \leq r < 2^e$ . Le coût des  $n$  premières insertions est finalement :

$$\begin{aligned} C(n) &= \sum_{i=0}^{e-1} \sum_{k=2^i}^{2^{i+1}-1} c(k) + r \\ &= \sum_{i=0}^{e-1} \sum_{k=2^i}^{2^{i+1}-1} 2 + r \\ &\in O(2^e + r) \\ &= O(n) \end{aligned}$$

Donc le coût amortie d'une seule insertion est fonction de  $O(1)$ .

**Coûts en milieu de liste :**

Contrairement à l'implantation par listes chaînées, l'accès à un élément quelconque est en  $O(1)$  puisqu'il s'agit d'un tableau. L'insertion et la suppression sont en  $O(n)$  *dans tous les cas* puisqu'il faut décaler tous les éléments suivants du tableau. Pour une liste chaînée, l'accès à un élément quelconque dans le cas général est en  $O(n)$ . Par contre, pour l'insertion et la suppression, on observe que bien que le coût soit  $O(n)$  en général, si on parcourt la liste pour d'autres raisons, on peut supprimer des éléments *au passage* en  $O(1)$ .

**Conclusion des Exercice 61 et Exercice 62**

Le point le plus remarquable des codes C des deux exercices tient à l'identité des fichiers *main.c* et de l'implantation de *list\_display*. Le fait d'implanter la signature de liste permet de ne se servir que d'elle et d'être absolument indépendant de l'implantation.

Du point de vue des coûts asymptotiques, on observe que l'implantation par tableau permet des accès plus rapide et l'implantation par chaînage permet des modifications plus rapides.

Du point de vue des constantes, le code des listes chaînées est plus complexe et demande certaines optimisations comme un noeud virtuel pour éviter trop de branchements conditionnels. Si on ne fait qu'ajouter en tête ou en queue de liste, l'implantation par tableau semble relativement correcte au défaut du surcoût de redimensionnement. Par contre, l'implantation par tableau évite trop d'allocations dynamiques et est optimal pour la quantité de mémoire utilisée *par élément*. Si la taille de la liste est bornée et proche par le dessous d'une puissance de 2, l'implantation par tableau est relativement optimale.

**Solution 63 (Axiomes algébriques).**

**Pile :** Toute pile est construite canoniquement sous la forme  $P = empiler(\dots(empiler(pile\_vide(), e_1)\dots), e_n)$ . On note  $P = [e_n, \dots, e_1]$  Donc il faut décrire les opérations selon les formes canoniques résultant de *pile\_vide* et *empiler* (sachant que certaines opérations comme *depiler(pile\_vide())* sont simplement impossibles).

Pour toute *Pile* :  $P$  quelconque et tout *Élément* :  $e$  quelconque, on pose :

- $est\_vide(pile\_vide()) = Vrai$
- $est\_vide(empiler(P, e)) = Faux$
- $sommet(empiler(P, e)) = e$
- $depiler(empiler(P, e)) = P$

**File :** Même remarque que pour les piles, la forme canonique d'une file est construite par compositions de *file\_vide* et d'*enfiler*. On utilise la même notation.

Pour toute *File* :  $F$  quelconque et tout *Élément* :  $e$  quelconque, on pose :

- $est\_vide(file\_vide()) = Vrai$
- $est\_vide(enfiler(F, e)) = Faux$
- $premier(enfiler(F, e)) = e$
- $dernier(enfiler(F, e)) = \begin{cases} dernier(F) & \text{si } \neg est\_vide(F) \\ e & \text{sinon} \end{cases}$
- $defiler(enfiler(F, e)) = \begin{cases} enfiler(defiler(F), e) & \text{si } \neg est\_vide(F) \\ file\_vide() & \text{sinon} \end{cases}$

**Solution 64 (Piles et files).**

On ne décrit que la construction d'une file *via* deux piles. La construction d'une pile *via* deux files est duale. Les analyses de correction et de complexité sont identiques.

**Construction de la structure :**

On veut définir une file comme un couple de deux piles  $P_e$  et  $P_s$  (avec  $e$  pour entrée et  $s$  pour sortie). On pose pour  $F = (P_e, P_s)$  :

- $file\_vide() = (pile\_vide(), pile\_vide())$
- $est\_vide(F) = est\_vide(P_e) \wedge est\_vide(P_s)$
- $premier(F) = sommet(P_e)$
- $enfiler((P_e, P_s), e) = (empiler(P_e, e), P_s)$

On définit alors une nouvelle opération spécifique à notre structure : *vider*( $P_e, P_s$ ), qui va dépiler chaque élément de  $P_e$  pour l'empiler sur  $P_s$  :

**Algorithme 27 : Vider**


---

**Entrées :** *Pile* :  $P_e$ , *Pile* :  $P_s$

```

1 tant que  $\neg est\_vide(P_e)$  faire
2    $P_s \leftarrow empiler(P_s, sommet(P_e));$ 
3    $P_e \leftarrow depiler(P_e);$ 
4 retourner  $(P_e, P_s);$ 

```

---

On montre par récurrence que pour tout éléments  $e_1, \dots, e_n, s_1, \dots, s_m$ , on a :

$$vider([e_n, \dots, e_1], [s_m, \dots, s_1]) = ([], [e_1, \dots, e_n, s_m, \dots, s_1])$$

On utilise cette opération pour construire les opérations *defiler* et *dernier* de  $F$  :

---

**Algorithme 28 : *dernier*( $F$ )**


---

**Entrées :** *File* :  $F = (P_e, P_s)$   
 1 **si** *est\_vide*( $P_s$ ) **alors**  
 2    $(P_e, P_s) \leftarrow \text{vider}(P_e, P_s)$  // On suppose évidemment que la file est non vide.  
 3 **retourner** *sommet*( $P_s$ )

---



---

**Algorithme 29 : *defiler*( $F$ )**


---

**Entrées :** *File* :  $F = (P_e, P_s)$   
 1 **si** *est\_vide*( $P_s$ ) **alors**  
 2    $(P_e, P_s) \leftarrow \text{vider}(P_e, P_s)$  // *idem*  
 3    $P_s \leftarrow \text{depiler}(P_s)$ ;  
 4 **sinon**  
 5    $P_s \leftarrow \text{depiler}(P_s)$ ;  
 6 **retourner**  $(P_e, P_s)$

---

**Correction :** il faut vérifier que les axiomes du type abstrait algébrique de *File* sont bien respectés par cette structure.

Les trois premiers axiomes sont immédiatement vérifiés :

- $\text{est\_vide}(\text{file\_vide}()) = \text{est\_vide}(\text{pile\_vide}()) \wedge \text{est\_vide}(\text{pile\_vide}()) = \text{Vrai}$
- $\text{est\_vide}(\text{enfiler}(F, e)) = \text{est\_vide}(\text{empiler}(P_e, e)) \wedge \text{est\_vide}(P_s) = \text{Faux}$
- $\text{premier}(\text{enfiler}(F, e)) = \text{sommet}(\text{empiler}(P_e, e)) = e$

Quatrième axiome : soit  $F = (P_e, P_s)$  un couple de deux piles.

- Si  $\neg \text{est\_vide}(P_s)$ , on a immédiatement  $\text{dernier}(\text{enfiler}((P_e, P_s), e)) = \text{dernier}((P_e, P_s))$  puisque  $P_s$  est inchangé et non vide et donc que l'algorithme *dernier* n'agit que sur  $P_s$ .
- Si  $\text{est\_vide}(P_s)$ , on peut écrire  $P_e = [e_n, \dots, e_1]$  (avec  $P_e = []$  si  $F = []$ ). Alors :

$$\begin{aligned} \text{dernier}(\text{enfiler}([e_n, \dots, e_1], [], e)) &= \text{dernier}([e, e_n, \dots, e_1], []) \\ &= \text{dernier}([], [e_1, \dots, e_n, e]) \quad \text{après vider car } P_s = [] \\ &= \text{sommet}([e_1, \dots, e_n, e]) \end{aligned}$$

$$\text{D'où } \text{dernier}(\text{enfiler}((P_e, []), e)) = \begin{cases} e & \text{si } \text{est\_vide}(P_e) \\ \text{dernier}(P_e) & \text{sinon} \end{cases}$$

Cinquième axiome : soit  $F = (P_e, P_s)$  un couple de deux piles.

- Si  $\neg \text{est\_vide}(P_s)$ , on a immédiatement  $\text{defiler}(\text{enfiler}(F, e)) = \text{enfiler}(\text{defiler}(F), e)$  puisque  $P_s$  est inchangé par enfilerment et non vide et donc que l'algorithme *defiler* n'agit que sur  $P_s$ .
- Si  $\text{est\_vide}(P_s)$ , on peut écrire  $P_e = [e_n, \dots, e_1]$  (avec  $P_e = []$  si  $F = []$ ). Alors :

$$\begin{aligned} \text{defiler}(\text{enfiler}([e_n, \dots, e_1], [], e)) &= \text{defiler}([e, e_n, \dots, e_1], []) \\ &= \text{defiler}([], [e_1, \dots, e_n, e]) \quad \text{après vider car } P_s = [] \\ &= ([], \text{depiler}([e_1, \dots, e_n, e])) \end{aligned}$$

Si  $\text{est\_vide}(P_e)$ , on a bien  $\text{defiler}(\text{enfiler}((P_e, []), e)) = \text{file\_vide}()$ . Dans l'autre cas, on a  $\text{defiler}(\text{enfiler}((P_e, []), e)) = ([], [e_2, \dots, e_n, e])$ , et on veut montrer que cet objet a exactement le même comportement que  $([e], [e_2, \dots, e_n])$ . C'est immédiat puisqu'il suffit de vérifier que les observables (*est\_vide*, *premier* et *dernier*) donnent les mêmes résultats.

**Complexité :** on suppose que les piles sont implantés comme des listes simplement chaînés (le double chaînage n'apporte rien ici).

Les trois opérations *est\_vide*, *premier* et *enfiler* sont immédiatement de complexité temporelle fonction de  $O(1)$ . Soit  $F = (P_e, P_s)$  une file<sup>10</sup>. On note  $|P_e|$  (resp.  $|P_s|$ ) le nombre d'éléments de  $P_e$  (resp.  $P_s$ ) selon la représentation canonique (c'est-à-dire la longueur de la liste chaînée). La fonction *vider* est de complexité temporelle  $O(|P_e|)$  puisqu'elle dépile chacun des éléments de  $P_e$ . Déterminer la complexité temporelle de *dernier* et *defiler* n'est pas évident puisqu'on ne sait pas si la fonction *vider* est appelée. Dans le pire des cas, *vider* est appelée et la complexité est linéaire selon  $P_e$ , donc selon la longueur de la file. Toutefois, on observe que chaque élément de la file sera *au pire* empilé et dépilé une unique fois de chacune des deux piles  $P_e$  et  $P_s$  (pour entrer et sortir). Ainsi, toute séquence de  $n$  fonctions de manipulation débutant sur une file *vide* nécessitera  $O(4n) = O(n)$  opérations élémentaires. Le coût *amorti* des opérations *defiler* et *dernier* est donc fonction de  $O(1)$ . La construction est donc équivalente dans la pratique à une implantation par une liste doublement chaînée.

**Solution 65 (Optimizing greater than abstraction).**

**Code :** solutions/part3/chapter2/listes\_xor

**Solution 66 (Fusion triée de listes).**

On utilise l'implantation des listes de l' **Exercice 61**

**Code :** solutions/part3/chapter2/fusion\_listes.c

La complexité temporelle est immédiatement fonction de  $O(n + m)$  (où  $n$  et  $m$  sont les longueurs des deux listes) puisqu'on effectue exactement  $n + m$  tours de boucle et que chaque tour est de complexité temporelle fonction de  $O(1)$

---

10. On a prouvé ci-dessus qu'un tel couple est bien une file lorsque manipulé par les opérations décrites.

# 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).
- [3] Thibaut BALABONSKI et al. *Informatique - MP2I/MPI - Cours et exercices corrigés*. Ellipses, 2022. ISBN : 9782340-070349.
- [4] D. BEAUQUIER, J. BERSTEL et Ph. CHRÉTIENNE. *Éléments d'algorithmique*. Fév. 2005. URL : <https://www-igm.univ-mlv.fr/~berstel/Elements/Elements.pdf>.
- [5] 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).
- [6] 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).
- [8] Stanislas DEHAENE. *La bosse des maths*. Odile Jacob, 2010. ISBN : 9782738125248.
- [9] C. FROIDEVAUX, M-C. GAUDEL et M. SORIA. *Types de données et algorithmes*. 1990. URL : [https://drive.google.com/file/d/1rGw2CLiD5fwHQF8r5SVdUyZ1SLnAXRHJ/view?usp=drive\\_link](https://drive.google.com/file/d/1rGw2CLiD5fwHQF8r5SVdUyZ1SLnAXRHJ/view?usp=drive_link).
- [11] Jean GOUBAULT-LARRECQ. *Algorithmique des graphes*. Jan. 2024. URL : [https://drive.google.com/file/d/1KV7sd00uU8qwL6TXlZjr-4cQ\\_WqWUikH/view?usp=drive\\_link](https://drive.google.com/file/d/1KV7sd00uU8qwL6TXlZjr-4cQ_WqWUikH/view?usp=drive_link).
- [12] 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).

- [13] 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).
- [14] Donald KNUTH. *The Art of Computer Programming*. Addison-Wesley, 2022. ISBN : 0201038013. URL : [https://drive.google.com/drive/folders/1n0Wh2rfAy49rzrFhdS300Z8hPma6WN8P?usp=drive\\_link](https://drive.google.com/drive/folders/1n0Wh2rfAy49rzrFhdS300Z8hPma6WN8P?usp=drive_link).
- [15] 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).
- [16] 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).
- [17] Kalle RUTANEN. *O-notation in Algorithm Analysis*. Mai 2022. URL : <https://arxiv.org/pdf/1309.3210>.
- [18] 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).
- [19] Alexander STEPANOV et Paul MCJONES. *Elements of Programming*. Pearson Education Inc., juin 2019. ISBN : 0578222141. URL : [https://drive.google.com/file/d/1LNMhVp\\_q9z5KnmLCC05Q1eo3rQ170S1S/view?usp=drive\\_link](https://drive.google.com/file/d/1LNMhVp_q9z5KnmLCC05Q1eo3rQ170S1S/view?usp=drive_link).



## MANUELS ET DOCUMENTATION



- [7] *C11 Specification*. Avr. 2011. 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*. Fév. 2023. 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



- [2] Sean Eron ANDERSON. *Bit Twiddling Hacks*. Mai 2005. URL : <http://graphics.stanford.edu/~seander/bithacks.html>.
- [20] Tyler WHITNEY et al. *Documentation sur le langage C*. 2023. URL : <https://learn.microsoft.com/fr-fr/cpp/c-language/>.