B. Garnier

Informatique

Une introduction à la programmation

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

MINES Saint-Étienne

Institut Mines-Télécom

27 février 2025

Préface

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

Section à sauter sans lire

Merci à Basile Tonlorenzi pour la relecture, les idées et tout et tout ! ∧∧

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 êtres humains dérange, prière d'envoyer un mail avec argumentation. Je réécrirais en conséquences 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 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 – avec raison – pourquoi écrire un nouveau livre de programmation quand tant d'autres existent déjà. La première raison, et la principale, est celle de la langue. 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. La seconde raison tient au fait que 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 ou tout du moins formelles 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 qui ne réintroduise pas l'intégralité des mathématiques de classes préparatoires, tout en conservant une certaine rigueur.

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 par soi-même les ressources, bibliographies ou webographiques, nécessaires à son apprentissage. Beaucoup d'enseignants en école d'ingénieurs utilisent ce fait comme argument pour bâcler leur cours. *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.

Un objectif est aussi d'apporter une préparation pour les examens de programmation en C de l'ISMIN, avec une certaine quantité d'exercices corrigés. Les chapitres *Programmer un ordinateur*, *Les fondamentaux du langage* et *Les bases du langage* sont suffisants pour cela. Il n'est pas utile à ceux qui ne souhaitent que valider l'UP d'aller plus loin si tel n'est pas leur souhait.

Pour ceux sûrs de savoir déjà programmer en C, voir le deuxième programme de la section 5.13 pour le vérifier [1].

Une bibliographie est disponible en **Annexe** qui propose de manière thématique certains ouvrages spécialisés [2]. Cette bibliographie couvre les thèmes suivants :

- Représentation binaire des nombres [11][2]
- Programmation de manière générale [9][13] [3]
- Programmation en C [7][14][5]
- Programmation sous Linux [8]
- Compilation [6][10][12][1]
- Programmation système [4]
- Informatique théorique [3]

Toutes coquilles relevées [4], suggestions d'amélioration ou autres remarques quelconques peuvent être envoyées à la section "Formation" du serveur Discord de l'association Minitel de l'ISMIN.

1. Enfin, juste histoire de mettre l'égo de côté quoi...
2. 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 ;(
3. Pas très spécialisé me direz-vous, mais *The Art of Computer Programming* est si extraordinaire que je ne peux qu'en recommander vivement la lecture.
4. 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 ;)

Premier petit apparté : parenté

Originellement, ce livre devait, du moins dans mon esprit, être entièrement affilié à l'association étudiante d'informatique de l'école, Minitel. Je considérais, et considère encore, qu'il est mal placé de se mettre en avant dans le cadre d'un travail associatif, puisque l'on travaille pour les autres avant toute chose, pour soi en dernier.
Pourtant, c'est la conscience assez aigüe de mon ignorance de nombreuses choses en matière de programmation et d'informatique qui a changé radicalement l'application de ce point de vue vis-à-vis de ce livre. Cela provient de la certitude que mon travail pouvait être entaché de bévues d'importances assez grandes. Alors, à ne signer que du nom de l'association, les erreurs lui seraient attribuées. Certains vont alléguer que la relecture permet la correction de ces erreurs. Cela est vrai seulement dans le cas d'erreurs locales, purement techniques. Certaines erreurs sont d'un ordre plus large, et n'apparaissent pas clairement. Elles fondent la vision que l'on peut avoir de concepts généraux – comme l'informatique, la programmation et plus particulièrement la programmation en C. En fait, ces *erreurs* dont j'ai pu entrevoir les possibles existences par la lecture de certaines œuvres extraordinaires [13][9] ont parfois amenés à la réécriture – encore très imparfaite – d'immenses pans de ce livre. C'est parce-que je ne tiens pas à infliger la responsabilité d'autres inévitables erreurs à l'association Minitel que mon nom est écrit sur la couverture.
Par ailleurs, la qualité stylistique comme pédagogique varie d'une section à une autre, que ce soit dû à la fatigue ou au fait que certaines sections écrites d'abord souffrent d'un manque d'expérience dans la rédaction.

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 [5], avec l'objectif ultime qu'en abordant un exercice, le lecteur se dise "Ah ? Tiens donc, rigolo.".

On utilise l'échelle de mesure de difficulté utilisé par Donald Knuth [6][9] :

	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.

5. Pas toujours réussi hein !
6. Le G.O.A.T !

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.

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 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 [7].

Pour ceux qui voudraient pratiquer plus la programmation [8], la banque d'exercices de *Leetcode* est assez fournie (`https://leetcode.com/problemset/`). Elle nécessite pour beaucoup d'exercices les notions présentés dans les trois premières parties.

7. 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 une faute frappe (genre 13 au lieu de 31), faut me prévenir si tel est le cas :)

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

Table des matières

Préface . . . 1
Paragraphe à sauter sans lire . . . 1
Motivation . . . 1
Difficulté des exercices . . . 3

I Prolégomènes **11**
Introduction de l'introduction . . . 12

1 L'ordinateur et les données **13**
1.1 L'ordinateur . . . 13
1.1.1 Vue de haut . . . 13
1.1.2 À l'intérieur . . . 14
1.1.3 Données manipulées . . . 15
1.2 Opérations logiques . . . 17
1.2.1 Arité d'un opérateur, d'une fonction . . . 17
1.2.2 Opérations logiques et tables de vérité . . . 17
1.2.3 Opérations logiques élémentaires . . . 18
1.2.4 Opérations de décalage . . . 20
1.2.5 Exercices . . . 20
1.3 Opérations arithmétiques . . . 21
1.3.1 Représentation d'un nombre entier en binaire . . . 21
1.3.2 Nombres signés . . . 22
1.3.3 Addition . . . 23
1.3.4 Soustraction . . . 24
1.3.5 Décalage arithmétique à droite . . . 24
1.4 Opérations flottantes . . . 25
1.4.1 Représentation des nombres à virgules . . . 25
1.4.2 Écriture en base 2 de nombres décimaux . . . 26
1.4.3 Norme *IEEE 754* . . . 27
1.4.4 Notation pour la suite du cours . . . 29
1.4.5 Approximation du logarithme par la norme *IEEE 754* . . . 29
1.4.6 Exercices . . . 32
1.5 Représentation hexadécimale . . . 32
1.6 Données textuelles . . . 33
1.6.1 Caractères ASCII . . . 34
1.6.2 Caractères imprimables et non imprimables . . . 34

2 Programmer un ordinateur **36**
2.1 Algorithmes et programmation . . . 36
2.1.1 Algorithmes et programmes . . . 38
2.2 Un peu de vocabulaire . . . 38

2.3 Langages de programmation . . . 42
2.3.1 Objectifs des langages de programmation . . . 42
2.3.2 Langages compilés et interprétés . . . 42
2.4 Le langage C . . . 43
2.4.1 Installer un éditeur et un compilateur . . . 44
2.4.2 Compiler le premier programme . . . 45
2.4.3 Analyse du premier programme . . . 46

II Du langage C **49**
Introduction . . . 50

3 Fondamentaux du langage **52**
3.1 Identifiants . . . 52
3.2 Commentaires . . . 54
3.2.1 Commentaires sur une ligne . . . 55
3.2.2 Commentaires par bloc . . . 55
3.3 Lignes de code . . . 55
3.4 Le point-virgule . . . 55
3.5 Point d'entrée du programme . . . 56
3.6 Directives du préprocesseur (1) . . . 58

4 Bases du langage **59**
4.1 Variables . . . 59
4.1.1 Taille et type d'une variable . . . 62
4.1.2 Entiers signés et non signés . . . 63
4.1.3 Environnement et blocs . . . 64
4.1.4 Exercices . . . 65
4.2 Formatage de texte . . . 65
4.2.1 Formatage des caractères spéciaux . . . 66
4.2.2 Formatage de variables . . . 67
4.2.3 Exercices . . . 68
4.3 Opérateurs sur les variables . . . 68
4.3.1 Opérateurs arithmétiques . . . 70
4.3.2 Opérateurs bit-à-bit . . . 70
4.3.3 Opérateurs logiques, ou relationnels . . . 71
4.3.4 Opérateurs d'assignation . . . 72
4.3.5 Exercices . . . 74
4.4 Projection de type . . . 75
4.4.1 Exercices . . . 76
4.5 Structures de contrôle . . . 77
4.5.1 Structures élémentaires . . . 77
4.5.2 Structures complexes . . . 80
4.5.3 Détails sur l'instruction **break** (et l'instruction **continue**) . . . 86
4.5.4 Exercices . . . 87
4.6 Routines . . . 88
4.6.1 Signature et prototype . . . 89
4.6.2 Note relative à la copie des arguments . . . 90
4.6.3 La pile d'exécution . . . 92
4.6.4 Effet de bord . . . 94
4.6.5 Exercices . . . 94
4.7 Pointeurs . . . 94

4.7.1 *lvalue* et *rvalue* . . . 98
4.7.2 Formatage en chaîne de caractères . . . 99
4.7.3 Les pointeurs en paramètres de routines . . . 99
4.7.4 Arithmétique des pointeurs et projection de type . . . 100
4.7.5 Pointeurs itérés . . . 103
4.7.6 Exercices . . . 103
4.8 Interagir avec les flux standards . . . 104
4.8.1 Flux d'entrée standard . . . 105
4.8.2 Flux de sortie standard et flux d'erreur standard . . . 106
4.8.3 Exemple pratique d'utilisation de *stderr* . . . 107
4.8.4 Exercices . . . 107
4.9 Tableaux statiques . . . 109
4.9.1 Tableaux . . . 109
4.9.2 Définition . . . 111
4.9.3 Les tableaux statiques, kékoi pour de vrai ? . . . 112
4.9.4 Exercices . . . 116
4.10 Allocation dynamique . . . 119
4.10.1 Introduction au tas . . . 119
4.10.2 Les routines malloc et free . . . 120
4.10.3 L'importance de réinitialiser le pointeur après libération . . . 121
4.10.4 Exercices . . . 123
4.11 Tableaux multidimensionnels . . . 123
4.11.1 Tableaux multidimensionnels statiques . . . 123
4.11.2 Tableaux multidimensionnels dynamiques . . . 124
4.11.3 Exercices . . . 125
4.12 Structures . . . 126
4.12.1 Tableaux de structures . . . 129
4.12.2 Initialisation à la déclaration . . . 131
4.12.3 Exercices . . . 132
4.13 Modulation et entêtes . . . 134
4.13.1 Fichier d'entête . . . 134
4.13.2 Fichier de code source . . . 135
4.13.3 Un exemple simple . . . 135
4.13.4 Compilation modulaire . . . 137
4.13.5 Problème des chaînes d'inclusions . . . 140
4.13.6 Garde-fous . . . 142
4.13.7 Exercices . . . 143
4.14 Chaînes de caractères . . . 143
4.14.1 Chaîne de caractères . . . 143
4.14.2 En langage C . . . 143
4.14.3 Se faciliter la vie avec les chaînes littérales . . . 144
4.14.4 Exercices . . . 145
4.15 Les flux de fichiers . . . 146
4.15.1 Droits des fichiers . . . 146
4.15.2 Les flux de fichiers en C . . . 148
4.15.3 Rediriger un flux standard vers un fichier . . . 150
4.15.4 Exercices . . . 151

5 Concepts avancés **152**
5.1 Virgule et expressions . . . 152
5.1.1 La virgule comme opérateur . . . 152
5.1.2 La virgule comme séparateur . . . 153

5.1.3 Conclusion . . . 154
5.2 Paramètres d'un programme . . . 154
5.2.1 Le (véritable ?) prototype de *main* . . . 154
5.2.2 Exercices . . . 155
5.3 Unions . . . 156
5.3.1 Motivation par un exemple . . . 156
5.3.2 Principe et syntaxe . . . 156
5.3.3 Exercices . . . 158
5.4 Champs de bits . . . 158
5.4.1 Motivation et principe . . . 158
5.4.2 Syntaxe . . . 159
5.4.3 Le revers de la médaille . . . 160
5.5 Classes de stockage . . . 161
5.5.1 Quelques détails de la structure d'un programme en mémoire . . . 162
5.5.2 Choisir la classe de stockage d'une variable en C . . . 163
5.6 const et restrict . . . 166
5.6.1 const . . . 166
5.6.2 restrict . . . 170
5.7 Construction de littéraux . . . 171
5.7.1 Motivation . . . 171
5.7.2 Syntaxe . . . 171
5.7.3 Cas d'utilisation . . . 172
5.8 Pointeurs de routines . . . 173
5.8.1 Motivation . . . 173
5.8.2 Syntaxe . . . 174
5.8.3 Tableaux de routines . . . 175
5.8.4 Exercices . . . 176
5.9 Directives du préprocesseur (2) . . . 176
5.9.1 Inclusion de fichiers externes . . . 177
5.9.2 Compilation conditionnelle . . . 178
5.9.3 Diagnostiques . . . 179
5.9.4 Pragma . . . 179
5.9.5 Macros . . . 180
5.9.6 Opérateurs de macros sur les symboles . . . 182
5.9.7 Exercices . . . 184
5.10 Routines variadiques . . . 184
5.10.1 Motivation . . . 184
5.10.2 Appels de routines . . . 184
5.10.3 Syntaxe et module *stdarg* . . . 185
5.10.4 Macros variadiques . . . 187
5.10.5 Exercices . . . 187
5.11 Alignement . . . 187
5.12 Assembleur et C . . . 187
5.13 Tricks récréatifs (ft. Basile) . . . 187
5.13.1 Indexation inversée et boutisme . . . 188
5.13.2 Définition *K&R* de fonctions . . . 188
5.13.3 Digraphes et trigraphes . . . 188
5.13.4 Un peu d'imagination . . . 189

III Petite parenthèse théorique **192**
Introduction . . . 193

IV Le vrai monde de la réalité réelle **194**
Introduction . . . 195

6 Outils à la programmation **197**
6.1 Makefiles . . . 197
6.1.1 Généralités . . . 197
6.1.2 Principe et premiers exemples . . . 197
6.1.3 Personnaliser la production grâce aux variables . . . 199
6.1.4 Variables automatiques . . . 200
6.1.5 Réduire le nombre de règles avec la *stem* . . . 200
6.1.6 Les caractères génériques . . . 201
6.1.7 Substitution de chaînes . . . 202
6.1.8 Les règles virtuelles . . . 202
6.1.9 Idées pour aller plus loin . . . 203
6.2 Utiliser un débogueur . . . 203
6.3 Git . . . 204
6.4 Le markdown . . . 204

7 Programmer proprement **205**
7.1 À propos du style . . . 206
7.2 La gestion des erreurs . . . 206
7.2.1 Un petit exemple pour prendre la température . . . 207
7.2.2 Codes d'erreur . . . 207
7.3 Tests unitaires . . . 208
7.4 Benchmarks . . . 208

V Annexes **209**

8 Correction des 54 exercices **210**
8.1 Première partie . . . 210
8.1.1 Opérations logiques sur les mots binaires . . . 210
8.1.2 Opérations arithmétiques . . . 213
8.1.3 Opérations sur les nombres à virgules . . . 213
8.1.4 Représentation hexadécimale . . . 214
8.1.5 Caractères ASCII . . . 214
8.2 Deuxième partie . . . 214
8.2.1 Bases du langage . . . 214
8.2.2 Concepts avancés . . . 235

Bibliographie **238**

Liste des tableaux

1.1 Table de vérité de l'opérateur $\vee$ . . . 18
1.2 Table de vérité de l'opérateur $\oplus$ . . . 18

1.3 Table de vérité de l'opérateur ∧ . . . 19
1.4 Table de vérité de l'opérateur ¬ . . . 19
1.5 Table des caractères ASCII . . . 34

4.1 Priorités des opérateurs en C . . . 69
4.2 Opérateurs arithmétiques . . . 70
4.3 Opérateurs bit-à-bit . . . 71
4.4 Opérateurs logiques . . . 71

Liste des définitions

2.1.1 Algorithme . . . 36
2.1.2 Agent algorithmique . . . 37
2.1.3 Programme . . . 38
2.2.4 BIOS . . . 39
2.2.5 Système d'exploitation . . . 39
2.2.6 Fichier . . . 40
2.2.7 Extension de nom fichier . . . 40
2.2.8 Répertoire et système de fichiers . . . 40
2.2.9 Répertoire de travail . . . 41
2.2.10 Chemin d'accès . . . 41
2.2.11 Instruction . . . 41
2.2.12 Exécutable . . . 42
2.2.13 Niveau d'abstraction . . . 42
3.1.14 Identifiant . . . 52
4.1.15 Type . . . 60
4.1.16 Environnement global . . . 64
4.1.17 Environnement local . . . 64
4.3.18 Opérateurs et opérandes . . . 68
4.3.19 Vérité d'une expression . . . 71
4.5.20 Flot d'exécution/de contrôle . . . 77
4.6.21 Prototype . . . 89
4.6.22 Signature . . . 90
4.6.23 Argument . . . 91
4.8.24 Flux . . . 104
4.9.25 Structure de donnée . . . 110
4.13.26 Module et programmation modulaire . . . 134
5.5.27 Portée d'une variable et durée de vie . . . 162
5.6.28 Aliasing . . . 170
5.8.29 Fonction d'ordre supérieur . . . 174

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 précisement à la programmation.

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

- les composants fondamentaux d'un ordinateur classique
- les opérations logiques élémentaires sur les mots binaires
- les interpréations des mots binaires pour représenter des nombres entiers, des nombres à virgules et des caractères textuels
- une première approche simple de l'idée d'algorithme
- quelques premières définitions de vocabulaire technique relatif à l'informatique pratique
- l'installation des outils de base nécessaires à la programmation en langage C (Linux/Windows)

Pour cela, il est supposé que le/la lecteur/rice a effectué deux ou trois ans de classes préparatoires et se trouve à la lecture de ce document en entrée d'école d'ingénieur. Un certain bagage mathématique, considéré comme acquis, sera en effet nécessaire. Car ces bases "culturelles" comportent un peu de théorie [9].

À la fin de cette partie, le/la lecteur/rice possédera la culture minimale pour apprendre à programmer en comprenant ce qui est manipulé matériellement et logiquement, et aura une idée de ce que signifie programmer.

9. Notamment pour calculer les valeurs numériques associés à des mots binaires

CHAPITRE

1

L'ORDINATEUR ET LES DONNÉES

L'ordinateur

1.1.1 Vue de haut

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

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

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

Un ordinateur est appareil qui "calcule". La signification mathématique du terme "calculer" n'a rien d'évidente. Calculer est au départ une notion que l'on peut qualifier d'assez intuitive et il est donc difficile d'apporter une définition formelle claire qui satisfasse ce qu'un humain peut subjectivement appeler le "calcul".
La notion de calculabilité, qui définit ce qui est calculable et ce qui ne l'est pas, apporte une définition du calcul. Cette notion est fondé sur des modèles fondamentaux du calcul (tous équivalents, et heureusement !) comme les machines de Turing et le λ-calcul. Les ordinateurs contemporains sont fondés sur le modèle de la machine de Turing et sont dits équivalents au modèle de la machine de Turing.
Sans plus de détail [a] car cela serait tout à fait hors-sujet, la machine de Turing est basiquement une bande infinie de caractères munit d'une tête de lecture/écriture qui se déplace latéralement sur cette bande et agit selon la valeur des caractères lues grâce à une fonction de transition.

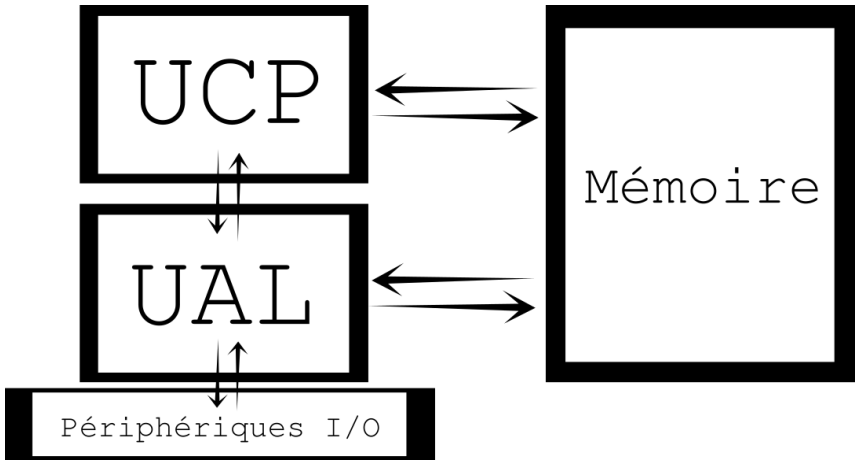
a. Pour plus de détail, voir le livre [3]

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

1.1.2 À l'intérieur

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

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



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

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

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

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

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

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

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

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

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

1.1.3 Données manipulées

Binaire

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

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

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

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

Octets

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

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

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

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

- $1\ Kio$ (kibioctet) $= 2^{10}\ o = 1024\ o$
- $1\ Mio$ (mebioctet) $= 2^{10}\ Kio = 1024\ Kio$
- $1\ Gio$ (gibioctet) $= 2^{10}\ Mio = 1024\ Mio$
- $1\ Tio$ (tebioctet) $= 2^{10}\ Gio = 1024\ Gio$
- $1\ Pio$ (pebioctet) $= 2^{10}\ Tio = 1024\ Tio$
- etc...

Petit apparté : octet comme unité d'information

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

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

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

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

Opérations logiques

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

1.2.1 Arité d'un opérateur, d'une fonction

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

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

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

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

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

1.2.2 Opérations logiques et tables de vérité

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

Les *tables de vérité* sont un moyen commode de visualiser les opérations logiques. Il s'agit de représenter toutes les sorties d'une fonction opératrice selon toutes les entrées possibles. Pour une fonction d'opérateur N-aire, on a 2^N possibilités d'entrées[4]. 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[5] est appelée définition *par compréhension*. Ce vocabulaire est emprunté à la théorie des ensembles puisqu'une fonction est un produit cartésien.

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

4. En effet, il n'y a que deux lettres dans l'alphabet $\{0, 1\}$.
5. Comme $f : x \mapsto x^2$ par exemple

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

On observe que le caractère *exponentiel* du nombre d'entrées et de sorties rend cette représentation inutilisable pour des fonctions d'arité grande.
En interprétant la lettre/symbole 0 par le nombre 0 et la lettre/symbole 1 par le nombre 1, on peut aussi décrire f par compréhension :

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

1.2.3 Opérations logiques élémentaires

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

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

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

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

OU logique inclusif

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

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

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

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

OU logique exclusif

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

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

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

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

ET logique

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

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

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

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

NON logique

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

A	$\neg A$
0	1
1	0

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

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

Extension aux mots

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

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

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

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

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

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

1.2.4 Opérations de décalage

Décalage à gauche

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

$$\begin{array}{rcccl} \ll & : & \mathcal{B}^N \times [\![0,N]\!] & \to & \mathcal{B}^N \\ & & ((a_{N-1}\dots a_0),i) & \mapsto & a_{N-1-i}\dots a_0 \underbrace{0\dots 0}_{i \text{ fois}} \quad \text{si } i<N \\ & & & \mapsto & 0\dots 0 \quad \text{si } i=N \end{array}$$

Décalage logique à droite

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

$$\begin{array}{rcccl} \gg_l & : & \mathcal{B}^N \times [\![0,N]\!] & \to & \mathcal{B}^N \\ & & ((a_{N-1}\dots a_0),i) & \mapsto & \underbrace{0\dots 0}_{i \text{ fois}} a_{N-1}\dots a_i \quad \text{si } i<N \\ & & & \mapsto & 0\dots 0 \quad \text{si } i=N \end{array}$$

1.2.5 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 ($+$, $-$, $*$, $/$, $|.|$, 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 :

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

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 :

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

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 définit est une arithmétique modulaire, c'est-à-dire que l'ensemble $[\![0; 2^N[\![$ est assimilé à l'ensemble des représentants des classes d'équivalence de $\frac{\mathbb{Z}}{2^N\mathbb{Z}}$.

1.3.1 Représentation d'un nombre entier en binaire

On considère la fonction suivante :

$$\begin{array}{rccc} btoi_u & : & \mathcal{B}^N & \to & [\![0; 2^N[\![\\ & & b & \mapsto & \sum_{i=0}^{N-1} (b_i 2^i) \end{array}$$

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 $[\![0; 2^N[\![$.
Sous la forme d'un polynôme de Horner évalué en 2 on a :

$$btoi_u(b) = b_0 + 2(b_1 + 2(b_2 + 2(\dots)\dots)) \tag{1.1}$$

On écrit plus simplement :

$$n = (b_{N-1}b_{N-2}\dots b_1 b_0)_2 = btoi_u(b)$$

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

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

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

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

où l'opération *modulo* noté % est définit 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 précisemment, $a \ \% \ b$ 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 $53 = 1 \times 2^5 + 1 \times 2^4 + 0 \times 2^3 + 1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0 = (110101)_2$

Exercice 7 (Du décimal au binaire) [15].
Trouver la représentation binaire des dix entiers naturels ci-dessous :

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

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

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

$$\begin{array}{rccl} *_{\mathbb{N}} & : & [\![0; 2^N[\![& \to & E \\ & & n & \mapsto & *(btoi^{-1}(n)) \end{array}$$

1.3.2 Nombres signés

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

$$(-2^{\dot{N}-1}), \ldots, \dot{-1}, \dot{0}, \dot{1}, \ldots, (2^{N-\dot{1}} - 1)$$

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

$$\begin{array}{rcccl} btoi_s & : & \mathcal{B}^N & \to & [\![2^{N-1}; 2^{N-1}[\![\\ & & b & \mapsto & \sum_{i=0}^{N-2}(b_i 2^i) - b_{N-1}2^{N-1} \end{array}$$

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

Proposition 1. $(\dot{btoi_s}(v)) = (\dot{btoi_u}(w)) \Leftrightarrow v = w$

Démonstration : $(\dot{btoi_s}(v)) \equiv \sum_{i=0}^{N-2}(b_i 2^i) - b_{N-1}2^{N-1} + b_{N-1}2^N[2^N]$
Donc $(\dot{btoi_s}(v)) = (\dot{btoi_u}(v))$, or la représentation en base 2 d'un entier naturel est unique, donc $v = w$.

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

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

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

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

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

Démonstration :

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

1.3.3 Addition

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

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

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 $2^{N-1} + 2^{N-1} \notin [\![0; 2^N[\![$, il faudrait un bit supplémentaire pour stocker l'information. Or ce bit n'est pas présent, il est donc simplement ignoré.

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

Démonstration :

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

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

Exemples : (pour $N = 8$)

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

1.3.4 Soustraction

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

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

1.3.5 Décalage arithmétique à droite

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

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

$$\begin{array}{rcccll} \gg_a & : & \mathcal{B}^N \times [\![0, N]\!] & \to & \mathcal{B}^N & \\ & & ((a_{N-1} \dots a_0), i) & \mapsto & \underbrace{a_{N-1} \dots a_{N-1}}_{i \text{ fois}} a_{N-1} \dots a_i & \text{si } i < N \\ & & & \mapsto & a_{N-1} \dots a_{N-1} & \text{si } i = N \end{array}$$

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

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

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

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

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

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

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

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

Opérations flottantes

1.4.1 Représentation des nombres à virgules

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

- π possède un nombre infini de chiffres dont l'information ne peut être stockée par une formule *calculable*[6]
- $\frac{1}{3}$ possède un nombre infini de chiffres après la virgule. Il suffit de stocker la partie entière 0 ainsi que la partie répétée des décimales, c'est-à-dire 3 (avec l'information de la répétition)

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

- 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 π).

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 [7] manipulent des mots binaires de taille fixe $N \in \{8, 16, 32, 64, 80, 128\}$

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

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

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

1.4.2 Écriture en base 2 de nombres décimaux

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

$$x = \lfloor x \rfloor + fract(x)$$

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

$$\lfloor x \rfloor = c_{n-1}10^{n-1} + c_{n-2}10^{n-2} + \cdots + c_0 10^0, \text{ où } n = \lceil log_{10}(\lfloor x \rfloor) \rceil$$

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

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

avec pour tout $i \in \mathbb{Z}$, $c_i \in [\![0; 9]\!]$

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

$$\begin{cases} \lfloor x \rfloor & = & c_{n-1}2^{n-1} + c_{n-2}2^{n-2} + \cdots + c_0 2^0 & \text{, où } n = \lceil log_2(\lfloor x \rfloor) \rceil \\ fract(x) & = & c_{-1}2^{-1} + c_{-2}2^{-2} + \ldots \end{cases}$$

avec pour tout $i \in \mathbb{Z}$, $c_i \in [\![0; 1]\!]$

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

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

7. Comme les processeurs *Intel x86*

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 1.4.3.1)

Exercice 8 (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.

1.4.3 Norme *IEEE 754*

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

Rappel de vocabulaire en notation scientifique

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

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

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

Par exemple :

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

Ici :

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

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

Il reste donc trois éléments, qui seront chacun stockés sur un certain nombre de bits.
Selon la norme *IEEE 754*, sur 32 bits :

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

Visuellement :

$$\underbrace{0}_{s}\ \underbrace{00000000}_{e}\ \underbrace{00000000000000000000000}_{m}$$

Proposition 8 (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.

Démonstration : 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).

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 $x = \pm \sum_{i=-\infty}^{N} \left(c_i 2^i\right)$ où N est minimal.

On a :

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

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.[8] 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 fixé (puisqu'on fixe le nombre de bits sur lesquels sera écrit l'exposant). On ajoute ce biais à la valeur N trouvé lors du calcul de la mantisse. Ainsi, pour $N \in [\![-2^{n-1} - 1; 2^{n-1} - 1]\!]$, on a $N + biais \geq 0$ 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 $N + biais$ pour le choix de la mantisse :

- $N + biais = 0$: le nombre est dit *"dénormalisé"*
- $N + biais \in [\![1; 2^n - 2]\!]$: 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 $x = (2^{-1}m) \times 2^{N+1}$

8. 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 réprésentant les nombres réels. C'est-à-dire qu'on peut indifférement comparer représentants ou représentés.

Une fois qu'on a déterminé la représentation binaire de $N + biais$ et les chiffres à conserver de la mantisse, il suffit de "coller" le bit de signe, la représentation binaire de $N + biais$ 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 $biais = 2^7 - 1 = 127$. Alors $N + biais = (biais + 1) + (N - 1) = 2^7 + 2^1 + 2^0 = (10000011)_2 > 0$ 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 $m = (01010010000000000000000)_2$

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

$$01000001101010010000000000000000$$

Exemple (dénormalisé) : Représentons le nombre $x = -9.1688559 \times 10^{-39}$ selon la norme *IEEE 754* sur 32 bits.
Il existe $p \in \mathbb{Z}$ tel que $2^p \leq x < 2^{p+1}$, c'est-à-dire $x = m \times 2^{\lfloor log_2(x) \rfloor}$.
On cherche alors $m = x \times 2^{-\lfloor log_2(x) \rfloor} = -9.1688559 \times 10^{-39} 2^{126} \approx 0.78 \approx (1.10001111010111000010100)_2 \times 2^{-1}$
On veut donc écrire selon la norme *IEEE 754* le nombre $x \approx 1.10001111010111000010100 \times 2^{-127}$

Sur 32 bits, $biais = 127$, donc $N + biais = 0$. Le nombre est donc dénormalisé[9]. On code alors :

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

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

$$10000000011000111101011100001010$$

1.4.4 Notation pour la suite du cours

On notera pour la suite $\mathbb{R}_{f32}$ (resp. $\mathbb{R}_{f64}$ et $\mathbb{R}_{f128}$) 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 a un vecteur $v \in \mathcal{B}^N$ associe le réel correspondant selon la norme *IEEE 754*.

1.4.5 Approximation du logarithme par la norme *IEEE 754*

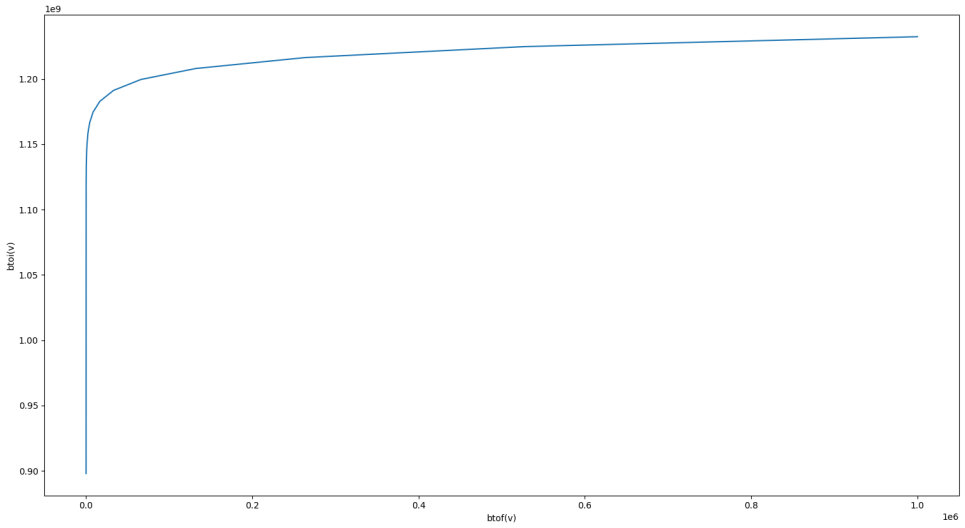
La norme *IEEE 754* présentée précédemment semble avoir été sortie du chapeau par un coup de baguette magique, comme si une telle norme était "évidente" pour résoudre les problèmes données en partie **1.4.1**. Cela n'est absolument pas évident, et sera essayé ici de présenter certaines conséquences du 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*.

9. Comme par hasard...

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

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

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



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 :

- $btof(v) = (-1)^s(m \times 2^{-23} + 1) \times 2^{e-127}$
- $btoi(v) = s \times 2^{31} + e \times 2^2 3 + m$

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

$$\begin{array}{rcll} log_2(btof(v)) & = & e - 127 + log_2\left(1 + 2^{-23}m\right) & \\ & \simeq & (e - 127) + \dfrac{1}{ln(2)}\left(2^{-23}m\right) & \text{car } ln(1+x) \sim_0 x \text{ avec } x = 2^{-23}m \approx 0 \end{array}$$

C'est-à-dire :

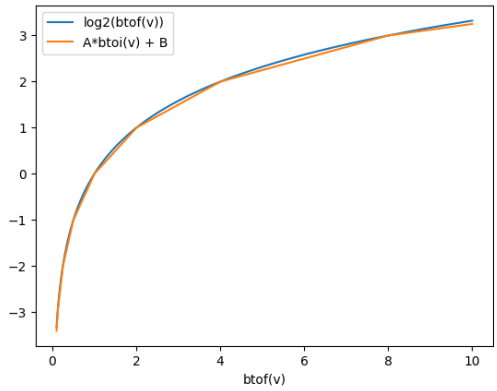
$$2^{23}log_2(btof(v)) + 2^{23}127 \simeq \frac{1}{ln(2)}m + 2^{23}e \approx btoi(v)$$

Remarque : l'approximation $\dfrac{1}{ln(2)} \approx 1$ 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 logarithmee base 2 sur les flottants :

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

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



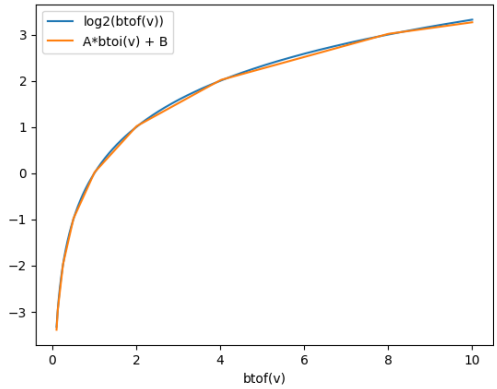
les courbes sont presque confondues mais une imprécision subsiste du fait des approximations effectués durant les calculs.

On peut toutefois obtenir une transformation linéaire beaucoup plus précise en utilisant les valeurs normalisés positives minimale et maximale pour calculer la régression linéaire [10].

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

- la pente de la régression linéaire $A = \dfrac{M - m}{btoi(ftob(M)) - btoi(ftob(m))}$ de représentation hexadécimale [11] 0x340003ce
- la constante à l'origine $B = log_2(m) - Am$ de représentation hexadécimale [12] 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

10. On a prouvé son existence comme approximation.
11. Voir 1.5 pour la représentation hexadécimale.
12. *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.

1.4.6 Exercices

Exercice 9 (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 seizes 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. [13] En effet, écrire en base 2 revient à décomposer selon les puissances de 2. En choisissant une puissance de 2 comme base d'écriture, on divise d'autant le nombre de chiffres requis.

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

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

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

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

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

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

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

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

Lien entre binaire et hexadécimale

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

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

Interprétation : Pour passer d'une écriture binaire à une écriture hexadécimale, il suffit d'écrire chaque mot[14] 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 : $0\mathsf{x2FA} = \underbrace{0010}_{2}\ \underbrace{1111}_{F}\ \underbrace{1010}_{A}$

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 10 (Conversion binaire-hexadécimale) [15].
Trouver les représentations binaires puis hexadécimale des nombres suivants. On utilisera l'écriture en base 2 pour les nombres entiers et la représentation 32 bits de la norme *IEEE 754* pour les nombres à virgules :

- 713705
- 8.8
- 42
- -1.1
- 101

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.

14. ou *mot*

1.6.1 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 FOF 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 1.5 – 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 : $(1100001)_2 = 0x61 = 97$. On peut également effectuer la correspondance dans l'autre sens, en réécrivant sous forme de caractères des mots binaires de 8 bits. Ainsi, 0x2B décrit le caractère ‘+’

1.6.2 Caractères imprimables et non imprimables

Certains caractères du tableau 1.5, 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 [15]

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

Exercice 11 (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 12 (Traduction ASCII 2) [10]. Écrire en hexadécimale le mot binaire qui décrit le texte suivant :

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

15. Pour les détails : 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 vieils normes de télécommunications (parfois encore usitées).

CHAPITRE

2

PROGRAMMER UN ORDINATEUR

Algorithmes et programmation

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

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

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

Définition 1 : Algorithme

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

Exemple : "Changer la valeur d'un nombre" n'est pas une instruction possible pour un algorithme, puisque le lecteur peut toujours *interpréter* l'instruction, et ne sait en fait toujours pas exactement ce

qu'il doit faire. Par contre l'instruction "Ajouter 1 à la valeur de la variable x" est une instruction, à condition que x soit connue, que 1 soit bien définie et que "Ajouter" soit une opération binaire bien définie sur 1 et x.

Tout simplement.

Algorithme 1 : Premier exemple simple

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

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

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

Définition 2 : Agent algorithmique

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

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

```
Bonjour !
Au revoir !
```

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

Algorithme 2 : 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 (c'est pas si con les singes...), et simplement lui ordonner d'exécuter cet algorithme par l'instruction "Va faire cuire les pâtes !".

Remarque : En petit malin, le lecteur attentif peut observer [1] 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

1. 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 :)

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).

2.1.1 Algorithmes et programmes

Définition 3 : Programme

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

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

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

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

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

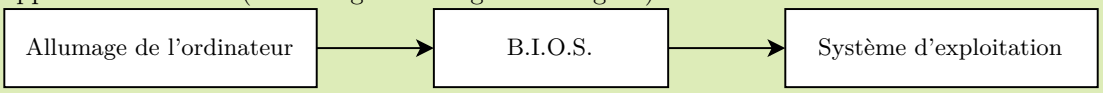
On peut cependant généraliser un peu la notion de *compilation* : on peut dire que compiler c'est lire une suite de caractères obéissant à une certaine syntaxe, en construisant une (autre) représentation de l'information que ces caractères expriment. De ce point de vue, beaucoup d'opérations apparaissent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

Un peu de vocabulaire

Cette table de vocabulaire est donné en préambule pour ne pas trop surcharger le texte ci-après et permettre que celui-ci conserve sa précision.

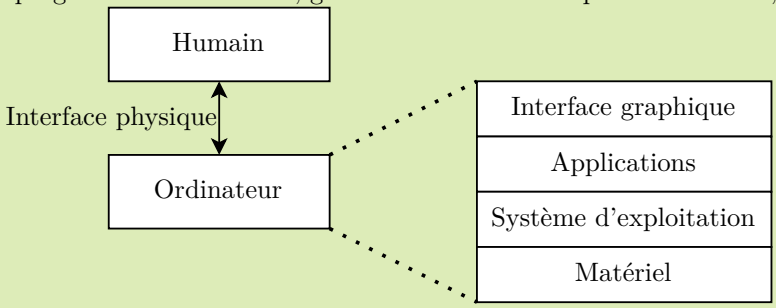
Définition 4 : BIOS

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).



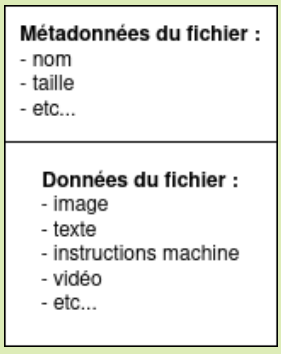
Définition 5 : Système d'exploitation

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



Définition 6 : 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 réunis sous un même nom et manipulé comme une unité, appelé le fichier. Les métadonnées d'un fichier fournissent des informations externes complémentaires.



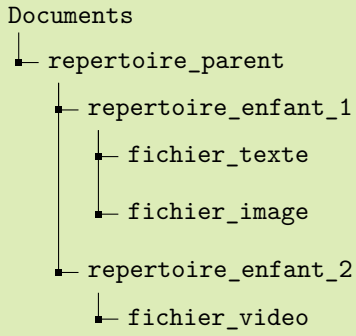
Définition 7 : Extension de nom fichier

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. [a]

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

Définition 8 : 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.



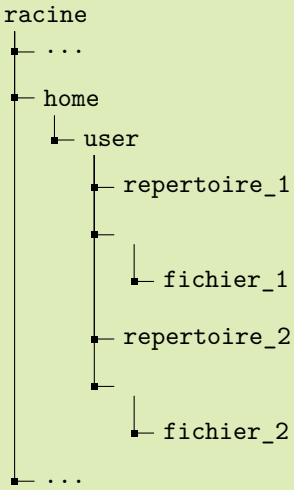
Définition 9 : Répertoire de travail

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 10 : 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



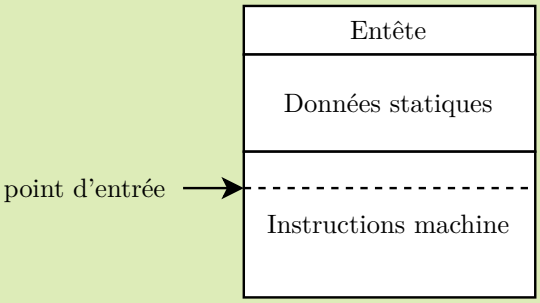
Ci-dessus, le chemin absolu du fichier 1 est *racine/home/user/repertoire_1/fichier_1*. Si on suppose que le répertoire de travail est *racine/home/user/repertoire_1*, alors le chemin relatif du fichier 2 est *../repertoire_2/fichier_2*.
En effet, "." désigne le nom du répertoire de travail, et ".." 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/*

Définition 11 : Instruction

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

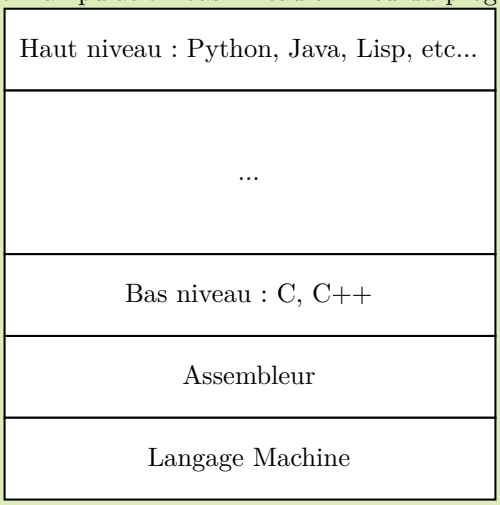
Définition 12 : Exécutable

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 débuter sa lecture des instructions.



Définition 13 : 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.



Langages de programmation

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

2.3.2 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 appellé *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. [2] Cela explique par exemple l'explosion de Python dans le monde du développement ces dix dernières années. [3]

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'apprendreà 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*[4]. dans l'objectif particulier de développer des systèmes d'exploitation. En

2. *Fun fact* : le langage interprété Lisp était très utilisé en intelligence artificielle au XX^{e} siècle. Comme les ordinateurs généralistes de l'époque étaient trop lents, des ordinateurs spécialisés pour interpréter ce langages ont été développés : les machines Lisp.

3. 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...

4. La médaille Richard-Hamming est décernée chaque année depuis 1988 par l'IEEE, pour honorer les contributions

effet, son rapprochement avec la machine, son extrême modularité, sa syntaxe claire et précise [5] 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 embarquées, 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 [6], le langage C a tendance à être remplacé pour le développement d'applications complexes par d'autres langages qui proposent certaines solutions abstraites [7], tiennent plus la main au programmeur en terme de sécurité [8] et/ou de stabilité de développement [9], proposent plus de fonctionnalités pré-écrites [10], etc...

2.4.1 Installer un éditeur et un compilateur

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

Éditeur de texte

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

Un premier programme

```c
#include <stdio.h>

int main(int argc, char **argv) {
  printf("Hello World !\n");
  return 0;
}
```

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/.

exceptionnelles à l'informatique et aux technologies de l'information.

5. 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 chuuut !
6. Qui se trouvent être suffisamment nombreux pour avoir une mauvaise influence sur la réputation du langage...
7. comme la programmation orientée objets
8. On pourra penser au *garbage collector* introduit dans quantité de langages
9. Il faut entendre : cette fois-ci le *design* de base du langage est correct.
10. Il n'y a qu'à voir la taille de la bibliothèque standard du C++ par rapport à celle du C

Interface en ligne de commande

La plupart des ordinateurs proposent ajourd'hui aux utilisateurs *lambda*[11] 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 pensé 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 effectués 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 pensé 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.

Compilateur

Sous Linux :

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

```
user@computer ~> sudo apt-get update && sudo apt-get upgrade
user@computer ~> sudo apt-get install gcc
user@computer ~> gcc -v
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 puis d'extraire l'archive dans C :/ de sorte à avoir un répertoire C :/mingw64/.

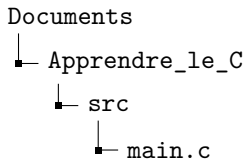
Le compilateur est alors dans le répertoire C :/mingw64/bin/ et pourra être utilisé pour compiler les programmes écrits en C ou en C++.

2.4.2 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[12] :

11. Sans rien de péjoratif, précisons.
12. Vous êtes bien sûr libre de faire autrement, il s'agit simplement de poser une structure conventionnelle pour la suite du cours.



Le répertoire “src” [13] 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 :

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\Apprendre_le_C> C:/mingw64/bin/gcc.exe src/main.c -o main.exe
3 C:\Users\user\Documents\Apprendre_le_C> 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” [14]
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.

2.4.3 Analyse du premier programme

On rappelle le premier programme écrit en C :

Un premier programme

13. Pour “source”
14. “-o” signifie *output* et “pedantic” spécifie au compilateur d’être intransigeant avec la spécification standard du langage.

```c
#include <stdio.h>

int main()
{
  printf("Hello World !\n");
  return 0;
}
```

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 language 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 [15]. 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.

Analyse ligne par ligne

```c
#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 [16]), 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.

15. Avec certaines contraintes théoriques et pratiques
16. 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

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.

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 inclue 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 [17].

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 avancées [18] qui abordent certaines subtilités et particularités de première nécessité douteuse mais d'utilité parfois avéré

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 lié 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 [5], bourrage de crâne tout à fait inutile, puisque rien ne pourra en être fait [19].

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 caractère universel à la programmation (on pensera par exemple au concept de *variable*). Ensuite que

17. Rien ne sera donc dit vis-à-vis des spécificités de la programmation de systèmes embarquées 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.

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

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

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. [20]
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 λ-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 [21]. Si l'utilisateur ne l'utilise pas consciemment, il apparaît ainsi systématiquement dans le fonctionnement interne des langages [22].

---

20. 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.

21. 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.

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

CHAPITRE

3

FONDAMENTAUX DU LANGAGE

Identifiants

Définition 14 : Identifiant

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

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

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

Exemples :

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

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

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

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

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

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

Les mots clés du langage C sont les suivants :

alignas (C23)	extern	sizeof	_Alignas (C11)
alignof (C23)	false (C23)	static	_Alignof (C11)
auto	float	static_assert (C23)	_Atomic (C11)
bool (C23)	for	struct	_BitInt (C23)
break	goto	switch	_Bool (C99)
case	if	thread_local (C23)	_Complex (C99)
char	inline (C99)	true (C23)	_Decimal128 (C23)
const	int	typedef	_Decimal32 (C23)
constexpr (C23)	long	typeof (C23)	_Decimal64 (C23)
continue	nullptr (C23)	typeof_unqual (C23)	_Generic (C11)
default	register	union	_Imaginary (C99)
do	restrict	unsigned	_Noreturn (C11)
double	return	void	_Static_assert (C11)
else	short	volatile	_Thread_local (C11)
enum	signed	while	

Remarque 1 : Quelque soit le standard utilisé pour votre propre compilation, il 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 bogues à la compilation.

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

Internationalité des identifiants

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

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.

Commentaires

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.

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. [2]

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 [3]. 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.

3.2.1 Commentaires sur une ligne

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

Exemple

```
printf("Salut les gens !\n"); // \n 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");
```

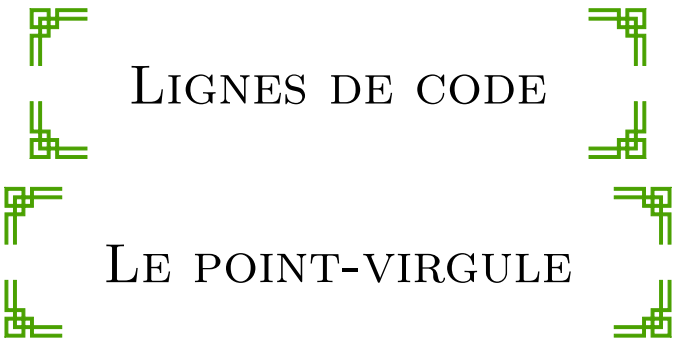
compile sans erreur et produit la sortie : Je ne dirais // pas bonjour.

3.2.2 Commentaires par bloc

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

Exemple

```
1  /*
2  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 (*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 3.2 et 3.6).

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) :

```c
#include <stdio.h> // Les instructions de préprocesseurs ne finissent pas par un point-virgule

int main() {
  // Les instructions finissent par des point-virgules mais pas les commentaires
  printf("Salut !\n"); printf("Deuxieme Salut !\n");
  return 0;
}
```

Petit apparté : 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

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.

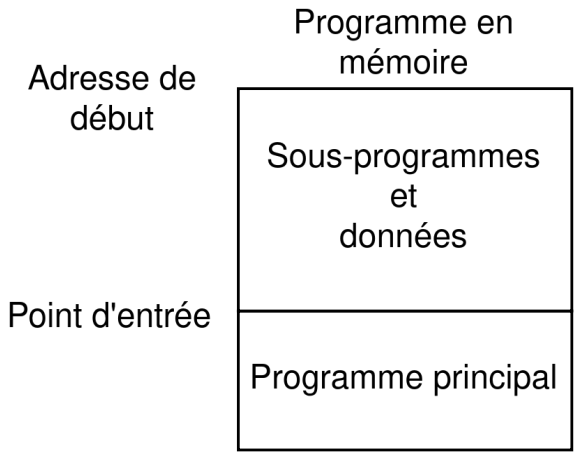


FIGURE 3.1 – Schéma simplifié d'un programme en mémoire

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 réussie" 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 :

```c
#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

```c
// L'ordre d'inclusion des modules est (presque) sans importance
#include <stdlib.h> // EXIT_SUCCESS et EXIT_FAILURE (entre autre)
#include <stdio.h> // printf (entre autre)

int main() {
  return EXIT_SUCCESS; // Signifie une exécution réussie du programme
}
```

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 [4].

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

```c
#include <stdio.h>

#define RETURN_CONSTANT 0

int main()
{
  return RETURN_CONSTANT;
}
```

Remarque 1 : La définition d'un symbole par la directive #define remplace la valeur définit 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 utilisé 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 [5]. Ces directives ne font pas partie *en soi* du langage.

Un approfondissement technique sera effectué ultérieurement.

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

CHAPITRE

4

BASES DU LANGAGE

Variables

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 $(r, g, b, a) \in [\![0, 255]\!]^4$, 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 de la variable
- une adresse mémoire
- une donnée sur N octets (ou $8N$ bits) à cette adresse

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

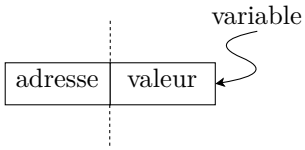


FIGURE 4.1 – Schéma du stockage d'une variable

Type d'une valeur de variable

Définition 15 : 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 [a] et un ensemble quelconque d'éléments. Cette correspondance est appelée un *type*.
On a vu un certain nombre d'interprétations dans le premier chapitre de la première partie ($btoi_u$, $btoi_s$, norme *IEEE 754* et caractères ASCII).

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

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 $\mathbb{R}$ [1]. Une correspondance supplémentaire entre $[\![0;127]\!]$ et les caractères de la langue anglaise vient en surcouche.

Le type permet donc de délimiter les N octets d'un mot binaire et de l'interpréter.

Le langage C propose les types de base suivants :

- Pour représenter des entiers
  - char : valeur $v \in [\![-128;127]\!]$ représentée par une donnée sur 1 octet (8 bits)
  - short int : valeur $v \in [\![-2^{15};2^{15}-1]\!]$ représentée par une donnée sur 2 octets (16 bits)
  - int : valeur $v \in [\![-2^{31};2^{31}-1]\!]$ représentée par une donnée sur 4 octets (32 bits)
  - long int : valeur $v \in [\![-2^{63};2^{63}-1]\!]$ représentée par une donnée sur 8 octets (64 bits)
- Pour représenter des nombres à virgule :
  - 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

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

Petit apparté : 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 $e \in E. \ldots$", 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 $e \in E. \ldots$", où E désigne un ensemble quelconque
- "$\underset{x \text{ indéterminé}}{\forall x}, x \in E \implies \mathcal{P}(x)$", 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
char 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 définit (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 par une fonction binaire renvoyant $Vrai$ si les deux objets sont égaux. Il permet cependant de changer à tout instant la valeur d'une variable déjà définie :

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

Petit apparté : 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 pour faire correspondre le sous-ensemble des entiers $[\![-128; 127]\!]$ et l'ensemble des caractères ASCII, bien que l'interprétation soit différente. Cela est compréhensible puisque les mots binaires utiles pour ces deux interprétations sont les mêmes, il s'agit cependant d'une petite bizarrerie conceptuelle du langage [a]. On peut écrire un caractère ASCII grâce aux guillements simples :

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

ce qui revient strictement à écrire :

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

a. Et ce n'est pas la seule, le C n'ayant peut-être pas été conçu "à l'arrache" mais en tout cas certainement pas de manière super carrée... il suffit de voir les tableaux statiques.

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 très largement utilisée [2]

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.

4.1.1 Taille et type d'une variable

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

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

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

Opérateur sizeof

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

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

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

TYPE a;
sizeof(a); // 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 :

```
typeof(int) a;
int b;
typeof(b) c;
```

```
int a;
int b;
int c;
```

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.*[3]

4.1.2 Entiers signés et non signés

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

```
char v = 130; // signé par défaut. La vérité est que v = 2 - 128 = -126
unsigned short int v2 = -1; // interpretation non signée, la vérité est que v2 = 65535
signed int v3 = -4; // 'signed' est inutile car implicite
// etc...
```

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

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

4.1.3 Environnement et blocs

Définition 16 : 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 précisement 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 17 : 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 langagee 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 :

```c
#include <stdlib.h>

// 'main' est déclarée dans l'environnement global :

int main()
{ // Premier bloc : environnement local de 'main'

  int a = 5; // déclaré dans l'environnement local de 'main'

  { // Définit un sous-environnement local dans 'main'

    // ici, 'a' vaut toujours 5

    double a = 3.14; // Déclaration d'une nouvelle paire nom-valeur pour 'a'

    // ici, 'a' vaut 3.14

  } // sortie du sous-environnement local

  // 'a' vaut à nouveau 5

  return EXIT_SUCCESS;
} // sortie de l'environnement local de 'main'

```

Remarque : 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.

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 :

```c
#include <stdlib.h>

int some_global_variable = VALUE;
int main() {
  char some_global_variable = OTHER_VALUE;
  // du code
}

// Un autre code peut aussi accéder à 'some_global_variable' et à 'main'
```

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

Cas d'application réel : On essaie en général d'alléger, dans la mesure où cela est utile[4], 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 bogues d'origine humaine qui consistent à réutiliser deux fois le même nom d'un même environnement dans des contextes différents.

4.1.4 Exercices

Exercice 13 (Interversion 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 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.

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

Ce chapitre est introduit pour des raisons purement pratique, pour permettre à tout(e) lecteur/rice de tester ses programmes ou d'afficher les résultats d'exécution d'un programme. [5]

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 \ [6]
- 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.

```c
#include <stdio.h>
#include <stdlib.h>

int main() {
// produit un avertissement à la compilation et affiche "'est pas affichable." :
  printf("% n'est pas affichable.");
// erreur de compilation  car '\ ' n'est pas un caractère
  printf("\ non plus.");
  return EXIT_SUCCESS;
}
```

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

```c
#include <stdio.h>
#include <stdlib.h>

int main() {
  printf("%% n'est pas affichable.\n");
  printf("\\ non plus.\n");
  return EXIT_SUCCESS;
}
```

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.

4.2.1 Formatage des caractères spéciaux

Les caractères considérés comme spéciaux sont :

- caractère de passage à la ligne suivante : \n [7]
- caractère retour au début de la ligne actuelle : \r [8]

5. Par ailleurs, le formatage de chaîne de caractère tel que présenté n'est pas une caractéristiques 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

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"*

- caractère de tabulation : \t
- caractère de chaîne de caractère " : \"

Exemples

```c
#include <stdio.h>
#include <stdlib.h>

int main() {
  printf("Pas de retour a la ligne apres cette phrase.");
  printf("Alors que la oui.\n");
  printf("Recommencer au \"debut\" : \rdebut\n");
  printf("Une tabulation : \t c'etait une tabulation.\n");
  return EXIT_SUCCESS;
}
```

4.2.2 Formatage de variables

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

On distingue les principaux modificateurs d'affichage :

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

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 :

```c
#include <stdio.h>
#include <stdlib.h>

#define N 3

int main() {
  printf("Valeur du nombre N : %d", N);
  return EXIT_SUCCESS;
}
```

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 :

```c
// 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 :

```c
#include <stdio.h>
#include <stdlib.h>

int main() {
  unsigned short int a = -7;
  int b = 10;
  printf("%u + %d = %d\n", a, b, a+b);
  return EXIT_SUCCESS;
}
```

4.2.3 Exercices

Exercice 14 (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 18 : Opérateurs et opérandes

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

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

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

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

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

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

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

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

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

```
TYPE v3 = (v2 <OP2> (v1 <OP1> v2)); // l'opérateur OP1 est calculé avant OP2
```

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

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

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

Pour tous les opérateurs présentés dans la suite, on pourra se référer à la table de priorités suivante [9] :

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 $\ll$ et $\gg$	Gauche à droite
6	< <= > >=	Relations $<$ et $\leq$ Relations $>$ et $\geq$	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

9. Copiée de https://en.cppreference.com/w/c/language/operator_precedence parce-que sur ce genre de chose, inutile d'être original.

TABLE 4.1 – Priorités des opérateurs en C

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

4.3.1 Opérateurs arithmétiques

On liste les opérateurs arithmétiques suivants :

Opération	Arité	Symbole mathématique	Symbole en C
Addition	2	$+$	$+$
Soustraction	2	$-$	$-$
Multiplication	2	$\times$	$*$
Division entière	2	$\div$	$/$
Modulo	2	%	%

TABLE 4.2 – Opérateurs arithmétiques

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

```c
#include <stdio.h>
#include <stdlib.h>

int main() {
  int a = 18;
  int b = 67;
  printf("a = %d; b = %d\n", a, b);
  printf("a + b = %d\n");
  printf("a - b = %d\n");
  printf("a * b = %d\n");
  printf("a / b = %d\n");
  printf("b / a = %d\n");

  return EXIT_SUCCESS;
}
```

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

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

4.3.2 Opérateurs bit-à-bit

On liste les opérateurs bit-à-bit suivants :

Opération	Arité	Symbole mathématique	Symbole en C
ET	2	$\wedge$	&
OU inclusif	2	$\vee$	|
OU exclusif	2	$\oplus$	$\wedge$
NON	1	$\neg$	$\sim$
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 4.3 – Opérateurs bit-à-bit

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

4.3.3 Opérateurs logiques, ou relationnels

Définition 19 : Vérité d'une expression

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

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

On liste les opérateurs logiques suivants :

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

TABLE 4.4 – Opérateurs logiques

Remarque : Il s'agit d'un OU *inclusif*. Le OU exlusif n'a pas de version "logique" en C.

Quelques exemples :

```c
int a = 5;
int b = 7;
printf("%d\n", !(b-a)); // b-a = 2 est vrai donc !(b-a) est faux -> 0
printf("%d\n", !!(a + b)); // -> !0 = 1
printf("%d\n", a != b); // -> 1
printf("%d\n" a == b); // -> 0
printf("%d\n", !!(a+b) == (a != b)); // -> 1
printf("%d\n", a != b && a == b); // -> 0
printf("%d\n", a == b || (a + b)); // -> 1
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, $a \neq b \Leftrightarrow a - b \neq 0$

4.3.4 Opérateurs d'assignation

Les opérateurs d'assignation sont à la fois les éléments les plus connus par les débutants et à la fois les opérateurs dont ces mêmes débutants ignorent tout des spécificités. Voici un exemple qui pourrait surprendre :

```c
#include <stdio.h>
#include <stdlib.h>

int main() {
  int a;
  printf("Initialement, a = %d\n", a = 0);
  printf("Puis a = %d\n", ++a);
  printf("Et enfin, a = %d\n", a <<= 3);
  return EXIT_SUCCESS;
}
```

Les différentes opérations présentent dans ce code sont expliqués dans la suite [10].

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 :

```c
int a;
int b = (a = 3) + 1;
```

```c
int a = 3;
int b = a + 1;
```

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

```c
#include <stdio.h>
#include <stdlib.h>

int main() {
  int a, b, c, d, e = (d = (c = (b = (a = 3))));
  printf("a : %d, b = %d, c = %d, d = %d, e = %d\n", a, b, c, d, e);
  return EXIT_SUCCESS;
}
```

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 :

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

```
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'incrémentation 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 optimisé ces opérations :

- ++variable ;[11]
- –variable ;

Il y a équivalence entre les expressions suivantes :

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

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

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

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

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

Dans ce cas, la valeur renvoyée est celle *avant* l'incrémentation 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;
```

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

4.3.5 Exercices

Voir [2] pour approfondir les calculs binaires optimisés.

Exercice 15 (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 16 (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 17 (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 18 (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 19 (Interversion sans effet de bord (2)) [15]. *idem* que pour l' **Exercice 18** mais l'unique opérateur autorisé est l'opérateur de OU exclusif (noté $\wedge$ en C).

Exercice 20 (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 57×14 :

```
1 #include <stdio.h>
2 #include <stdlib.h>
3
```

```c
int main() {
  int a = 57;
  int r;
  // ici : code à déterminer
  printf("57x14 = %d", r);
  return EXIT_SUCCESS;
}
```

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

Exercice 21 (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 :

```c
int x = ...; // valeur quelconque, positive comme négative
unsigned int abs_x;
// ici : code à déterminer;
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 :

```c
int some_integer = 7851;
int divider = 100;
double a = some_integer/divider;
printf("%.2lf\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 convertit implicitement en double par l'opérateur =.
Il faudrait que la division effectuée ne soit pas une division entière. Pour cela, il est nécessaire de modifier l'interprétation d'au moins une des deux variables (some_integer ou divider) pour qu'elle soit interprétée comme un double AVANT la division. On ne souhaite cependant pas modifier le type d'une des deux variables de manière définitive.

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

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

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

- projection de nombre à virgule vers entier : donnée modifiée et valeur approximée de l'ancienne (car perte de décimales)
- projection d'entier vers nombre à virgule : donnée 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 : donnée non modifiée et valeur modifiée (simple changement d'interprétation)
- projection de pointeurs T_1* vers T_2* : donnée non modifiée [13]

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
```

4.4.1 Exercices

Exercice 22 (Quelques évaluations entières) [13].
Dire pour chaque expression si elle est vraie ou fausse (respectivement : différente ou égale à 0) :

```
int e1 = 0xFFFFFF00 != 0xFFFFFFFF - (char)(-1);
int e2 = 0xFFFFFFFF - (unsigned char)(-1) - 0xFFFFFF00;
int e3 = !(e1 == e2) || (1 ^ e2);
int e4 = (unsigned char)(!(((64 ^ e3) % 8) - 1) + 256) - 2;
int e5 = e1 && e3;
```

12. Parce-que le langage C est conceptuellement claqué au sol, mais *trènkile*.
13. et c'est tout car la valeur et la donnée des pointeurs sont identiques

Structures de contrôle

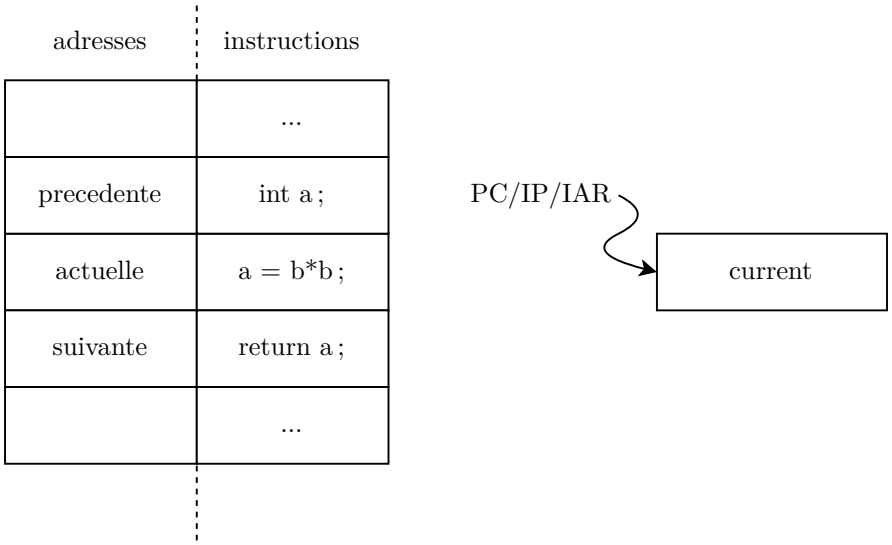
Définition 20 : Flot d'exécution/de contrôle

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

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

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

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



La première sous-section se veut une introduction plus bas-niveau du contrôle du flot d'exécution, c'est-à-dire détaillant de manière moins abstraite et plus technique le contrôle du flot d'exécution. [14] Il est possible d'aller directement à la sous-section 4.5.2 décrivant les structures complexes, majoritairement utilisées en langage C.

4.5.1 Structures élémentaires

Voir [4] pour plus de détails.

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

- la séquence

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

- 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. En langage C, c'est le point-virgule indiquant la fin d'une instruction qui se charge d'indiquer le passage à la suivante.

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

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 accélére l'accès aux instructions par le processeur, mais limite le nombre possible d'instructions. [15]

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 :

```
1 #include <truc.h>
2
3 int main() {
4   goto fin;
5   // ici : opérations jamais exécutées
6 fin:
```

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

```
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). [16]
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
```

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

16. voir https://wiki.osdev.org/CPU_Registers_x86#EFLAGS_Registeret https://en.wikipedia.org/wiki/FLAGS_registerpour une liste des drapeaux des processeurs Intels

Algorithme 3 : Traduction en pseudo-code

```
1 al ← 200;
2 al ← 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 sautés
7 Fin du programme;
```

```
1 ; ...
2 mov al, 10
3 boucle:
4 ; trucs à répéter 10 fois
5 sub al, 1
6 jnc boucle ; saute si al - 1 >= 0
7 fin_de_boucle:
8 ; ...
```

4.5.2 Structures complexes

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

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

Les détails de la construction de ces structures abstraites ne seront pas décrits ici. Cela peut en effet dépendre du compilateur. [17]

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

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

Branchement if-else :

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

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

Algorithme 4 : Branchement conditionnel

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

```c
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 :

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

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

On pourrait aussi écrire de manière plus lisible :

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

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

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

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

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

```c
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 defaut"); // sera toujours exécuté
```

Opérateur ternaire

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

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

```c
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 [18], sera simplement réduite à ses conséquences : l'utilisation d'une condition ternaire permet d'optimiser la vitesse d'exécution des instructions par l'UCC [19].

Un petit exemple d'utilisation :

```c
/*
<math.h> nécessite l'option -lm
à la fin de la ligne de commande pour compiler :
  gcc main.c -o main --pedantic -lm
*/
#include <stdlib.h>
#include <math.h> // NaN (Not A Number), sqrt

int main() {
  double a = ...; // valeurs
  double b = ...; // quelconques
  double c = ...;

  // Résoud une équation à solutions réelles de degré 2
  double d = b*b - 4*a*c;
  double x1 = (d < 0) ? NAN : (-b - sqrt(d))/(2*a);
  double x2 = (d < 0) ? NAN : (-b + sqrt(d))/(2*a);

```

18. Pour les curieux, voir https://drive.google.com/file/d/1bkLb3OByL_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.

```
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 posséde quelques avantages et inconvénients :

- avantages :
  - très lisible
  - permet une optimisation plus poussé du code par le compilateur [20]
  - 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 test que des égalités
  - les valeurs à égaler doivent être prédéterminés 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 [21], ou d'un simple menu dans l'application d'un restaurant [22].

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

```c
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");
  // et d'autres instructions
```

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 impressionants, 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

```
  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" [23] 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);
14   return EXIT_SUCCESS;
15 }
```

23. Littéralement.

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

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

Exercice 23 () [07]. Modifier le programme donné en exemple pour qu'il affiche toutes les valeurs $u_1, \ldots, u_N$.
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 :

```
do {
  // Bloc d'instructions
} while (condition);
```

```
// Bloc d'instructions
while (condition) {
  // Même bloc d'instructions
}
```

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*

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

L'idée est en générale d'utiliser l'instruction de *pas* pour parcourir une structure. Il s'agit d'une itération sur un élément de parcours, raison qui donne son nom à cette syntaxe de boucle. Son application va être vue dans la section 4.9 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 :

```c
// 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);
}
```

4.5.3 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.

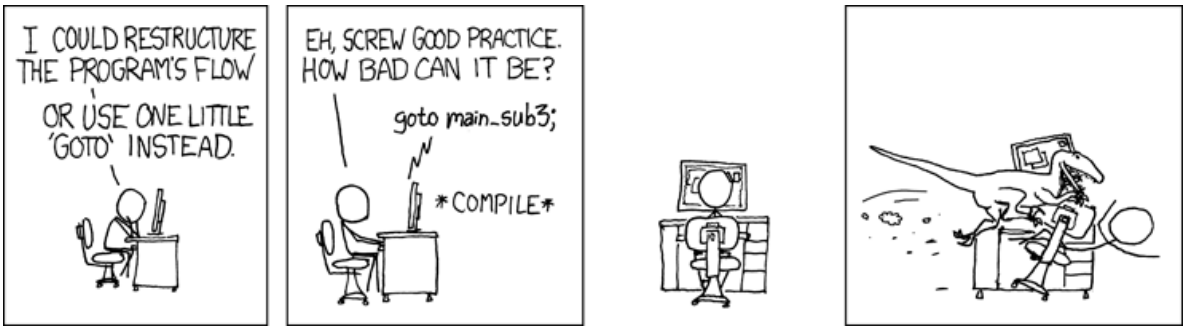
La réponse tient simplement : l'instruction break casse les blocs d'instructions while, do-while, for et switch, sans aucune exception. Le contrôle du programme est ensuite directement rendu à la première instruction suivant le bloc. Par exemple :

```c
#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*[24].

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

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.[25] En effet, ce genre de saut est effectué sans contrôle (donc augmente les risques de bogues) et permet moins d'optimisations de la part du compilateur car son fonctionnement ne possède pas de logique intrinsèque, au contraire des boucles *for* et *while* qui suivent des règles contraintes par des conditions. 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 avéré.[26]

4.5.4 Exercices

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

Exercice 24 (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 $100 < age$[27]

Essayer de minimiser le nombre de branchements conditionnels (objectif : trois if maximum, et seule la phrase correcte doit s'afficher[28]).

Exercice 25 (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 :

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

Exercice 26 (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[29].

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

Routines

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

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

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

Nous allons voir un cas particulier de sous-programme appelé la *routine*[30], dont il n'existe que deux formes[31] 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 :

```c
void procedure_name(parametres) {
  // bloc d'instructions
  return; // termine la routine, c'est-à-dire
}
```

```c
TYPE function_name(parametres) {
  // bloc d'instructions
  return VALUE; // VALUE de type TYPE       e la routine
}
```

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

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

$$\begin{array}{rccc} addition & : & [\![-2^{31}; 2^{31}-1]\!]^2 & \to & [\![-2^{31}; 2^{31}-1]\!] \\ & & (a, b) & \mapsto & a+b \end{array}$$

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.

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.

```c
#include <stdio.h>
#include <stdlib.h>

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

int main() {
  printf("%d + %d = %d\n", 2, 3, addition(2, 3));
  return EXIT_SUCCESS;
}
```

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

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

```c
void afficher_diff_0(int x) {
  if (x == 0) {
    return; // termine la routine
  }
  printf("x != 0\n"); // jamais exécuté si x = 0
}
```

4.6.1 Signature et prototype

Définition 21 : Prototype

Le prototype d'une routine est la donnée de :

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

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

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

int addition(int a, int b) { // Déclaration 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 :

```c
#include <stdio.h>
#include <stdlib.h>

// Chaque ligne ci-dessous montre l'existence de la fonction 'addition' au compilateur
int addition(int a, int b); // Prototype de la fonction 'addition'
// OU
int addition(int, int); // Les noms des paramètres sont optionnels dans le prototype

int main() {
  printf("%d + %d = %d", 2, 3, addition(2, 3));
  return EXIT_SUCCESS;
}

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

Définition 22 : Signature

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

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

4.6.2 Note relative à la copie des arguments

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

```c
#include <stdio.h>
#include <stdlib.h>

void proc(int c) {
  v = 48; // ERREUR : v non déclarée dans cette fonction
  return c;
}

int main() {
  int v = 5;
  proc(v+1);
  v = c;// ERREUR : c non déclarée dans cette fonction
  return EXIT_SUCCESS;
}
```

En particulier :

```c
#include <stdlib.h>

int polynome(char p) { // Déclaration de la fonction 'addition'
```

```
  p = (p*p - p + 1);
  return (int)p;
}
int main() {
  char p = 3;
  polynome(p);
  return EXIT_SUCCESS;
}
```

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

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

Définition 23 : Argument

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

Ainsi, dans le code suivant :

```
#include <stdlib.h>

int polynome(char p) { // Déclaration de la fonction 'addition'
  return (int)(p*p - p + 1);
}
int main() {
  int a = 3;
  polynome(a);
  return EXIT_SUCCESS;
}
```

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 :

```
#include <stdlib.h>

int polynome(char p) { // Déclaration de la fonction 'addition'
  p = (p*p - p + 1);
  return p; // projection de type implicite
}
int main() {
  int a = 3;
  polynome(a);
  if (a == 3) {
    printf("a non modifie\n");
  }
  a = polynome(a); // a = 3*3 - 3 + 1 = 7
  if (a != 3) {
    printf("a modifie\n");
  }
```

```
17    return EXIT_SUCCESS;
18  }
```

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

4.6.3 La pile d'exécution

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

```
1   #include <stdio.h>
2   #include <stdlib.h>
3
4   #define X ... // 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  }
```

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

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

La pile contient *entre autre* :

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

Les variables contenues dans la pile ne sont accessibles que dans la routine qui les a créé. 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 apparté : À propos de l'allocation de la mémoire

L'allocation de mémoire de variables est entièrement géré par le compilateur. En effet, les variables crées dans un programme sont analysés avant la compilation et le programme en binaire finale sera en vérité une alternative au code écrit par le programmeur qui sécurisera l'accès à la mémoire pour éviter certains bogues.
Par exemple, la taille de la pile au chargement du programme est généralement de taille fixe (cela dépend du système d'exploitation, mais on considère ici des systèmes classiques comme Linux ou Windows). Ainsi, empiler de trop nombreux appels de routines les uns sur les autres peut amener à une erreur de dépassement de pile (*stack overflow* en anglais). De la même façon, il ne devrait pas être possible d'allouer de trop nombreuses variables sur la pile dans une seule fonction. Certaines techniques permettent toutefois d'éviter des erreurs à ce niveau en stockant les variables les plus anciennes/les moins utilisées ailleurs que sur la pile, comme par exemple sur le tas ou même sur le disque dur (pour d'immenses programmes) malgré la perte de vitesse que cela engendre. On suppose en effet que ces variables ne sont utilisées que rarement et qu'une baisse de vitesse pour leur accès est négligeable.
Il est cependant possible de gérer par soi-même l'allocation des variables en mémoire. C'est ce qu'on appelle l'*allocation dynamique*. La pile n'est jamais [a] utilisée pour ce type d'allocation.

a. à deux trois détails près...

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

Remarque 1 : l'utilisation de blocs de code fait aussi bien le travail. Il s'agit donc d'abord de structuration du code et de facilité d'utilisation par un tiers.

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 [32] 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

32. Extrême et peu réaliste, certes

que tel.[33] Il est par exemple absurde de programmer une routine qui est si spécfique qu'elle ne sera appelée qu'une seule fois dans tout un programme.

4.6.4 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 extérieurs 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[34].

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

4.6.5 Exercices

Exercice 27 (Encore Fibonacci) [11].
Écrire une fonction de prototype `unsigned int fibo(int n)` ; qui à n renvoie f_n où $(f_n)_{n\in\mathbb{N}}$ est définie telle que :

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

Tester ensuite cette fonction pour toutes les valeurs de $n \in [\![0; 100]\!]$

Exercice 28 (Puissances entières) [10]. Écrire une routine équivalente à la fonction :

$$\begin{array}{rccc} f & : & [\![-2^{31}, 2^{31}[\![\times [\![0, 2^{32}[\![& \to & \mathbb{Z} \\ & & (x, n) & \mapsto & x^n \end{array}$$

Dans quelle mesure cette fonction est-elle correcte ?

Probablement la section la plus simple de ce document, l'utilisation des pointeurs n'induira jamais de bogues étranges et inexpliqués dans un programme C. Tout un chacun peut affirmer sans hésitation

33. 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 ∧∧

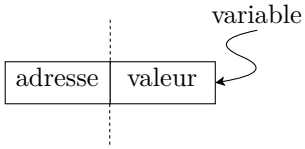
34. Voir la section 4.8 sur les flux standards pour plus de détails.

qu'il n'y a rien de plus *trivial* que la programmation avec des pointeurs.
Tout va bien se passer...
Maintenant que le lecteur est ***rassuré***, commençons...

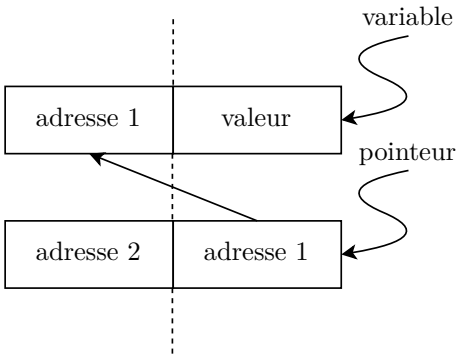
Rappel : Une variable peut être approximée comme la collection de trois informations :

- une étiquette, aussi appelée le nom de 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 [35]. En particulier, ils permettent une flexibilité très importante des programmes écrits. En effet, une variable est définie dans un certain espace. Ainsi, une variable d'une routine est locale à cette routine, et n'existe pas en dehors. Une adresse mémoire peut potentiellement être utilisée hors de routines spécifiques. Les possibilités des routines sont donc décuplées puisque celles-ci peuvent interagir avec leur environnement autrement que par la valeur de retour (dans le cas d'une fonction). Cela donne également une plus grande importance aux procédures.

Cela amène cependant un inconvénient : en perdant un contrôle absolu sur l'existence de données, il devient possible *par erreur* d'accéder à des zones mémoires qui ne sont plus utilisés. Il s'agit là de l'erreur la plus classique en programmation en C [36].

Un pointeur est définit selon la syntaxe suivante :

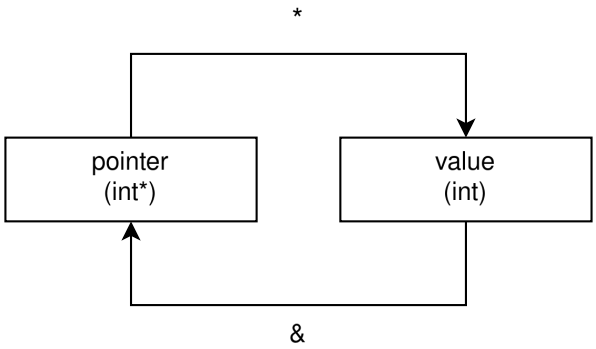
```
TYPE* ptr;
```

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

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

35. Et presque seulement en C...
36. traduite par le fameux message d'erreur *Segmentation Fault*

soit logique ou bit-à-bit. Il est cependant possible de l'utiliser comme opérateur unaire. Sa signification est alors tout autre. &v renvoie l'adresse mémoire à laquelle se situe la variable v.



On peut donc écrire :

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

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

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

```
int *pointeur = NULL; // 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
```

Petit apparté : 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 [a]
- le segment de pile : contient la pile d'exécution

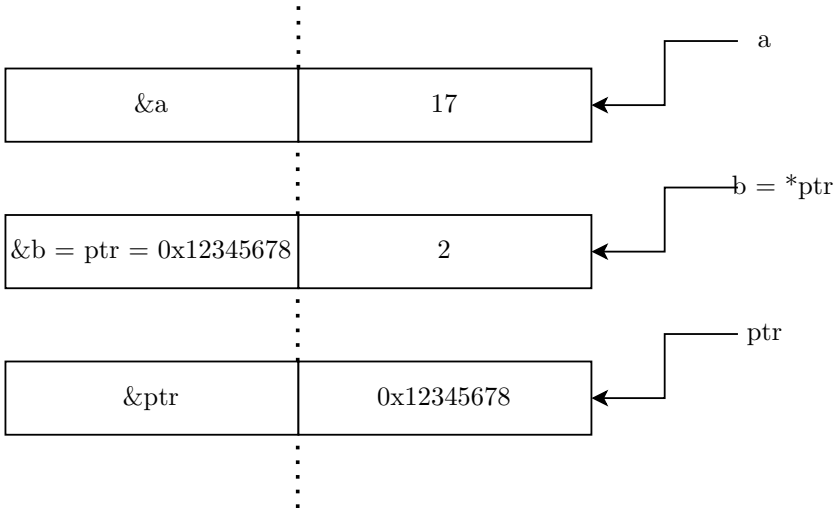
Une erreur de segmentation provient principalement de la tentative d'accès à un segment de manière non autorisée. Il peut s'agir d'un segment du programme propre ou d'un autre programme. Dans tous les cas, cela est représentatif de l'accès non autorisé par le système d'exploitation ou par le processeur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

L'opérateur $*$, lorsqu'il est utilisé commme 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 :

```c
#include <stdio.h>
#include <stdlib.h>

int main() {
  int a = 58;
  int b = 67;
  int* ptr = &a; // pointe vers a
  printf("Origine : a = ", *ptr);
  *ptr = 17; // pareil que a = 17
  printf("a = %d\n", a);
  ptr = &b; // pointe vers b
  printf("Origine : b = ", *ptr);
  *ptr = 2; // pareil que b = 2
  printf("b = %d", b);
  return EXIT_SUCCESS;
}
```

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*[37].

4.7.1 *lvalue* et *rvalue*

Les codes ci-dessus peut avoir provoquer chez le lecteur un froncement de sourcils, en particulier si celui-ci est porté à la rigueur des 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 dernière 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.

37. 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*[38] : symbole qui permet d'évaluer l'*identité* d'un objet, d'un texte binaire
- les *rvalues*[39] : 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 :

```
int a;
int b = 6;
int* p = &a;
*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.

4.7.2 Formatage en chaîne de caractères

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

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

int main() {
  int a = 58;
  int b = 67;
  int* ptr = &a; // pointe vers a
  printf("&a = %p et a = %d\n", &a, a);
  printf("&ptr = %p, ptr = %p et *ptr = %d\n", &ptr, ptr, *ptr);
  ptr = &b; // pointe vers b
  printf("&b = %p et b = %d\n", &b, b);
  printf("&ptr = %p, ptr = %p et *ptr = %d\n", &ptr, ptr, *ptr);
  return EXIT_SUCCESS;
}
```

4.7.3 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.

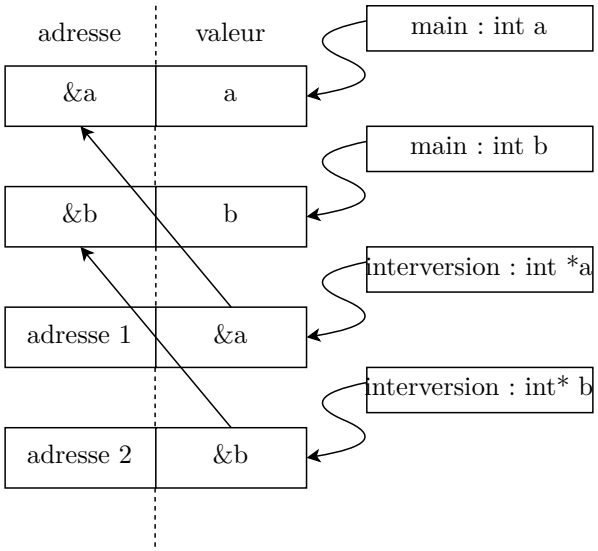
C'est-ici qu'interviennent les pointeurs :

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

```c
#include <stdio.h>
#include <stdlib.h>

void interversion(int* a, int* b) {
  int tmp = *a;
  *a = *b;
  *b = tmp;
}

int main() {
  int a = 5;
  int b = 7;
  interversion(&a, &b);
  printf("a = %d et b = %d\n", a, b);
  return EXIT_SUCCESS;
}
```

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

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



Alors, $*a$ dans interversion est la valeur de a dans la fonction main et $*b$ dans interversion est la valeur de b dans la fonction main. On procède alors à une interversion avec effet de bord, comme dans l' **Exercice 13** .

4.7.4 Arithmétique des pointeurs et projection de type

Un point important n'a pas été abordé sur les pointeurs [40] : l'arithmétique des pointeurs, fortement liée à la projection de type sur les pointeurs.

Considérons le code suivant :

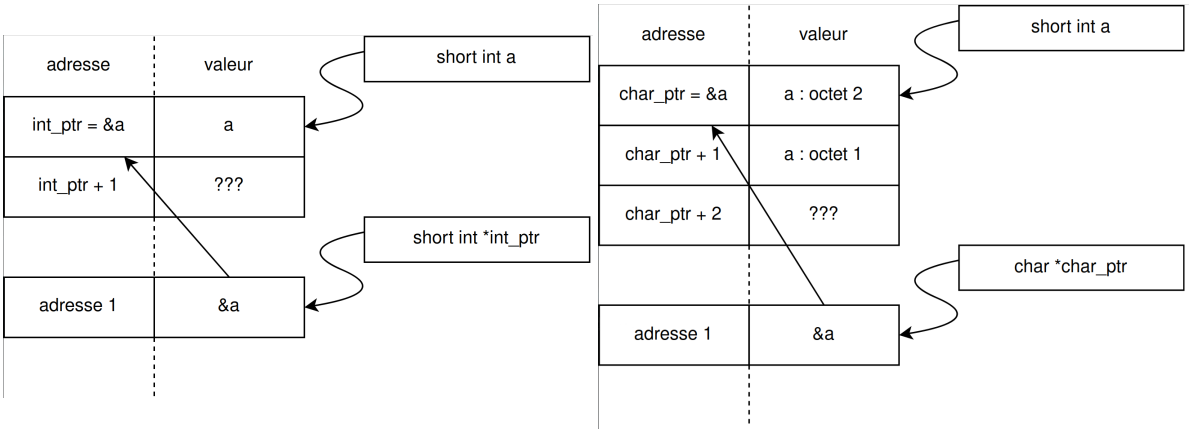
40. Sans jeux de mots aucun... puisque les jeux de mots laids font des gambettes ;)

```c
#include <stdio.h>

int main() {
  short int a = 7187;
  short int *int_ptr = &a;
  printf("%d\n", *int_ptr);
  char *char_ptr = (char *)int_ptr; // projection de type explicite
  printf("%d\n", *char_ptr);
  printf("%d\n", *(char_ptr + 1));
}
```

On observe que :

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

La projection de type en char*, qui pointe vers des valeurs de 1 octet, permet d'accéder aux valeurs de chacun des octets pris séparémment 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 :

```c
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[41].

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 :

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

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

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

Les raisons de l'existence de cette facilité d'écriture seront détaillées dans la section 4.9 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 4.10 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 :

```
void *pointeur_quelconque = NULL;

// projection de type explicite pour éviter les avertissements à la compilation
int *int_ptr = (int*)pointeur_quelconque;
```

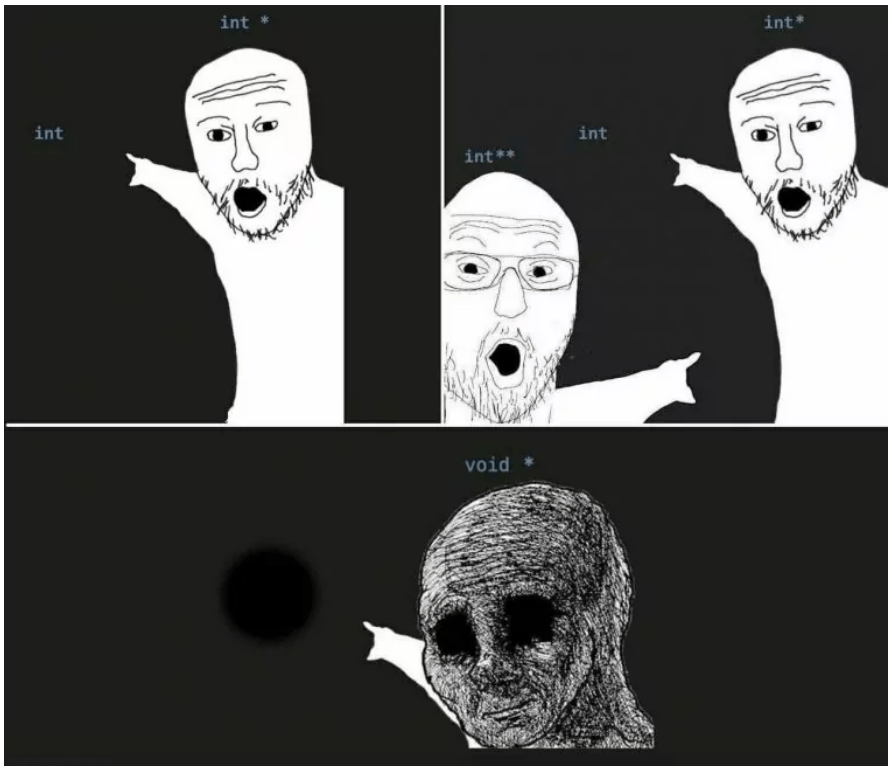
Par ailleurs, l'addition sur un pointeur non typé est "classique" :

```
void *pointeur_quelconque = NULL;
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 4.10 sur les tableaux dynamiques qui montre une application essentielle des pointeurs.

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

4.7.5 Pointeurs itérés



4.7.6 Exercices

Exercice 29 (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 pointé 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 pointé par x à la puissance n

Exercice 30 (Interversion sans effet de bord (3)) [10]. Écrire une procédure `void swap(int *a, int *b);` qui échange les valeurs des variables pointés par a et b sans utiliser de variable temporaire. Quel est le comportement du programme si $x = y$? (c'est-à-dire que la même adresse est passé en argument pour les deux paramètres)
Si cela induit une erreur de comportement de la procédure, corriger cette erreur.

Exercice 31 (Distance de Manhattan) [25]. Cet exercice est là uniquement comme exercice technique[42]. La méthode utilisée est à la fois inefficace et immonde à lire et à programmer. On se rapportera aux **Structures** pour un implantation correcte.

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.

42. et rigolo (?)

2. Écrire une fonction `double substract(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*[43]

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

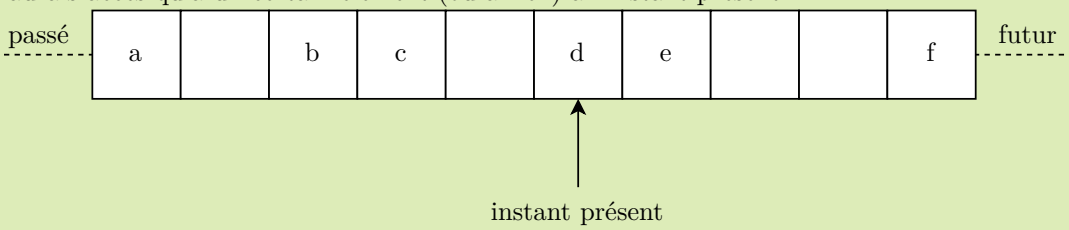
Exercice 32 (Valeur absolue (2)) [18]. En utilisant seulement des projections de type et une opération bits-à-bits, écrire une fonction `float abs(float x)` ; qui calcule la valeur absolue de x.

Exercice 33 (Racine carrée inverse rapide (2)) [25]. Reprendre le résultat de l' **Exercice 9** 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 24 : 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'aurais 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 comportement 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*
- au flux de sortie d'un texte sur un terminal
- au flux d'entrée du clavier

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

- au flux des positions du curseur, manipulé par une souris ou un pad
- au flux d'une communication internet entre un client et un serveur
- au flux résultant de l'écriture et de la lecture d'un fichier sur un disque
- au flux résultant de l'écriture et de la lecture d'une variable en mémoire
- etc...

La manipulation des flux standards passe par la bibliothèque <stdio.h>, qui est la bibliothèque standard des entrées et sorties.

De manière générale, un flux peut être de deux types :

- *textuel* : suites de caractères terminées par le caractère de fin de ligne '\n' qui forment des lignes
- *binaire* : suites de mots binaires regroupés en blocs de multiplets

Remarque : On peut aussi considérer qu'un flux textuel est un flux binaire dans le sens où du texte est représenté par des mots binaires dans un ordinateur. Cependant, considérer de manière abstraite qu'il s'agit de texte au sens humain permet de faciliter l'analyse théorique. [44]

En particulier, trois flux *textuels* sont créés par défaut au moment de l'ouverture d'un terminal :

- le flux de sortie standard représenté en C par *stdout*
- le flux d'entrée standard représenté en C par *stdin*
- 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. [45]

4.8.1 Flux d'entrée standard

On s'intéresse ici à la manipulation du flux d'entrée *stdin*. Il est possible de manipuler le flux des entrées clavier dans le terminal par la fonction scanf :

```
int x;
printf("Entrer un nombre : ");
int counter = scanf("%d", &x); // 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 (puisqu'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 :

44. Voir la théorie du langage dans la bibliothèque Minitel
45. Dont les fonctionnements techniques ne reposent pas tout à fait sur le concept de flux ceci étant dit...

```c
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);
```

4.8.2 Flux de sortie standard et flux d'erreur standard

Il est possible en C de choisir sur quel flux on veut afficher les données grâce à la fonction fprintf. Cette fonction prend un argument en plus, qui est le flux vers lequel envoyer les données :

```c
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 :

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

```c
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 et bête, est la suivante : *pourquoi ?*

La réponse, quant-à-elle, n'est pas particulièrement compliqué, mais tout de même moins évidente que la question.
Pour des raisons d'optimisation, les flux peuvent utiliser un *tampon mémoire* (*memory buffer* en anglais). En effet, l'accès en écriture à des supports comme la carte vidéo ou un disque dur est en général beaucoup plus lent que l'écriture dans la RAM. Pour cette raison, les fonctions d'affichage tentent de minimiser le nombre d'accès effectués par le programme à ces supports. Au lieu d'envoyer directement à travers le flux les données textuelles, on les stockent 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* :

```c
#include <stdlib.h>
#include <stdio.h>
#include <stdio_ext.h>

int main() {
  printf("Size of the default stdout buffer : ");
  printf("%ld\n", __fbufsize(stdout));
  fprintf(stderr, "Size of the default stderr buffer : ");
  fprintf(stderr, "%ld\n", __fbufsize(stderr));
  return EXIT_SUCCESS;
}
```

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 disfonctionnement, 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*[46].

4.8.3 Exemple pratique d'utilisation de *stderr*

On rappelle que scanf renvoie le nombre de valeurs saisies valides. Ainsi, à l'exécution du programme ci-dessus, si l'utilisateur entre des valeurs incorrectes il faut pouvoir avertir et donner un message d'erreur si la suite devient potentiellement bloquante :

```c
#include <stdlib.h>
#include <stdio.h>

int main() {
  int x, y, counter;
  printf("Entrer deux nombres : ");
  if ((counter = scanf("%d %d", &x, &y)) != 2) // appuyer sur "Entrée" pour valider la saisie
  {
    fprintf(stderr, "Erreur saisie incorrecte.\n"); // \n seulement pour lisibilité
  } else {
    printf("%d valeurs saisies.", counter);
    printf("Les deux nombres saisis sont %d et %d", x, y);
  }
  return EXIT_SUCCESS;
}
```

4.8.4 Exercices

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

46. https://fr.wikipedia.org/wiki/Historique_(informatique)

Exercice 34 (Distance Euclidienne) [11]. Écrire un programme qui demande à l'utilisateur d'entrer deux points $A = (x_A, y_A), B = (x_B, y_B) \in \mathbb{R}^2_{f64}$, les affiche et donne le carré de la distance euclidienne entre ces deux points :

$$\|A - B\|^2_{euclide} = (x_A - x_B)^2 + (y_A - y_B)^2$$

Exercice 35 (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 36 (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
Youpiii 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^{er} janvier 1970. On peut utiliser pour cela le module time.h du C :

```c
#include <time.h>

int main() {
  time_t seconds = time(NULL);
  printf("Nombre de secondes depuis le 1er janvier 1970 : %ld\n", seconds);
}
```

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 [a] entre 0 et $RAND\_MAX$ [b] inclu.

Dans la pratique :

```c
#include <stdlib.h>
#include <time.h>

int main() {
  srand(time(NULL));

  /*
  Reste du programme utilisant rand()
  */
}
```

a. C'est-à-dire le terme suivant de la suite chaotique.
b. Constante définie dans stdlib.h

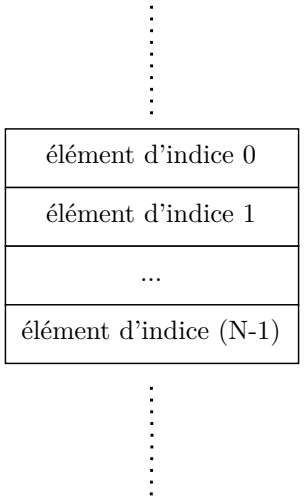
Tableaux statiques

4.9.1 Tableaux

Définition 25 : 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.

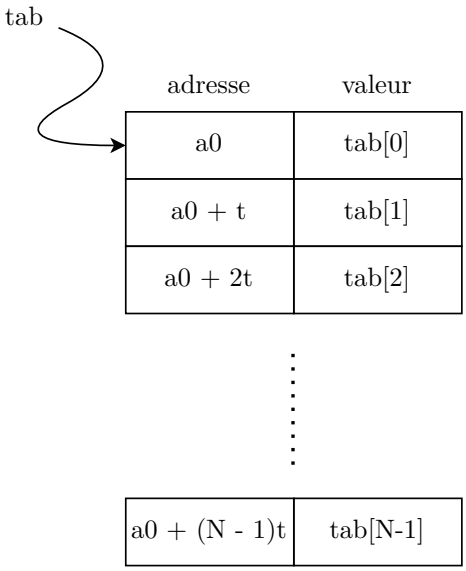
Le *tableau* est une des structures de donnée les plus simples. Il s'agit de l'équivalent informatique des N-uplets en mathématiques. Un tableau permet de stocker plusieurs variables sous une même étiquette de manière contigüe en mémoire :



Un tableau est dit *indicé*, 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 [47] (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.

4.9.2 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 4.10). Pour des raisons de compatibilité et pour éviter certains bogues, il est expressément recommandé d'éviter cette pratique douteuse.

Il est par contre tout à fait possible de préciser la taille d'un tableau par un symbole 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 :

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

```c
#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* !!!

```c
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 !!!
```

```c
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;
```

La raison, qui « expliquera » au passage l'utilisation des accolades {...} sera vue dans la section 5.7 sur la construction de littéraux dans le chapitre 5 sur les concepts avancés du langage.

4.9.3 Les tableaux statiques, kékoi 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 :

```c
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char **argv) {
  int array[3] = {1, 2, 3};

  printf("array = %p | &array = %p | &(array[0]) %p\n", array, &array, &(array[0]));

  return EXIT_SUCCESS;
}
```

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**[48]. 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 : [49]

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. [50]
- les opérateurs sizeof, typeof et _Alignof ne convertissent pas l'identifiant en pointeur mais travaillent uniquement sur le type du tableau, puisque cela est suffisant.

Cette règle amène à plusieurs propriétés.

Accès aux éléments par incrémentation du pointeur

En considérant la déclaration d'un tableau quelconque :

```
TYPE array[N] = {...};
```

L'égalité suivante est vraie pour tout indice positif strictement inférieur à N :

```
array[indice] == *(array + index);
```

En effet, tab est ici interprété comme un pointeur TYPE* vers la première case. Il est donc possible d'accéder aux cases du tableau comme pour un pointeur. En fait, il faut plutôt penser à cela dans le sens inverse : l'opération a été définie sur les pointeurs, et on retrouve cette facilité d'écriture pour manipuler les tableaux.

Remarque : L'addition est une opération commutative... donc le code suivant est tout à fait correct :

```
index[array] == array[index]; // == *(array + index)
```

48. Voir 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
49. L'appelation n'a rien d'officiel, mais je trouvais qu'elle décrivait bien l'état de fait.
50. Si ça vous paraît tordu... À moi aussi pour être honnête.

Ainsi, le code suivant est correct :

```c
#include <stdio.h>
#include <stdlib.h>

int main() {
  int some_array[10];
  for (unsigned int i = 0; i < 10; i++) {
    i[some_array] = i;
    printf("%d;", some_array[i]);
  }
  printf("\b \n");
  return EXIT_SUCCESS;
}
```

La notation n'est pas utilisée car assez illisible. [51]

Passage d'un tableau en argument à une routine

Le langage C autorise à passer un tableau en argument à une routine (voir **Exercice 38**). 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 :

```c
#include <stdio.h>
#include <stdlib.h>

void procedure(int array[]) { // donne l'impression d'être un tableau
  printf("%ld", sizeof(array));
}

int main() {
  int test[5] = {1, 2, 3, 4, 5};

  printf("%ld", sizeof(test)); // -> sizeof(int) * 5 = 20
  procedure(test); // -> sizeof(int*) = 8

  return EXIT_SUCCESS;
}
```

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 définit 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 :

51. Question d'habitude...

```c
#include <stdio.h>
#include <stdlib.h>

void incr_all(int array[], unsigned int length) {
  for (int i = 0; i < length; i++) {
    array[i]++;
  }
}

int main() {
  int test[5] = {1, 2, 3, 4, 5};

  incr_all(test, 5);
  incr_all(test, 5);

  printf("%d = %d\n", test[2], *(test + 2));

  return EXIT_SUCCESS;
}
```

L'incrémentation de tous les éléments du tableau est permanente et subsiste dans la suite du programme. Cela est logique, puisque c'est un pointeur vers les éléments du tableau qui est donné à la procédure.

Renvoi d'un tableau par une fonction

Il a déjà été dit qu'un tableau ne pouvait être renvoyé par une fonction. Cependant, un pointeur peut l'être. Une première intuition serait donc de créer un tableau dans la fonction, puis de renvoyer un pointeur vers celui-ci :

```c
#include <stdio.h>
#include <stdlib.h>

char *test_function() {
  char test_array[10] = {3, 1, 4, 1, 5, 9, 2, 6, 5, 4};
  return test_array;
}

int main() {
  // Déclaré comme un pointeur car test_function renvoie un pointeur :
  char *ret_array = test_function();

  /* la ligne suivante produit une erreur À LA COMPILATION :
  char ret_array[10] = test_function();
  */

  printf("Premier element (?) : %d\n", *ret_array);
  return EXIT_SUCCESS;
}
```

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 ![52]

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 ![53]

4.9.4 Exercices

Exercice 37 (Les routines, direction la seconde classe !) [05]. Justifier que les routines en C ne sont pas non plus des entités de première classe.

Exercice 38 (Routines classiques de manipulation de tableaux) [20]. Soit T un tableau de $l(T) \in \mathbb{N}$ éléments. On note pour $i, j \in [\![0; l(T)-1]\!]$:

- $T[i]$: 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 $k \in [\![0; l(T[i:j])-1]\!]$, 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.
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
;
Sorties : tableau A trié par effet de bord
;
$i \leftarrow 0$;
tant que $i < l(A)$ **faire**
    Trouver l'indice i_{min} du minimum de $A[i:l(A)]$;
    Échanger dans A les éléments d'indices i et $i_{min}+i$;
    $i \leftarrow i+1$;

52. 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.
53. Pour celles et ceux qui se demandent : oui, il y a une suite à cette phrase, mais ceci est une autre histoire...

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 39 (Recherche dichotomique) [12M]. On reprend les notations introduites dans l' **Exercice 38** .

1. Soit T un tableau d'entiers trié par ordre croissant. Soit $i \in [\![0; l(T)-1]\!]$.Soit $x \in \mathbb{Z}$. Montrer que :
$$x \in T \Rightarrow \begin{cases} x \leq T[i] & \Rightarrow \quad x \in T[0:i] \\ x > T[i] & \Rightarrow \quad x \in T[i:l(T)] \end{cases}$$
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 40 (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 l' **Exercice 26** , 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 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 : $\pi(x)$ est le nombre de nombres premiers inférieurs ou égaux à x. Les théorèmes de théorie des nombres peuvent a priori être utilisés librement, ou la démonstration de la dernière question passe à une difficulté 45 environ

Algorithme 8 : Algorithme du crible d'Eratosthène

Entrées : $N > 0$ un entier naturel
;
Sorties : les entiers naturels inférieurs à N
;
Initialisation
Déclarer un tableau P de N booléens;
Initialiser : $P[0] = Faux$ et $\forall i \in [\![1; l(P)-1]\!], P[i] = Vrai$;
Boucle
On pose $p := 2$;
tant que $p^2 \leq N$ **faire**
  pour $(f := p \ to \ \left\lceil \frac{N}{p} \right\rceil)$ **faire**
    $P[pf] = Faux$;
  tant que $P[p] = Faux$ **faire**
    $p \leftarrow p + 1$;

Allocation dynamique

4.10.1 Introduction au tas

Un programme informatique, lorsqu'il est chargé en mémoire, comporte un troisième espace mémoire [54] nommé *le tas* (*heap* en anglais). Il se nomme ainsi car il correspond à un tas de mémoire accessible au programmeur. [55]

Le tas est un espace mémoire de taille extensible [56] contrairement à la pile d'exécution. Il peut être aggrandit 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 contigües en mémoire, contrairement à la pile. Cela ralentit l'accès à celles-ci.
- les espaces mémoires alloués aux variables ne sont pas libérés automatiquement à la sortie des routines, **c'est au programmeur d'être responsable**

Le deuxième point est particulièrement important, puisqu'il peut être la source de graves bogues connus sous le nom de fuites de mémoire (*memory leaks* en anglais). La fuite de mémoire apparaît lorsqu'une variable est allouée dynamiquement sur le tas, et que le programmeur oublie de la libérer avant la fin de la routine. Le système d'exploitation continue de considérer que la zone mémoire est utilisée jusqu'à ce que le programme se termine. Dans le cadre d'applications conséquentes, cela peut être vraiment problématique. [57] Par ailleurs, libérer incorrectement la mémoire peut amener à des failles de sécurité importantes.

Un autre bogue très récurrent qui empêche totalement le fonctionnement du programme est l'accès à une variable dont l'espace mémoire a été libéré : s'ensuite l'erreur de segmentation, de code -11, provoqué par le système d'exploitation qui perçoit cela comme une attaque du système ! [58] Ce bogue est rencontré en quasi-permanence chez les débutants, dès lors que des structures complexes sont manipulées en mémoire. Il arrive aussi très fréquemment [59] chez les programmeurs expérimentés. En effet, il est d'une facilité extraordinaire de rencontrer ce bogue dès lors que la programmation manque un tout soit peu de rigueur. [60]

54. Les deux premiers sont :
    - le code
    - la pile d'exécution

55. 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'appelation ne fait pas référence à la structure de donnée homonyme

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

57. 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 !

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

59. Quoi que ceux-ci en disent ;)

60. 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"

4.10.2 Les routines malloc et free

Pour gérer "soi-même" [61] de la mémoire sur le tas, deux fonctions de la bibliothèque *stdlib.h* fournissent les fonctionnalités suivantes :

- allouer un espace mémoire, grâce à malloc [62]
- libérer un espace mémoire précédemment alloué, grâce à free

Le détail du fonctionnement interne des deux fonctions ne sera pas explicité. [63]

Voici les prototypes de ces deux fonctions issus de la documentation officielle :

```c
#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.

```c
#include <stdio.h> // printf
#include <stdlib.h> // EXIT_SUCCESS, malloc et free

int main() {
  int* array = (int*)malloc(sizeof(int) * 5); // Alloue 4 * 5 = 20 octets.

  for (int i = 0; i < 5; i++) {
    printf("%d\n", *(array + i));
  }

  free(array); // Autorise de futures allocations à cette adresse
  array = NULL; // "oublie" l'ancienne position du tableau en mémoire
  return EXIT_SUCCESS;
}
```

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 :

61. C'est les routines qui font le gros du boulot hein ! Faudrait pas se mettre à voler par les chevilles x)
62. Pour *Memory Allocation*
63. Mais pour le plaisir : https://wiki.osdev.org/Memory_management, 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/

```
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 !

```
#include <stdio.h> // printf
#include <stdlib.h> // EXIT_SUCCESS, malloc et free

int* make_array(unsigned int n) {
  int *array = (int*)malloc(sizeof(int) * n); // Alloue (4 * n) octets
  for (unsigned int i = 0; i < n; i++) {
    array[i] = i;
  }
  return array;
}

int main() {
  int* array = make_array(10); //  tableau de 4 * 10 = 40 octets

  for (unsigned int i = 0; i < 10; i++) {
    printf("%d\n", array[i]);
  }

  free(array); // la mémoire doit toujours être libérée
  array = NULL; // et le pointeur réinitialisé
  return EXIT_SUCCESS;
}
```

4.10.3 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 :

```c
#include <stdio.h>
#include <stdlib.h>

#define USER_SIZEOF_DATA_STRUCTURE ...
#define IS_ADMIN_OFFSET ... // value < USER_SIZEOF_DATA_STRUCTURE

int main() {
  char *guest = NULL, *admin = NULL;

  // les informations d'un premier utilisateur sont stockés dans un tableau de données
  guest = malloc(USER_SIZEOF_DATA_STRUCTURE);
  // un des octets stocke les droits d'utilisateur (1 si admin, 0 sinon)
  guest[IS_ADMIN_OFFSET] = 0; // pas admin par défaut

  // du code

  // quelque part :
  free(guest);
  // guest pointe toujours vers sa zone mémoire !!!

  // encore du code

  // et puis autre part :
  admin = malloc(USER_SIZEOF_DATA_STRUCTURE);
  admin[IS_ADMIN_OFFSET] = 1;
  // (guest == admin) est vrai ici !!!!!!

  // encore et toujours du code

  if (guest == NULL || guest[IS_ADMIN_OFFSET] == 0) {
    printf("Cette section nécessite les droits d'administrateur !\n");
    return EXIT_FAILURE;
  }

  printf("Acces a la zone d'administration autorise\n");

  // du code administrateur

  free(admin);
  return EXIT_SUCCESS;
}
```

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

```
user@computer ~/working_directory> ./main
Acces a la zone d'administration autorise
```

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 :

```c
void some_proc() {
  // code
  char* t = malloc(N);
  // 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.

4.10.4 Exercices

Exercice 41 (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 la partie 1.3.1)

Tableaux multidimensionnels

Maintenant que sont acquis les concepts de tableaux statiques et dynamiques, il devient possible de contruire la structure de donnée un peu plus complexe qu'est le tableau multidimensionnel, au sens informatique de tableaux imbriquées. Voici l'exemple d'un tableau 2-dimensionnel quelconque :

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

Remarque : Un tableau 2-dimensionnel n'est pas nécessairement rectangulaire, puisque chaque ligne peut être de taille quelconque. Par exemple, on peut imaginer le tableau 2-dimensionnel suivant :

$$\begin{bmatrix} 1 & & \\ 2 & 3 & \\ 4 & 5 & 6 \\ 7 & 8 & \\ 9 & & \end{bmatrix} \tag{4.1}$$

4.11.1 Tableaux multidimensionnels statiques

Commençons par le moins utile. On ne peut définir statiquement que des tableaux multidimensionnels rectangulaire, c'est-à-dire dont chacune des lignes possède autant de colonnes que les autres.

Dans le cas statique, la déclaration d'un tableau D-dimensionnel de taille $N_1 \times N_2 \cdots \times N_D$ revient exactement à allouer statiquement en mémoire un "grand" tableau de $N_1 \times N_2 \cdots \times N_D$ cases. Et c'est seulement lors de l'indexation du tableau que l'aspect multidimensionnel ressort.

La syntaxe pour déclarer un tableau D-dimensionnel de taille $N_1 \times N_2 \cdots \times N_D$ est la suivante :

```
#define N1 ...
#define N2 ...
...
#define ND ...

TYPE array[N1][N2]...[ND]; // D 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

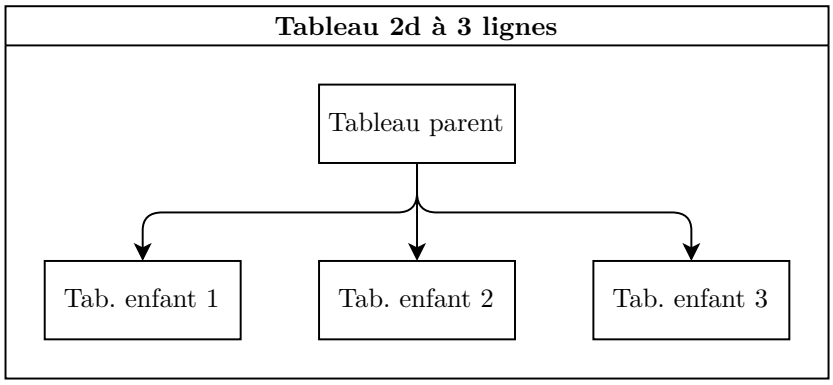
Limitations/Inconvénients des tableaux multidimensionnels statiques :

- 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.

4.11.2 Tableaux multidimensionnels dynamiques

L'idée de l'allocation dynamique de tableaux multidimensionnels est la suivante : on veut allouer dynamiquement un tableau qui va contenir des sous-tableaux que l'on va également allouer dynamiquement.



Si les sous-tableaux contiennent des valeurs de type TYPE, alors chacun d'entre-eux sera de type TYPE*. Par conséquent, le tableau parent doit être de type TYPE**. On en déduit le code suivant pour définir un tableau rectangulaire à N_{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 4.1, on peut imaginer le code suivant :

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

int min(int a, int b) {
  return (a > b) ? b : a;
}

int main() {
  int** triangle = (int**)(malloc(sizeof(int*) * 5));
  for (int i = 0; i < 5; i++) {
    triangle[i] = (int*)(malloc(sizeof(int) * (1 + min(5 - 1 - i, i))));

    // initialise le tableau avec des 0
    for (int j = 0, j < (1 + min(5 - 1 - i, i)); j++) {
      triangle[i][j] = 0;
    }
  }
  free(triangle);
  triangle = NULL; // pas obligatoire car sortie de routine
  return EXIT_SUCCESS;
}
```

4.11.3 Exercices

Exercice 42 (Afficher un tableau $2d$) [10]. Écrire un programme (et pas une routine) qui affiche les éléments d'un tableau $2d$ d'entiers.

Exercice 43 (Affichage du triangle) [13]. Compléter le programme ci-dessus pour remplir le

tableau 4.1. Écrire ensuite une procédure print_triangle(int **triangle, int n_lines) ; qui affiche ce tableau (où n_lines désigne le nombre de lignes du tableau)

Remarque : L'exercice ci-contre met en évidence la possibilité de passer en argument des tableaux multidimensionnels.

Exercice 44 (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 $(L_i)_{i\in[\![0;n[\![}$ et chacune de ces lignes est un tableau de valeurs typées :

$$\begin{cases} & mat & = & (L_0, L_1, \ldots, L_{n-1}) \\ \forall i \in [\![0; n[\![& L_i & = & (a_{i,0}, \ldots, a_{i,m-1}) \end{cases}$$

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 $b \in \mathcal{M}_{k,m}$

Structures

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 :

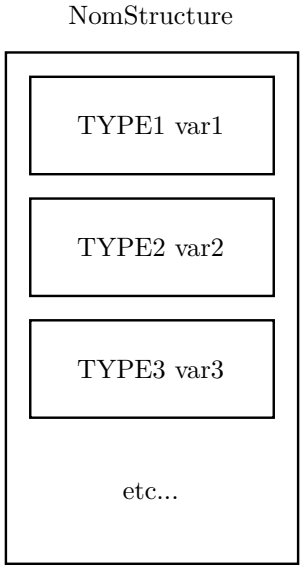
- la modélisation d'une lentille dans une simulation physique par une taille, la modélisation de sa forme par des courbes paramétriques, les propriétés du matériau, etc...
- le bouton d'une application par une taille, une position dans l'espace, une couleur, un texte, etc...
- un personnage d'un jeu 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^e 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 les structures entrent en jeu.

Une structure permet de définir un type complexe. C'est un moyen de stocker dans une seule entité plusieurs types d'informations. Chacune de ces informations possédera sa propre étiquette au sein de cette entité, ainsi que son propre type. C'est un peu comme une boîte qui contient plusieurs variables :



La syntaxe de définition d'une structure en C est la suivante :

```
struct NomStructure { // 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 *membre*, de la structure ne peut être initialisée. Pour utiliser une structure, il faut au final créer des variables dont le type est cette structure :

```
struct NomStructure x; // 'x' est créé sans membres initialisés.
```

Il est possible d'y penser un peu comme à la construction d'une maison : on commence par dessiner le plan de la maison et une fois cela fait il suffit de fabriquer plein de maisons grâce à ce plan.

On peut ensuite accéder aux membres d'une structure via un nouvel opérateur binaire, l'opérateur d'accès à un membre noté "." :

```
x.var1 = ...; // accède au membre 'var1' de 'x'
x.var2 = ...; // accède au membre 'var2' de 'x'
x.var3 = ...; // accède au membre 'var3' de 'x'
// etc...
```

Un exemple :

```c
#include <stdio.h>
#include <stdlib.h>

struct Point3D {
  double x;
  double y;
  double z;
};

int main() {
  struct Point3D p;
  p.x = 1.0;
  p.y = 2.0;
  p.z = -3.0;
  printf("p = (%lf, %lf, %lf)\n", p.x, p.y, p.z);

  return EXIT_SUCCESS;
}
```

Remarque 1 : Une structure ne peut contenir au maximum que 127 membres.

Remarque 2 : Au contraire des tableaux statiques, toute structure définie est considérée comme une entité de première classe. Ainsi, il est possible d'écrire des routines dont les paramètres sont des structures, et qui prennent en argument des structures identiques :

```c
#include <stdio.h>
#include <stdlib.h>

struct Point3D {
  double x;
  double y;
  double z;
};

void afficher(struct Point3D p) {
  printf("(%lf, %lf, %lf)\n", p.x, p.y, p.z);
}

int main() {
  struct Point3D p;
  p.x = 1.0;
  p.y = 2.0;
  p.z = -3.0;
  afficher(p);

  return EXIT_SUCCESS;
}
```

Remarque : Il est aussi possible qu'un membre d'une structure soit lui-même une structure :

```c
#include <stdio.h>
#include <stdlib.h>
```

```
3
4  struct Couple {
5    double x;
6    double y;
7  };
8
9  struct Line {
10   struct Couple point;
11   struct Couple direction;
12 };
13
14 // or
15
16 struct Line {
17   struct Couple {
18     double x;
19     double y;
20   } point;
21   struct Couple direction;
22 };
23
24 int main() {
25   struct Line l;
26   // . 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 }
```

4.12.1 Tableaux de structures

Comme les structures définies sont considérés comme des entités de première classe, 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 :

```
...

int main() {
  int i = ...;
  struct Time* running_times = malloc(sizeof(struct Time) * 10);
  running_times[i].hour = ...;
  return EXIT_SUCCESS;
}
```

Il est ainsi possible de travailler sur des pointeurs de structure :

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

struct Something {
  int a;
  char b;
};

int function(struct Something* s_ptr) {
  (*s_ptr).a += (*s_ptr).b;
  return (*s_ptr).a;
}

int main() {
  struct Something s;
  s.a = 0;
  s.b = 1;
  for (int i = 0; i < 10; i++) {
    s.b = function(&s) / s.b;
  }
  printf("%d\n", s.a);
  return EXIT_SUCCESS;
}
```

On observe que l'opérateur d'accès à un membre est prioritaire sur l'opérateur d'indirection $*$[64]. Oublier de parenthéser amène à une erreur puisque ptr.member n'existe pas. En effet, un pointeur n'est qu'une adresse vers la structure. Il n'a donc pas de membres.
La notation apparaît cependant très lourde à écrire, alors une notation purement facilitatrice existe en C, la flèche :

64. En effet, l'opérateur d'indirection ne fait qu'ajouter un décalage à l'adresse de la variable structurée pour accéder au membre de la structure. Ce calcul est effectué par le compilateur. Pour cette raison, l'opérateur d'accès à un membre fait partie des opérateurs les plus prioritaires du langage.

```c
struct MyStruct {
  int a;
  int b;
};

struct MyStruct *s = ...;

printf("a = %d, b = %d\n", s->a, s->b);
```

```c
struct MyStruct {
  int a;
  int b;
};

struct MyStruct *s = ...;

printf("a = %d, b = %d\n", (*s).a, (*s).b);
```

4.12.2 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 :

```c
struct MyStruct {
  int a;
  char b;
  void *c; // pointeur quelconque
  long double d;
};

 // Initialisation séquentielle
struct MyStruct s = {
  2<<20,
  6,
  NULL,
  3.141592653589
};
```

```c
struct MyStruct {
  int a;
  char b;
  void *c; // pointeur quelconque
  long double d;
};

// Initialisation sélective
struct MyStruct s = {
  .d =  3.141592653589,
  .a = 2<<20,
  .c = NULL,
  .b = 6
};
```

L'initialisation séquentielle suit exactement l'ordre des membres de la structure.
L'initialisation sélective permet de choisir très exactement quel membre reçoit quelle valeur.

Remarque : Les retours à la ligne sont optionnels et ne servent qu'à la lisibilité. Le code suivant est valide :

```c
struct Point {
  double x;
  double y;
};

struct Point p = {.y = 4.5, 3.2}; // Pointe vers x = 3.2, y = 4.5
```

Il est parfaitement possible de mélanger les initialisations séquentielles et sélectives. Dans un tel cas, l'initialisation séquentielle reprend au dernier membre désigné par une initialisation sélective. Partant, le code suivant initialise le membre a à 1 et le membre d à 30.560.

```c
struct MyStruct s = {1, .d = 30.560};
```

Alors que le code ci-dessous initialise le membre c à NULL, le membre d à 48, le membre b à 0 et le membre c à $\&s$.

```c
struct MyStruct s = {.c = NULL, 48, .b = 0, &s}; // d non initialisé
```

4.12.3 Exercices

Exercice 45 (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 44** pour qu'elles utilisent la structure struct Matrix.

Exercice 46 (Listes chainées) [27]. L'objectif de cet exercice est d'introduire une nouvelle manière de structurer les données. La structure de données dite de *liste chaînée* permet de stocker des données non adjacentes en mémoire, au contraire des tableaux. Les *listes chaînées* ont l'avantage inédit par rapport aux tableaux de pouvoir changer de taille au cours de l'exécution du programme. Il devient possible d'ajouter ou de supprimer des éléments *dynamiquement*. On peut ainsi imaginer une liste d'items d'un utilisateur qui pourrait s'accroitre indéfiniment [65], comme un carnet d'adresses par exemple.

On commence par construire la structure permettant de stocker un *noeud* de la liste, c'est-à-dire un de ses éléments. La valeur du noeud est la valeur de l'élément de la liste :

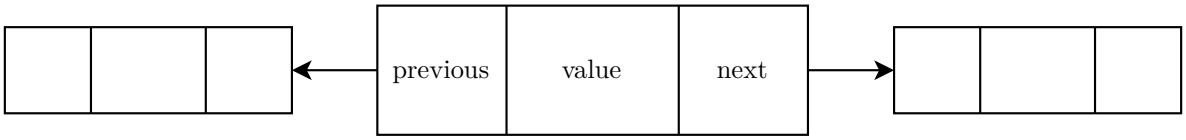


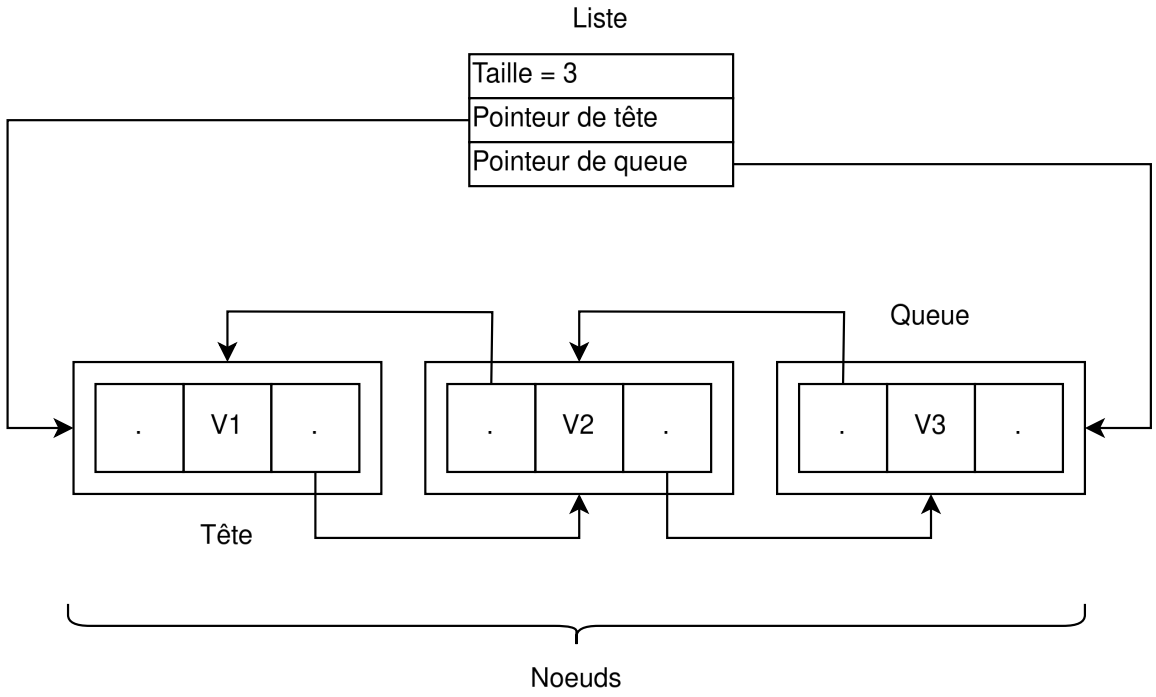
FIGURE 4.2 – Noeud d'une liste

L'idée fondamentale des listes chaînées 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) :

65. Dans la limite de la mémoire de l'ordinateur



4. Écrire une fonction `struct LinkedList* linkedlist_new()` ; qui alloue dynamiquement une structure `struct LinkedList* lst` initialisée telle que :
   - $lst \rightarrow head$ et $lst \rightarrow previous$ ne pointent sur rien
   - $lst \rightarrow length = 0$
5. Écrire une fonction `char linkedlist_is_empty(struct LinkedList *l)` ; qui renvoie 1 si la liste est vide, et renvoie 0 sinon.
6. Écrire une procédure `void linkedlist_push_on_head(struct LinkedList *l, int value)` ; qui crée un noeud de valeur *value* et l'insère *en tête de liste*, c'est-à-dire qu'après insertion la valeur de la *tête* sera *value*. On pensera à traiter séparément le cas où la liste est vide (c'est-à-dire $lst \rightarrow length = 0$). En effet, on rappelle que si $lst \rightarrow head$ ne pointe sur rien, le considérer comme un noeud provoque une erreur du fait de l'accès à une zone mémoire non autorisée. Par ailleurs, si la liste contient un unique élément, la *tête* est égale à la *queue*.

   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.

Modulation et entêtes

Dans le cas du développement d'une application complexe, il arrive très souvent que la division d'un programme informatique en routines dans un unique fichier soit insuffisante. En effet, ces routines peuvent couvrir des domaines très différents. Dans le cas d'un jeu-vidéo, cela peut aller des routines pour s'occuper de l'inventaire d'un personnage jusqu'aux routines s'occupant de la physique des objets du jeu en passant par les routines pour ouvrir, lire et écrire les fichiers de sauvegarde. Toutes ces routines très différentes ont besoins de structures de données différentes qui peuvent être accompagnés elles-mêmes d'une multitude de routines de manipulation. Bref... Autant dire que mettre tout ça dans un seul et unique fichier de code, ça risque d'être un poil touffu. En particulier, la gestion de l'évolution d'un unique aspect du code source indépendamment du reste devient extrêmement confuse, sans parler de l'organisation en équipes quand les travaux de toute une équipe peuvent interférer.

La solution à cela est simple : écrire dans des fichiers différents les routines qui s'occupent d'aspects différents du programme/de l'application développé(e).

Pour cela, on distingue en langage C deux catégories de fichiers :

- les fichiers d'entêtes (*headers* en anglais) d'extensions *.h*
- les fichiers de code source d'extensions *.c*

Définition 26 : Module et programmation modulaire

Un module est la donnée d'un fichier d'entête et d'un fichier de code source. Un programme informatique *modulaire* est un programme constitué de plusieurs modules. On appelle *programmation modulaire* le développement d'un programme par modules. Un module est un composant du programme qui peut lui-même utiliser d'autres modules pour son bon fonctionnement.

On peut penser aux modules informatiques comme aux modules d'un vaisseau spatial. Chacun est indépendant dans son fonctionnement interne mais tous les modules sont interdépendants d'un point de vue abstrait puisqu'ils sont chacun constitutif du programme et ne peuvent pas résoudre le problème individuellement. [66]

```
user@computer ~/working_directory> ls
main.c module1.c module1.h module2.c module2.h etc...
user@computer ~/working_directory>
```

4.13.1 Fichier d'entête

L'objectif du fichier d'entête d'un module est de donner au compilateur toutes les informations relatives au module qui ne sont pas le code source des routines proprement dit. Il s'agit donc :

- des instructions préprocesseurs (notamment les #define et #include)
- des 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. En effet, il s'agit du fichier que l'on inclut par la directive de préprocesseur #include :

66. Si cela est le cas, le programme n'est plus modulaire. On peut alors s'interroger sur la qualité de la conception.

```
#include "chemin/vers/mon/module.h"
```

Le fichier peut se situer théoriquement n'importe où dans le système de fichiers du système d'exploitation.

4.13.2 Fichier de code source

Le fichier de code source, ou simplement fichier source, fournit le code des fonctions décrites par le fichier d'entête. Il peut également décrire des fonctions internes au module non présentes dans le fichier d'entête, qui ne pourront donc pas être utilisés en externe du module.

L'unique particularité du fichier source est de devoir inclure, par la directive #include, le fichier d'entête auquel il est associé :

```
#include "chemin/vers/mon/module.h"

/*
Code source des fonctions internes au module,
non accessibles hors du module
*/

// Code source des fonctions déclarées dans l'entête
```

4.13.3 Un exemple simple

On se propose ici de coder un module *vec2* qui contient des outils pour la manipulation de vecteurs en deux dimensions. [67]

On suppose se situer dans un répertoire de travail <some_path>. On considère trois fichiers dans ce répertoire :

- "main.c"
- "vec2.c"
- "vec2.h"

On nomme identiquement le fichier d'entête et le fichier source d'un module pour des raisons de lisibilités. [68] 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
```

67. 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.

68. 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.

```c
struct Vec2 {
  double x;
  double y;
};

// prototypes de routines

struct Vec2 vec2_new(double x, double y);
struct Vec2 vec2_copy(struct Vec2 p);

void vec2_display(struct Vec2 p);
struct Vec2 vec2_scalar_multiply(struct Vec2 p, double k);
struct Vec2 vec2_add(struct Vec2 p1, struct Vec2 p2);
struct Vec2 vec2_sub(struct Vec2 p1, struct Vec2 p2);
```

"vec2.h" décrit bien l'interface du module qui sera implanté réellement dans le fichier source :

vec2.c

```c
#include "vec2.h" // Nécessaire pour accéder à l'interface (décrite dans l'entête)

// implantation, dans un ordre quelconque, des routines

struct Vec2 vec2_new(double x, double y) {
  struct Vec2 v = {x, y};
  return v;
}
struct Vec2 vec2_copy(struct Vec2 p) {
  struct Vec2 v = {p.x, p.y};
  return v;
}

void vec2_display(struct Vec2 p) {
  printf("(%lf, %lf)", p.x, p.y);
}

struct Vec2 vec2_scalar_multiply(struct Vec2 p, double k) {
  struct Vec2 v = {p.x * k, p.y * k};
  return v;
}

struct Vec2 vec2_add(struct Vec2 p1, struct Vec2 p2) {
  struct Vec2 v = {p1.x + p2.x, p1.y + p2.y};
  return v;
}

struct Vec2 vec2_sub(struct Vec2 p1, struct Vec2 p2) {
  struct Vec2 v = {p1.x - p2.x, p1.y - p2.y};
  return v;
}
```

Finalement, le fichier *"main.c"* peut inclure le module *vec2* pour utiliser les structures et routines proposés :

main.c

```c
#include <stdlib.h>

#include "vec2.h" // Accède à l'interface décrite
// -> permet d'utiliser les routines et structures décrites

int main() {
  struct Vec2 v = vec2_new(1.414, 3.14159);

  vec2_display(v);

  return EXIT_SUCCESS;
}
```

4.13.4 Compilation modulaire

La compilation modulaire introduit quelques notions plus complexes relatives à la compilation d'un programme. En effet, compiler "naïvement" le programme comme à l'habitude en ne précisant que le fichier *"main.c"* conduit à un dramatique message d'erreur, passablement incompréhensible sans plus d'informations :

```
user@computer ~/working_directory> ls # ou dir sous Windows
main.c vec2.c vec2.h
user@computer ~/working_directory> gcc main.c -o main
# ou chemin/vers/gcc.exe main.c -o main.exe sous Windows
/usr/bin/ld : /tmp/ccCAnFoX.o : in function "main" :
main.c:(.text+0x25) : undefined reference to        "vec2_new"
/usr/bin/ : 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 [12]
4. l'édition des liens [12][10]

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 [69], ou de prendre en compte les interfaces décrites par les modules inclusent par #include, de remplacer les constantes définies par #define, etc...

69. Par exemple graĉe à #ifdef

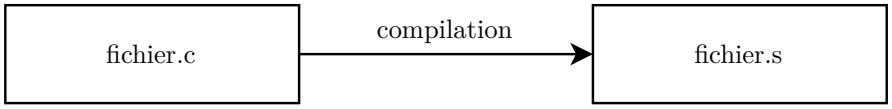
Compilation

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 [70]
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
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 [71]

À ce niveau, est associé à chaque fichier source un fichier d'extension *.s* qui correspond au langage assembleur *gas* utilisé par *gcc* [72]



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*. [73]

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>
```

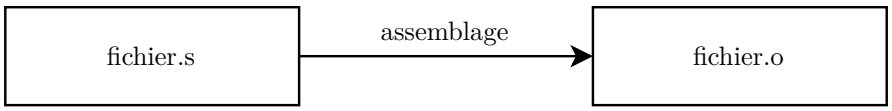
qui peut être lu par un éditeur de texte classique. [74]

Assemblage

gcc fait appel à un programme tiers nommé *assembleur* qui génère le code machine associé au code assembleur. [75].

70. 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
71. 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.
72. Pour plus de détails : https://en.wikipedia.org/wiki/GNU_Assembler
73. 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.
74. Assez évident puisqu'un programme assembleur est un texte... Enfin je dis ça...
75. 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.

À ce niveau, est associé à chaque fichier source un fichier *objet* (généralement d'extension *.o*). Ce fichier contient le code machine qui sera exécuté par l'ordinateur, ainsi que les symboles utiles pour l'édition de liens que sont les noms de fonctions par exemple.



On peut générer le fichier objet d'un programme grâce à l'argument `-c` :

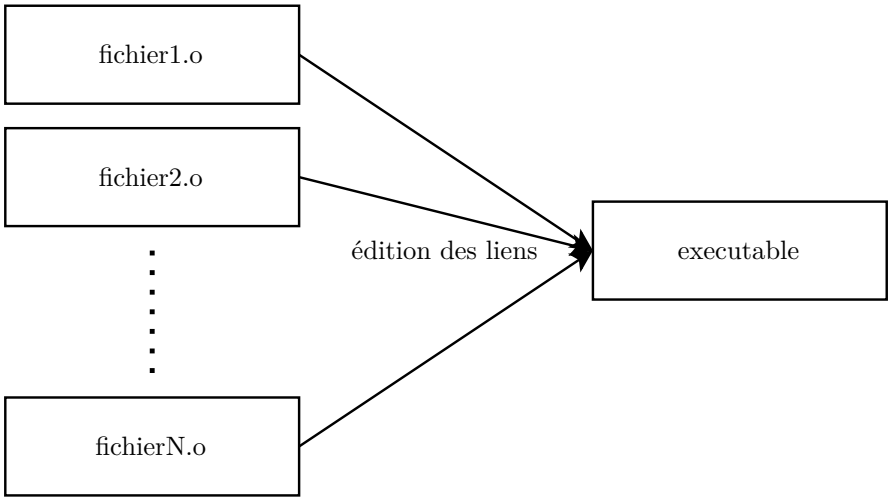
```
user@computer ~/working_directory> ls
main.c vec2.c vec2.h
user@computer ~/working_directory> gcc main.c -c
user@computer ~/working_directory> ls
main.c main.o vec2.c vec2.h
user@computer ~/working_directory>
```

Cette fois-ci, le fichier objet devient assez illisible par un éditeur de texte puisqu'il contient du code machine.

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 dûes 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_diplay* 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. [76]

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 [77]. Certains outils d'automatisation simplifient ces opérations et permettent de gagner en productivité (voir la section 6.1 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.

4.13.5 Problème des chaînes d'inclusions

Une erreur très courante chez les débutants en programmation modulaire est le problème de la *double inclusion*. Il s'agit d'un bug lors de l'exécution des directives préprocesseurs provoqué par les chaînes d'inclusion. On l'illustre par l'exemple suivant, avec deux modules :

- un module *module1*
- un module *module2*

Le *module1* est dépendant du *module2* et le programme principal est lui-même dépendant des *module1* et *module2* :

module1.h

76. Et même deux en l'occurrence.
77. Qui est en général beaucoup plus complexe, avec l'utilisation de plus de paramètres.

```c
struct SomeStructure {
  int thing;
};
```

module2.h

```c
#include "module1.h" // Accède à l'interface de "module1"

/*
Ici : Une interface quelconque
*/
```

main.c

```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

```c
#include "module2.h"
struct SomeStructure {
  int thing;
};
```

module2.h

```c
#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.

4.13.6 Garde-fous

On introduit de nouvelles directives de préprocesseur ici :

- #ifdef et #ifndef
- #else
- #endif

Il s'agit de directives de préprocesseurs qui permettent d'ignorer certaines sections du code lors de la compilation. L'utilisation est la suivante :

```
#ifdef SOME_CONST
/*
Code exécuté si SOME_CONST est définie
*/
#else // pas obligatoire
/*
Code exécuté si SOME_CONST n'est pas définie
*/
#endif
```

```
#ifndef SOME_CONST
/*
Code exécuté si SOME_CONST n'est pas définie
*/
#else // pas obligatoire
/*
Code exécuté si SOME_CONST est définie
*/
#endif
```

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
```

```c
#include "module1.h" // Accède à l'interface de "module1"

void array_sort(int array[], unsigned int length);

#endif
```

4.13.7 Exercices

Exercice 47 (Un module de listes chaînées) [13]. Écrire un module qui contiennent les structures et fonctions définies dans l' **Exercice 46** .

Chaînes de caractères

Rappel : La représentation ASCII est détaillé en table 1.5

4.14.1 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. [78]

4.14.2 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. [79]

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. [80]

La manipulation de caractères et de chaînes de caractères suit une nomenclature précise :

- on représente un caractère entre guillements simples : 'a', 'b', '=', '!', ' :', etc...
- on représente une chaîne de caractères entre guillemets doubles : "Hello World !", "42 is the answer", etc...

78. 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 ;)
79. Autant dire qu'en C, il n'y a que le strict minimum hein !
80. Coïncidence ? Je ne crois pas.

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 :

```
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 ?*[81], le langage C pose par convention que la fin d'une chaîne de caractères est indiqué par le caractère spécial $\backslash 0 \equiv$ *[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
```

4.14.3 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 !");
```

81. 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 ?*

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 [82], 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 guillements, plus un 0 pour finir la chaîne.
Ainsi, il est possible d'écrire simplement :

```
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 manipulé par printf, c'est-à-dire la première.

4.14.4 Exercices

Exercice 48 (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* ≡ *Not A Number* et $inf \equiv \infty$, que l'on obtenir par 0/0 et 1/0. Par ailleurs, ces deux "nombres" peuvent être utilisés dans la calculatrice.

Indication : un caractère est un nombre, il peut donc être utilisé dans un aiguillage 'switch-case'

Exercice 49 (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 :

82. Étonnant n'est-ce pas ? Qui l'aurait crû ?

- “42” : 42
- “-53” : −53
- “ +74TEXTE” : +74

Note : *Aucune routine de la bibliothèque standard ne doit être utilisée dans* atoi*!!!*[83]

Les flux de fichiers

Recommandation : revoir le vocabulaire sur les fichiers de la partie 1 du cours.

La manipulation de fichiers est vitale pour l'écriture de programmes persistents, c'est-à-dire dont certains états peuvent durer même quand le programme n'est pas en cours d'exécution. On peut par exemple penser à des fichiers de configuration de logiciels. La sauvegarde de certains états particuliers permet aussi au logiciel de traiter des données différentes et de stocker de manière persistente les résultats du/des traitements. On peut penser à tous les fichiers édités par des éditeurs de texte, des sauvegardes de jeux-vidéos, ou de manière beaucoup plus générique à la simple existence de l'entièreté des fichiers sur un disque dur qui permettent de donner une “mémoire longue durée” à l'ordinateur pour qu'il puisse s'exécuter en conservant une part des traitements effectuées.

La bibliothèque standard vous fournit différentes fonctions pour manipuler les fichiers, toutes déclarées dans l'en-tête <*stdio.h*>. Toutefois, celles-ci manipulent non pas des fichiers, mais des flux de données en provenance ou à destination de fichiers.

Cela permet notamment d'effectuer à nouveau des optimisations grâce aux tampons, et de rediriger des flux. On peut par exemple imaginer rediriger un flux standard de sortie vers un fichier pour écrire un *log*[84], ou rediriger le flux de lecture d'un fichier vers la console (pour en afficher le contenu par exemple).

4.15.1 Droits des fichiers

Cette sous-section n'est destiné qu'aux utilisateurs des systèmes Unix comme Linux. En effet, sous Windows, le système est légèrement différent bien que par certains aspects plus proche qu'il n'y paraît. Il ne sera pas traité. [85] Le but est simplement de comprendre l'existence de droits. Pour plus de détails, voir https://www.linuxtricks.fr/wiki/droits-sous-linux-utilisateurs-groupes-permissions.

Les fichiers sous Linux ont 3 types de droits :

- lecture (noté *r* comme *read* en anglais) : possibilité de lire le contenu du fichier
- écriture (noté *w* comme *write* en anglais) : possibilité de modifier le contenu du fichier
- 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 :

83. Trop facile sinon...
84. https://fr.wikipedia.org/wiki/Historique_(informatique)
85. Pour plus d'informations : https://learn.microsoft.com/en-us/windows/security/identity-protection/access-control/access-control

- a un propriétaire [86]
- 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) [87]
2. les utilisateurs du groupe du fichier (noté g comme *group* en anglais)
3. les autres utilisateurs (noté o comme *others* en anglais)

Chaque fichier possède donc en données supplémentaires les droits de r, w et x pour chacune de ces catégories.
Pour observer les droits des fichiers on peut utiliser l'argument *-l* (pour *long list*) de la commande *ls* dans un terminal Linux :

```
user@computer ~/working_directory> ls -la
total 20
-rwxrwxr-x 1 user user 15960 sept. 21 18:38 main
-rw-rw-r-- 1 user user   105 sept. 21 18:37 main.c
```

Petit apparté : 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*. 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 :

$$\underbrace{-}_{\text{type}}\ \underbrace{---}_{\text{propriétaire}}\ \underbrace{---}_{\text{groupe}}\ \underbrace{---}_{\text{autres}}$$

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 :

$$-rwxrw----$$

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

86. Il ne s'agit pas nécessairement du créateur du fichier puisque la propriété peut être transmise
87. Appeler "utilisateur" le propriétaire est au mieux vague, au pire dénué de sens. Mais bon, c'est comme ça.

Pour la suite, il faut s'assurer que les fichiers manipulés par un programme C ont bien les droits nécessaires. L'utilisateur considéré pour cela est l'utilisateur qui a exécuté le programme. On peut *uniquement pour tester les programmes C dans le cadre de ce cours* ajouter l'intégralité des droits de lecture et d'écriture à un fichier existant par la commande :

```
user@computer ~/working_directory> chmod a+rw somefilename
user@computer ~/working_directory>
```

4.15.2 Les flux de fichiers en C

Pour manipuler des fichiers en C, il faut fondamentalement quatre routines :

- une routine d'ouverture/création de flux
- une routine d'écriture dans un flux
- une routine de lecture d'un flux
- une routine de fermeture d'un flux

Heureusement, <*stdio.h*> fournit toutes ces routines :

- fopen pour ouvrir/créer un flux de fichier
- 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.[88]

Un classique de l'ISMIN

Rien ne vaut un exemple à ce stade.[89] 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 :

```c
#include <stdio.h>
#include <stdlib.h>

struct Person {
  char name[50];
  long int number;
};

struct Person person_make() {
  struct Person p;
  printf("Nom : ");
  scanf("%s", p.name);
  printf("Numero : ");
  scanf("%ld", &(p.number));
  return p;
}

void person_add_to_file(const char *pathname, struct Person p) {
  FILE *fd = fopen(pathname, "ab");
  if (fd == NULL) {
```

88. EOF est une constante définie dans <*stdio.h*> et vaut généralement −1
89. 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.

```c
    fprintf(stderr, "Erreur d'ouverture du fichier %s\n", pathname);
    return;
  }
  if (!fwrite(&p, sizeof(struct Person), 1, fd)) {
    fprintf(stderr, "Erreur d'ecriture dans annuaire.data\n");
  }
  fclose(fd); // nécessaire : permet au fichier d'être réouvert plus tard
}

void person_print_numbers_from_file(const char *pathname) {
  FILE *fd = fopen(pathname, "rb");
  if (fd == NULL) {
    fprintf(stderr, "Erreur d'ouverture du fichier %s\n", pathname);
    return;
  }
  struct Person p;
  while (fread(&p, sizeof(struct Person), 1, fd) == 1) {
    printf("%s : %ld\n", p.name, p.number);
  }
  fclose(fd); // nécessaire : permet au fichier d'être réouvert plus tard
}

int main() {
  struct Person p = person_make();
  person_add_to_file("annuaire.data", p);
  person_print_numbers_from_file("annuaire.data");
  return EXIT_SUCCESS;
}
```

4.15.3 Rediriger un flux standard vers un fichier

La redirection de flux a été évoquée précédemment et on montre ici comment la réaliser.

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 :

```c
#include <stdio.h>
#include <stdlib.h>

int main() {
  freopen("log.txt", "w", stderr);
  fprintf(stderr, "Erreur dans le log !!!");
  fclose(stderr);
  return EXIT_SUCCESS;
}
```

4.15.4 Exercices

Voir les annales d'*Algorithme et Programmation 1*. Toutes les notions nécessaires ont été abordées. Une correction par Minitel (et par les profs (beurk !)) sera proposé aussi vite que possible sur le NextCloud.

CHAPITRE

5

CONCEPTS AVANCÉS

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 [1].

5.1.1 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 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 :

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

unsigned int _assign_count = 0;
#define ASSIGN(X) (_assign_count++, X)

int main(int argc, char **argv) {
  int a = ASSIGN(42);
  int b = ASSIGN(64);
  b = ASSIGN(a + b);
  int c = ASSIGN(_assign_count*b + a);
  printf("Nombre d'assignations : %u\n", _assign_count);
  return EXIT_SUCCESS;
}
```

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. L'opérateur d'assignation est en effet prioritaire sur la virgule.

```
int x = 1, 2, 3;
```

est incorrect car équivalent à :

```
(((int x = 1), 2), 3);
```

qui n'a pas de sens.

Expression à partir d'un bloc

TODO

5.1.2 La virgule comme séparateur

La virgule sert également dans un cas déjà observé : la séparation des déclarations de variables.

```c
int a = 3, b = a + 1;
```

5.1.3 Conclusion

Ben voilà c'est tout... [2]

Les programmes informatiques sont des routines, et plus particulièrement des fonctions, puisqu'ils retournent leur valeur de statut :

```c
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*

5.2.1 Le (véritable ?) prototype de *main*

Il existe en vérité deux formes standards possible pour le prototype de main :

```c
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 [3]

2. J'espère que cette conclusion dithyrambique plaira ^^
3. Le double pointeur est ainsi justifié :
   - les crochets désignent le tableau de caractères
   - l'étoile désigne le tableau des tableaux de caractères

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 $argv[0]$ = nom du programme :

```c
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char **argv) {
  printf("%s\n", argv[0]);
  return EXIT_SUCCESS;
}
```

```
user@computer ~/working_directory> gcc main.c -o main
user@computer ~/working_directory> ./main
./main
user@computer ~/working_directory> cd ..
user@computer ~> working_directory/main
working_directory/main
user@computer ~>
```

5.2.2 Exercices

Exercice 50 (Liste des arguments) [11]. 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 51 (Un cat minimaliste) [10]. 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>
```

Unions

5.3.1 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 destiné qu'à stocker un seul évènement. Analysons l'utilisation mémoire de la structure Event :

- $4 + 4 = 8$ octets pour stocker la sous-structure mouse
- $4 + 1 + 1 = 6$ octets pour stocker la sous-structure keyboard

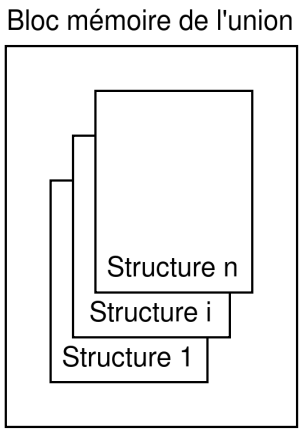
pour un total de $8 + 6 = 14$ octets.

En soi, il suffirait pourtant de seulement 8 octets pour stocker soit la sous-structure mouse, soit la sous-structure keyboard. C'est à cela que sert l'union.

5.3.2 Principe et syntaxe

L'idée de l'union est assez simple. Alors que la structure agrège les données [4] les unes à la suite des autres en mémoire, l'union les superpose dans le même espace mémoire :

4. En respectant ou non l'alignement



Ce qui en C s'écrit avec la même syntaxe que pour les structures, en utilisant à la place le mot-clé *union* :

```c
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 :

```c
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 [5]
- 8 octets pour union Event

Dans le cas de la programmation de systèmes embarquées ou pour des structures qui seront alloués un grand nombre de fois en mémoire, le gain de place est non négligeable. Il est ici d'un facteur 2 car il n'y a que deux évènements. La bibliothèque *Xlib* qui sert d'interface au serveur de fenêtrage X11 de Linux utilise une union de 33 structures pour stocker les évènements. [6]

Utiliser les unions à la place de structures quand cela est possible est une bonne pratique de programmation à assimiler.

5. Et pas 14 du fait de l'alignement mémoire, voir la section dédiée
6. Voir : https://tronche.com/gui/x/xlib/events/structures.html

5.3.3 Exercices

Cette section vient en continuité de la section précédente 5.3 sur les unions. Les champs de bits constituent en effet un autre moyen d'optimiser le stockage des données en espace. Il s'agit cependant d'une perte en vitesse d'exécution car la plupart des processeurs ne peuvent pas manipuler directement les bits d'un octet mais doivent appliquer des opérations bit-à-bit dessus.

5.4.1 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 $[\![0;31]\!]$ 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 :

$$\underbrace{00000000000}_{\text{inutilisées}}\ \underbrace{aaaaaaaaaaaa}_{\text{12 bits pour l'année}}\ \underbrace{mmmm}_{\text{4 bits pour le mois}}\ \underbrace{ddddd}_{\text{5 bits pour le jour}}$$

```c
#include <stdio.h>

unsigned short read_year(unsigned int date) {
  return (date >> 9) & 0xFFF;
}

unsigned char read_month(unsigned int date) {
  return (date >> 5) & 0b1111;
}

unsigned char read_day(unsigned int date) {
  return date & 0b11111;
}
  unsigned int write_day(unsigned short year, char month, char day) {
  return (year & 0xFFF) << 9 | (month & 0b1111) << 5 | (day & 0b11111);
}

int main(int argc, char *argv[]) {
  unsigned d = write_day(2003, 12, 22);
  printf("%u <=> %u/%u/%u\n", d, read_day(d), read_month(d), read_year(d));
  return 0;
}
```

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 ! [7]

5.4.2 Syntaxe

Les champs de bits [8] 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 MyBitField {
  int f1 : width_of_f1;
  ...
  int fn : width_of_fn;
};
```

Ici, chaque champ f_i utilisera exactement w_i bits :

$$\text{MyBitField} \equiv \underbrace{f_1}_{w_1 \text{bits}} \cdots \underbrace{f_n}_{w_n \text{bits}}$$

et la structure sera donc stockée sur un nombre d'octets plus limité. [9]

Pour revenir à l'exemple des dates :

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

struct Date {
  unsigned year;
  unsigned month;
  unsigned day;
};

struct BFDate {
  unsigned year : 12;
  unsigned day : 5;
  unsigned month : 4;
};

int main(int argc, char *argv[]) {
  printf("%ld\n", sizeof(struct Date)); // -> 12
  printf("%ld\n", sizeof(struct BFDate)); // -> 4

```

7. 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...

8. 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 :|

9. On a pas toujours $sizeof(\text{MyBitField}) = \left\lceil \frac{\sum_{i=1}^{n} w_i}{8} \right\rceil$ à cause de l'alignement.

```
20    struct BFDate d = {.day=22, .month=12, .year=2003};
21    printf("%u/%u/%u\n", d.day, d.month, d.year); // -> 22/12/2003
22    return EXIT_SUCCESS;
23  }
```

La taille de la structure struct BFDate est bien celle d'un entier sur 4 octets, comme cela avait déjà été déterminé dans la sous-section 5.4.1 précédente.

Par ailleurs, l'accès en lecture/écriture des champs est bien plus aisée !

5.4.3 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 :

```c
#include <stdlib.h>
#include <stdio.h>

struct BFDate {
  unsigned year : 12;
  unsigned day : 5;
  unsigned month : 4;
};

int main(int argc, char *argv[]) {
  struct BFDate d = {.day=22, .month=12, .year=2003

  for (unsigned int i = 0; i < 4294967295; i++) {
    d.day += 1;
    d.month += 1;
    d.year += 1;
  }
  printf("%u/%u/%u", d.day, d.month, d.year);
  return EXIT_SUCCESS;
}
```

```c
#include <stdlib.h>
#include <stdio.h>

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 < 4294967295; i++) {
    d.day += 1;
    d.month += 1;
    d.year += 1;
  }
  printf("%u/%u/%u", d.day, d.month, d.year);
  return EXIT_SUCCESS;
}
```

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) [10] du compilateur par :

10. Il s'agit du niveau d'optimisations par défaut, raison pour laquelle il n'y pas besoin de le spécifier habituellement. Compiler à un niveau zéro d'optimisation permet de plus grandes facilités de déboguage.

```
gcc main.c -o main -O0
```

on obtient à l'exécution dans les mêmes conditions logiciels (mêmes autres processus en cours d'exécution) les deux temps suivants :

- Sans champs de bits : 7.10 s
- Avec champs de bits : 37.59 s

L'utilisation des champs de bits provoque un ralentissement de 528%. En compilant avec différents niveaux d'optimisations (entre 1 et 3) :

1. $rapport = \frac{2.42}{0.94332} = 257\%$
2. $rapport = \frac{2.15}{0.00299} = 71906\%$
3. $rapport = \frac{2.82}{0.00281} = 100356\%$

Les rapports extraordinaires des optimisations de niveau 2 et 3 proviennent du fait que le compilateur "précalcule" logiquement les résultats des additions du fait d'un schéma dans le code qui a été repéré. En vérité, il n'y a même plus de boucle dans le code après la compilation.

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 [11], 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 [12] 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.

11. 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.
12. 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.

Définition 27 : 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.

Les classes de stockage permettent de définir avec plus de précision la portée et la durée de vie d'une variable au moment de sa déclaration.

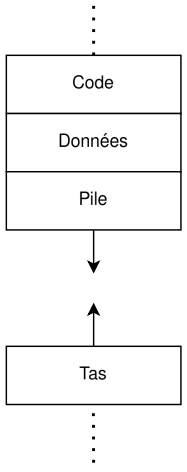
5.5.1 Quelques détails de la structure d'un programme en mémoire

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*[13]
- 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[14] 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 :

13. i.e. durables, la notion sera détaillée dans la suite.
14. 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...

- une section de données
- la pile
- le tas
- un registre

Section de données

Les sections de données d'un programme sont présentes dans le fichier binaire du programme compilé. Les sections de données ne peuvent donc ni grandir ni rétrécir puisque le code source du programme (avant compilation) décrit entièrement la taille des données. Elles sont de taille fixe et certaines données peuvent avoir une valeur initiale. On distingue alors deux types de sections de données :

- les sections initialisées : les valeurs initiales sont indiquées dans le fichier binaire du programme compilé
- les sections non initialisées : seule la taille de la donnée est précisée.

Une conséquence très importante de l'impossibilité d'une section de données de changer de taille est que toutes les variables définies dans cette section ne peuvent être supprimées. C'est-à-dire que leur valeur est *persistante*. Une variable de section de données présente dans une routine ne voit pas sa valeur réinitialisée à la sortie de la routine.

Variables allouées dynamiquement

Les données des variables allouées dynamiquement sont présentes dans le tas. Cependant, le pointeur vers ces données est lui stocké indépendamment de ces données. Il peut donc être stocké dans l'un des quatre lieux de stockage de variable décrits ci-dessus.

Registres

Les registres sont de très petites unités de mémoire, en général constituées de bascules *RS* ou *JK* en séries. Chaque bascule stocke un bit. Les processeurs ne possèdent pas une grande quantité de registres car ceux-ci sont très coûteux par rapport à la quantité de mémoire qu'ils proposent. Leur accès est toutefois extrêmement rapide, en écriture comme en lecture. Ils servent donc souvent comme variables temporaires pour les calculs du processeur [15].

Il est très difficile [16] d'accorder correctement les registres aux variables pendant les calculs. Cette opération est en général effectuée de manière automatisée par le compilateur.

5.5.2 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 [17]
- *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 [18]
- *extern* permet d'étendre une variable globale sur plusieurs fichiers sources.

Les deux classes de stockage les plus importantes sont *static* et *extern*. [19]

On peut les utiliser de la façon suivante :

15. 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.
16. Au sens informatique de complexité, il s'agit d'un problème d'ordonnancement NP-difficile.
17. En deux mots : complètement inutile
18. au contraire des variables globales qui sont persistentes mais... globales
19. Par "plus importantes" on entend : possèdent des applications pratiques régulières.

```
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 :

```
CLASSE TYPE x;
TYPE CLASSE x;
```

La classe *auto*matique

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 [20]. 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*.

20. Puisqu'il n'existe pas une infinité de registres.

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 :

```c
#include <stdio.h>

void example() {
  static int i = 0; // l'initialisation n'a lieu qu'à la première exécution
  printf("%d", i++);
}

int main() {
  ...
  for (unsigned int u = 0; u < 10; u++) {
    example();
  }
  ...
}
```

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 :

```c
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 globables 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

```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*.

On s'est intéressé dans la section précédente 5.5 aux différents moyens permettant d'indiquer au compilateur où les variables déclarés 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 [21] [22] :

- 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.

5.6.1 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;
```

21. Sans blague, genre y a pas un titre de section pour l'indiquer...
22. Certains langages plus complexes peuvent en avoir plus, comme le Rust par exemple qui est une vraie usine à gaz syntaxique.

Il faut faire attention à un détail : la valeur d'une constante ne peut lui être donné 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
```

Types complexes

Dans le cas des types complexes, chaque champ peut être définit 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 [23]. Cet espace peut être déclaré constant ou non [24]. En cela, l'application d'un const sur un pointeur peut être ambigüe. 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" :

23. Sans blague !
24. Malgré des apparences trompeuses, ceci n'est pas un concours de lapalissades.

```c
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 :

```c
// 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 :

```c
const int somestats[42] = {...};

array_sort(somestats, 42); // erreur
```

On a pas l'assurance que l'espace emémoire 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 :

```c
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 [25], le pointeur vers le nœud virtuel peut être déclaré lui-même constant :

```c
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.

25. 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.

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 :

```
const int x = 42;
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 :

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

struct Type {
  const int a;
  float b;
  const char c;
};

int main() {
  struct Type v = {.a = 5, .c = 'a'};
  int *p = &v.a; // avertissement à la compilation
  p = (int*)((&v.a)); // aucun avertissement (gcc comme clang)
  return EXIT_SUCCESS;
}
```

En effet, les capacités d'analyse *sémantique* du compilateur sont limités [26]. 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 en 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 modifié. C'est une assurance *grammaticale*.

26. On en déduit que sont aussi limités ses capacités d'optimisation qui sont directement liés.

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.

5.6.2 restrict

Définition 28 : 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 :

```
{
  void *restrict ptr = ...;
  // déclaration qu'aucun alias de ptr n'est présent dans ce bloc.
}
```

Ce déclarateur d'intention 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 :

```
void swap(int *restrict x, int *restrict y) {
  *x = *x ^ *y;
  *y = *x ^ *y;
  *x = *x ^ *y;
}
```

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.

5.7.1 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', etc, '!', '\n', '\0'};
printf(_anonym_array);
```

L'objectif est de montrer la généralisation la construction de littéraux pour des types quelconques.

5.7.2 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 [27] :

```
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
```

5.7.3 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);
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});
```

27. Bien que la notation soit lourde pour aucune raison...

```
14    }
15    return EXIT_SUCCESS;
16  }
```

Pointeurs de routines

5.8.1 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 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 implanté 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 29 : 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 permettrait 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...)

5.8.2 Syntaxe

Une routine en mémoire est une simple adresse, stockée 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 routines, et même de définir des tableaux de routines.

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 :

```
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 [28] peut alors être implantée de la manière suivante :

28. Allons au plus simple, même si le code n'est pas "efficace" en soi

```c
// definitions de struct Point, cmp_x_y et cmp_y_x ici.

void swap(struct Point array[], unsigned int i, unsigned int j) {
  struct Point tmp = array[i];
  array[i] = array[j];
  array[j] = tmp;
}

// le pointeur vers 'r' ne doit pas être modifié (relation modifiée) durant la fonction i_min
unsigned int i_min(const struct Point a[], unsigned int l, char (*const r)(struct Point, struct Point)) {
  unsigned int m = 0;
  for (unsigned int i = 1; i < l; i++) {
    if (r(a[i], a[m]) == 1) { // a[i] < a[m] selon la relation donnée
      m = i;
    }
  }
  return m;
}

// idem
void selection_sort(struct Point a[], unsigned int l, char (*const r)(struct Point, struct Point)) {
  for (unsigned int i = 0; i < l; i++) {
    for (unsigned int j = i; j < l; j++) {
      unsigned int m = i_min(a + i, l - i, r);
      swap(a, i, m + i);
    }
  }
}

int main() {
  ...

  struct Point arr[10] = {...};
  selection_sort(arr, 10, cmp_x_y);

  ...
}
```

5.8.3 Tableaux de routines

On peut faire des tableaux de routines de même qu'on définirait des tableaux de pointeurs :

```c
#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

  ...
}
```

5.8.4 Exercices

Exercice 52 (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 46** et implanter une fonction `void linkedlist_map(struct LinkedList *l, void (*const f)(int*));` qui applique la procédure f 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 [29] sont autorisés, ils sont remplacés par le caractère simple correspondant
   - les lignes continues sont fusionnés une seule ligne (rappel : 3.3)
   - 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 [30].

Cette section vise à préciser en détails les directives de préprocesseurs déjà introduites en section 3.6 et à les compléter avec les directives de définition de *macro*instructions (appelées simplement *macros* par la suite).

On peut observer le résultat du traitement du préprocesseur en exécutant l'instruction :

29. Voir 5.13
30. 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.

```
> gcc main.c -E -o main.preprocessed # génère un fichier texte
> cat main.preprocessed # affichage du fichier texte
... (assez long)
```

5.9.1 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 protoypes et des définitions de structure, comme cela a été vu dans la section 4.13 sur la modulation.

Son usage peut toutefois être généralisé :

filename.ext

```
const char t[] = "Hello World !\n";
```

main.c

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

int main() {
  #include "filename.ext"
  printf("%s", t); // Hello World !\n
  return EXIT_SUCCESS;
}
```

Ce type de code ne doit être utilisé que dans des cas particuliers.

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 [31] et doivent être programmés "en dur" dans le programme. L'image peut être représentée par du code C :

```
const unsigned char image[] = {
  // octets de couleur des pixels de l'image
}
```

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 :

31. ou difficilement

```
#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 4.13 sur la modulation).

5.9.2 Compilation conditionnelle

Motivation

La compilation conditionnelle a été abordée succintement dans la section 4.13.6 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
#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 4.13.6 sur les garde-fous les instructions raccourcies #ifdef et #ifndef

5.9.3 Diagnostiques

Les instructions de diagnostique 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 précisement 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
```

5.9.4 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 4.13 :

```
#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.

5.9.5 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 :

```c
#include <stdio.h>
#include <stdlib.h>

#define float int

int main() {
  float x = 3.14;
  printf("%d\n", x); // affiche 3 sans avertissements
  return EXIT_SUCCESS;
}
```

On peut définir des textes plus longs :

```c
#include <stdio.h>

#define SALUT printf("Salut !!!\n");
#define FIN printf("Au revoir !\n"); return 0;
#define DIRE_SALUT_ET_PARTIR SALUT FIN

int main() {
  DIRE_SALUT_ET_PARTIR
}
```

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 :

```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 :

```c
#include <stdio.h>
#include <stdlib.h>

#define MAX(A, B) ((A) > (B) ? (A) : (B)) // ne dépend ni du type de 'a' ni de celui de 'b'

int main() {
  int a = 2;
  int b = 5;
  printf("%d\n", MAX(a, b)); // 5
  double x = 3.14;
  float y = 1.414;
  printf("%f\n", MAX(x, y)); // 3.140000
  return EXIT_SUCCESS;
}
```

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* :

```c
#include <stdio.h>
#include <stdlib.h>

#define SYRACUSA(X) X = (!(X%2)) ? (X >> 1) : (X * 3 + 1);

int main() {
  int n = 53;
  SYRACUSA(n);
  printf("%d\n", n); // 160
  return EXIT_SUCCESS;
}
```

5.9.6 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 :

```c
#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 :

```c
#include <stdio.h>
#include <stdlib.h>

#define AFFICHER(texte) printf(#texte "\n")

int main() {
  AFFICHER(Salut les pingouins !!!); // affiche Salut les pingouins !!!\n
  return EXIT_SUCCESS;
}
```

On peut écrire ainsi des macros utilitaires :

```
#define WARN(expression) fprintf(stderr, "Avertissement : "#expression"\n")

...

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 ## :

```
#define AB int
#define C A##B // <=> #define C int

#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 :

```
pri##ntf("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[32] l’initialisation par une macro fonctionelle 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)
}
```

5.9.7 Exercices

Exercice 53 (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.

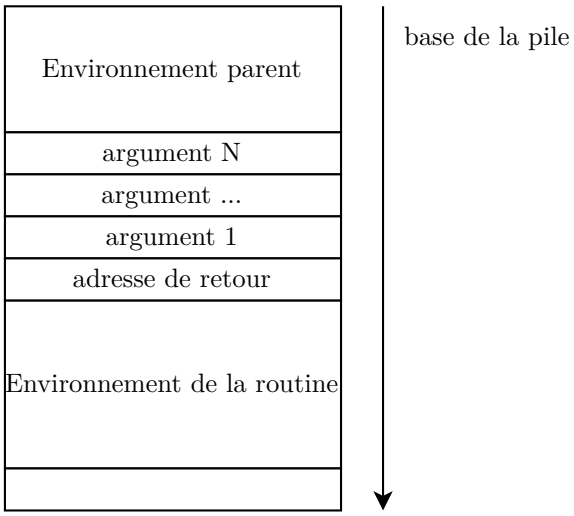
5.10.1 Motivation

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 ∧∧

5.10.2 Appels de routines

Il existe plusieurs standards d’appel de routines. Lors d’un appel de routine *suivant le standard du langage C*[33], les arguments sont poussés sur la pile suivant le schéma simplifié suivant :

32. Dans le cas où il y aurait une très grand quantité de procédures
33. Pour des raisons d’optimisation, le compilateur peut utiliser d’autres standards, voir https://en.wikipedia.org/wiki/X86_calling_conventions pour des détails.



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.

5.10.3 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 :

```c
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"*[34].

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)

```c
#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);
}
```

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) : initalise une liste d'arguments *dest* comme une copie de la liste *source*

Ainsi, on peut écrire :

```c
#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);
  }
```

34. https://github.com/gcc-mirror/gcc/blob/master/gcc/ginclude/stdarg.h pour le vérifier.

```
    va_end(args);
    return sum;
}
```

5.10.4 Macros variadiques

TODO

5.10.5 Exercices

Exercice 54 (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.

TODO

TODO

L'objectif de cette section récréative est l'écriture de codes C illisibles, et même dirait-on particulièrement immondes. On utilisera pour cela divers outils du langage C plus ou moins peu inutilisés :

- l'indexation inversée
- la définition *K&R* de fonctions (première version du C)
- les digraphes et trigraphes
- la concaténation de symboles préprocesseurs
- les tableaux anonymes (construction de littéraux)
- etc...
- un peu d'imagination

Pour rester pragmatique, observer des codes très mal écrits ou utilisant des fonctionnalités extrêmement peu utilisés et juste là pour être illisibles amène à apprécier par contraste l'importance d'un style d'écriture clair et compréhensible.

5.13.1 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 interprétent correctement [35].

Cela ouvre des possibilités pour écrire des codes très moches.

5.13.2 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}
```

5.13.3 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 appelés 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
```

35. Voir le boutisme en informatique : https://fr.wikipedia.org/wiki/Boutisme

```c
int main(int argc, char **argv) {
  printf("%c", "Salut"<:0:>);
  return EXIT_SUCCESS;
}
```

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 :

```c
%:include <stdlib.h>
%:include <stdio.h>

int main(int argc, char **argv) ??<
  // What's the fuck is happening ????????/
  ceci fait toujours partie du commentaire ')=à- ç+qrh k poj )à-çtàç)'78320  É"'"??/
  et ça aussi LTMY ,<SMFL, SDRM LJRST +98602....'@9ÇÇÇ$

  printf("%c", "Salut"<:0:>);
  return EXIT_SUCCESS;
??>
```

Toutefois, cette possibilité offerte par le langage pour d'excellentes raisons a été supprimé dans la version *C23*[36]. Il faut donc forcer le compilateur à utiliser une version plus ancienne du langage :

```
user@computer ~/working_directory> gcc main.c -o main -std=c11 # version de 2011
user@computer ~/working_directory> ./main
S
user@computer ~/working_directory>
```

5.13.4 Un peu d'imagination

Ici, un exemple écrit par Basile Tonlorenzi :

```c
#include <stdio.h>
#include <stdlib.h>

#define AB int
#define A ch
```

36. Malheureusement pour de pas si mauvaises raisons, mais certains n'ont visiblement pas le sens de l'humour...

```c
%:define _ x
#define B ar
#define C A##B
??=define P C

int f(x, y)
  int** _;
  int** y; <%return *0[y:>*(*0<:x]);
}

int main(int argc, char** argv){
  if(argc < 2){
    printf("File UN chiffre entre 0 et 9 enculé \n"); // fleuri mais efficace
      return EXIT_FAILURE;
  }

  C y = 0[1<:argv]:> - '0';
  auto* a = &y ; // A changer #include .virgule
  printf("%d \n", f(&a,&a));
}
```

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>
```

```c
#include <stdio.h>
#include <stdlib.h>

#define AB int
#define A ch
#define $(A) A,A
%:define _ x
#define B ar
??=define C A##B

int f(x, y)
  /* NE PAS SUPPRIMER CETTE LIGNE DE COMMENTAIRE */
  int** _; // Mais putain c'est quoi ce bordel ????/
  while (y) {goto 5; continue; 5: y = -y;}
  int** y; <%return *'\0'[y:>*\
  (*(__LINE__-2*putchar('\b'))<:x]);
  static int very_useful = 0x000a6425;
}

int main(int _, char** argv){
  if(_ = !!!!(_-2)){
    printf("File UN chiffre entre 0 et 9 enculé"); // fleuri mais efficace
    return !!!!(_-2);
  }
  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, hihihi ![37]

37. (rire vraiment très diabolique)

Partie III

Petite parenthèse théorique

Introduction

Les deux premières parties de ce livre se sont concentrés 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 40** 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 [38]
- 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 [39].

D'autres exercices comme l' **Exercice 39** et l' **Exercice 46** 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 :

- pour la culture, le modèle de calcul des machines de Turing et la notion de calculabilité qui explore les limites de ce qui peut être calculé par un ordinateur
- 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
- les approches algorithmiques fondamentales pour la résolution de problèmes

Les exercices seront une opportunité pour découvrir les algorithmes classiques sur les structures de données élémentaires et les approches algorithmiques.

Dans le cadre de l'ISMIN, cette partie devrait suffire à apporter au lecteur toutes les connaissances théoriques fondamentales exigées durant le cursus [40] en ce qui concerne la programmation.

38. 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.
39. 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...
40. Voire parfois plus.

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 [41] 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 [42] 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 *README*s
- 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 [43]
  - 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 du développement, son coeur, mais non pas son tout.

41. Ainsi que nombre de notions gravitant autour comme les représentations binaires des nombres ou quelques principes de la programmation système

42. 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.

43. déjà évoqué en fin de première partie et début de deuxième

CHAPITRE

6

OUTILS À LA PROGRAMMATION

Makefiles

make est un logiciel installé par défaut sur les environnements Linux qui permet d'automatiser certaines tâches, en particulier des tâches de compilation de programmes. L'objectif de cette section est d'expliquer le principe général de fonctionnement de ce logiciel et de détailler son utilisation dans le cadre du développement logiciel en C.

6.1.1 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*.

6.1.2 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
- d'actions à effectuer sur les ressources pour produire la cible

Un fichier *make* classique est construit de la manière suivante :

```
CIBLE: RESSOURCES # Un commentaire
  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 :

```
main: main.o # commence par chercher à produire 'main.o'
  gcc main.o -o main # Produit 'main'

main.o: main.c # 'main.c' est déjà produit, donc inutile de le produire
  gcc main.c -c # Produit 'main.o'
```

Un deuxième un peu moins inutile[1] :

```
main: main.o module1.o module2.o # commence par chercher à produire 'main.o'
  gcc main.o module1.o module2.o -o main # Produit 'main'

main.o: main.c # 'main.c' est déjà produit, donc inutile de le produire
  gcc main.c -c # Produit 'main.o'

module1.o: module1.c
  gcc module1.c -c

module2.o: module2.c
  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>
```

1. 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 :

```
main: main.c module1.c module2.c
  gcc main.c module1.c module2.c -o main # Produit 'main'
```

Une petite explication s'impose, en trois points :

- *make* ne produit pas une cible si celle-ci est déjà à jour. On entend par à jour que la date de modification de la cible est plus récente que la date de modification de chacune de ses ressources. Ainsi, il est inutile de produire une seconde fois ce qui l'a déjà été.
- L'opération la plus lourde à la compilation est la génération des fichiers objets à partir des fichiers sources. L'édition de liens est très rapide en comparaison
- En précisant des règles de production séparées pour les fichiers objets, on garantit que seules les modules ayant subis des modifications sont recompilés. Les autres n'ont pas besoin de l'être

Dans le cadre de très gros projets, le gain de temps à la compilation est significatif.

6.1.3 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 :

```
EXE_NAME = main
COMPILER = gcc
OBJS = main.o module1.o module2.o

$(EXE_NAME): $(OBJS)
  $(COMPILER) $(OBJS) -o $(EXE_NAME) # Produit 'main'

main.o: main.c
  $(COMPILER) main.c -c

module1.o: module1.c
  $(COMPILER) module1.c -c

module2.o: module2.c
  $(COMPILER) module2.c -c
```

6.1.4 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 [2] :

Symbole	Signification
$@	Nom de la cible
$<	Nom de la première ressource
$?	La liste des noms, séparés par des espaces, de toutes les ressources plus récentes que la cible
$^	La liste des noms de toutes les ressources, séparés par des espaces

En utilisant ces variables, l'exemple ci-dessus se réécrit avec plus de concision :

```
1  EXE_NAME = main
2  COMPILER = gcc
3  OBJS = main.o module1.o module2.o
4
5  $(EXE_NAME): $(OBJS)
6    $(COMPILER) $^ -o $@ # Produit 'main'
7
8  main.o: main.c
9    $(COMPILER) main.c -c
10
11 module1.o: module1.c
12   $(COMPILER) module1.c -c
13
14 module2.o: module2.c
15   $(COMPILER) module2.c -c
```

6.1.5 Réduire le nombre de règles avec la *stem*

Le symbole % désigne le "radical" *quelconque* d'un mot (en anglais *stem*, littéralement *tige* [3]). 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

2. Voir https://www.gnu.org/software/make/manual/html_node/Automatic-Variables.html pour les autres.
3. 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.

6.1.6 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 $*$.[4] 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 :

```
EXE_NAME = main
COMPILER = gcc

$(EXE_NAME): *.c
  $(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 interprété. Ainsi :

```
EXE_NAME = main
COMPILER = gcc
SRC = *.c

$(EXE_NAME): $(SRC) # la généralisation est effectuée ici
  $(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 effectué lorsque "$*$.c" remplace *SRC* dans la ressource.

Pour forcer l'interprétation de $*$, on utilise le mot-clé *wildcard* :

```
EXE_NAME = main
COMPILER = gcc
SRC = $(wildcard *.c) # la généralisation est effectuée ici

$(EXE_NAME): $(SRC)
  $(COMPILER) $^ -o $@
```

4. Voir https://www.grymoire.com/Unix/Bourne.html#uh-4pour les autres.

6.1.7 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 finit par .o"*

Ce qui résoud effectivement le problème. On peut maintenant écrire le *Makefile* suivant :

```
1  EXE_NAME = main
2  COMPILER = gcc
3  SRC = $(wildcard *.c)
4  OBJS = $(SRC:%c=%.o)
5
6  $(EXE_NAME): $(OBJS)
7    $(COMPILER) $^ -o $@
8
9  %.o: %.c
10   $(COMPILER) $< -c
```

et ne plus jamais se préoccuper de quelle commande écrire pour compiler. Taper *make* dans le répertoire de travail suffit !

6.1.8 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 aura survenue puisque l'action ne produit pas la cible. Il faut donc préciser que cette règle est fausse à *make* (*phony* en anglais) :

```
clean:
  rm -f *.o $(EXE_NAME)

.PHONY: clean
```

On peut alors appeler cette règle :

```
user@computer ~/working_directory> make clean
rm -f *.o main
user@computer ~/working_directory> make
gcc main.c -c
gcc module1.c -c
gcc module2.c -c
gcc main.o module1.o module2.o -o main
user@computer ~/working_directory>
```

6.1.9 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 [5] et un second répertoire pour les entêtes [6])
- 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

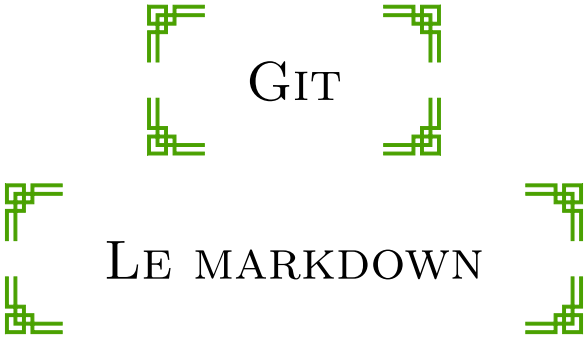
On peut ajouter par défaut le paramètre *-lm* pour utiliser la bibliothèque math.h.

On peut ajouter un répertoire *lib* au projet pour y stocker des bibliothèques externes (d'extensions ".a" ou ".so")

Utiliser un débogueur

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 basiques qu'il propose.

5. Le nom standard est *src*
6. Le nom standard est *include*


Git
Le markdown

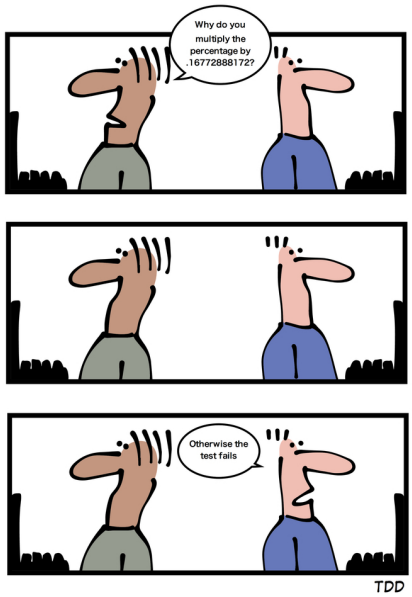
CHAPITRE

7

PROGRAMMER PROPREMENT

À PROPOS DU STYLE

LA GESTION DES ERREURS



La gestion d'erreurs en programmation est absolument fondamentale. Les raisons sont multiples :

- un ordinateur physique n'est pas théorique, et un programme subit tous les défauts de l'implantation. On peut par exemple penser aux opérations sur les nombres flottants qui induisent une imprécision. Les résultats obtenues sur les flottants sont donc dans la majorité des cas faux pour la modélisation de nombres réels
- les programmes développés sur un ordinateur ont été écrits par des humaines faillibles. Ainsi, il n'est jamais sûr à 100% qu'un programme écrit soit correct, bien qu'il soit démontré correct en théorie.

Bien sûr, dans le cas de "petits" programmes, ne pas gérer les erreurs n'a qu'une incidence minime sur le développement. Dans le cas de très grands projets, des erreurs humaines ont de fortes chances de se glisser dans certaines étapes du développement et les tester et les gérer est extrêmement important pour éviter des bogues incontrôlés et surtout non localisés.

Les manquements à la gestion des erreurs et aux tests sont la source première des failles de sécurité des systèmes informatiques. [1]

1. Le test logiciel fait malgré tout partie (de mon point de vue subjectif) des domaines de l'informatique pratique les plus atrocement chiants 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.

7.2.1 Un petit exemple pour prendre la température

Lorsque l'on commence à programmer, en C notamment, les erreurs de segmentation sont parmis les plus fréquentes.[2] Pour les réduire, il faut au maximum réduire les probabilités de manipuler des pointeurs non initialisés. Pour cela, il faudrait tester après chaque initialisation de pointeur (par un malloc par exemple) si le pointeur est effectivement non NULL.

Avec un malloc cela ressemblerait à :

```
int *x = NULL;
x = (int *)malloc(10 * sizeof(int));
if (!x) {
  fprintf(stderr, "Erreur d'allocation memoire !\n");
}
```

En toute rigueur, c'est ce qu'il faudrait faire. En effet, si l'ordinateur ne dispose plus d'assez de mémoire vive pour allouer dynamiquement la variable, malloc renverra NULL. Dans les faits, effectuer un test à chaque allocation est assez lent. Si on sait qu'on alloue pas 3 ou 4 *Go* d'un coup, la probabilité d'erreur est furieusement proche de zéro. Et si le système d'exploitation n'arrive pas à allouer une centaine d'octets dynamiquement, il y a fort à parier qu'il ne puisse rien faire du tout de toute manière et qu'une erreur de segmentation n'ait aucune incidence dans la pratique.

Remarque : Dans le cas de la programmation de systèmes embarqués, le paragraphe précédent est nul et non avenu puisque les systèmes embarqués doivent fonctionner avec des ressources parfois extrêmement limités. La question est alors plus de trouver une autre manière de faire qui optimise la quantité de mémoire utilisée.

La gestion des erreurs est donc dépendante du contexte.

7.2.2 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

2. Ah bon ?

de caractères lues ou écrits. C'est donc en contrôlant ce nombre que l'on sait si la fonction a échoué. La lecture du manuel est ici indispensable.

Tests unitaires

C'est pour cela que les tests unitaires existent. L'objectif des programmes de tests unitaires est de vérifier des routines isolées (unitaires) dans des cas particulier d'exécution qui pourrait potentiellement amener à une erreur en comparant la sortie de chaque routine sur une entrée e prédéterminée avec une sortie attendue $s(e)$. Il faut alors tester tous les entrées les plus particulières.

Chaque routine d'un programme doit en théorie être testée par des tests unitaires. Dans la pratique, du fait des ressources humaines nécessaires à la réalisation de ces tests, on ne test que les routines "non triviales" d'un programme, c'est-à-dire celles qui peuvent présenter des fautes potentielles.

L'importance des tests unitaires est multiple :

- Correction efficiente des bogues : une routine bogué est plus facilement 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

CHAPITRE

8

CORRECTION DES 54 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 envoyé par mail à l'adresse *bastiengarnier17@gmail.com* dans l'optique d'amélioration des futures éditions.

Première partie

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

8.1.1 Opérations logiques sur les mots binaires

Solution 1 (La compréhension pour mieux comprendre). On peut établir un lien entre les opérations logiques élémentaires et les opérations élémentaires en théorie des ensembles (union, union disjointe, intersection, privation). En effet :

- $A \cup B = \{e \mid e \in A \vee e \in B\}$, on a : $\mathbb{P}(e \in A \cup B) = \mathbb{P}(e \in A) + \mathbb{P}(e \in B) - \mathbb{P}(e \in (A \cap B))$
- $A \uplus B = \{e \mid e \in A \oplus e \in B\}$, on a : $\mathbb{P}(e \in A \uplus B) = \mathbb{P}(e \in A) + \mathbb{P}(e \in B) - 2 \times \mathbb{P}(e \in (A \cap B))$
- $A \cap B = \{e \mid e \in A \wedge e \in B\}$, 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 :

- $\vee \ : \ (a,b) \mapsto a+b-ab$
- $\oplus \ : \ (a,b) \mapsto |a-b|$
- $\wedge \ : \ (a,b) \mapsto a \times b$
- $\neg \ : \ a \mapsto 1-a$

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 $*_i, i \in [\![0, M-1]\!]$ 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 $T = \{e \in \mathcal{B}^N \mid *(e) = 1\} = \{t_1, \ldots, t_{Card(T)}\}$.

D'où, en posant que l'égalité de deux éléments vaut 1 et leur différence vaut 0 [1] :

$$*(e) = 1 \Leftrightarrow \bigvee_{i=1}^{Card(T)} (e = t_i) \tag{8.1}$$

On note $e = e_{N-1} \ldots e_0$ et pour tout $i \in [\![1, k]\!]$, $t_i = t_{i,N-1} \ldots t_{i,0}$.

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 $K_1(t_i) = \{k \in \{[\![0, N-1]\!] \mid t_{i,k} = 1\}$ et $K_0(t_i) = \{k \in \{[\![0, N-1]\!] \mid t_{i,k} = 0\}$.
Alors :

$$\begin{aligned} e = t_i \quad \Leftrightarrow \quad & \bigwedge_{k \in K_1(t_i)} (e_k = t_{i,k}) \bigwedge_{k \in K_0(t_i)} (e_k = t_{i,k}) \\ & \bigwedge_{k \in K_1(t_i)} (e_k = 1) \bigwedge_{k \in K_0(t_i)} (e_k = 0) \\ & \bigwedge_{k \in K_1(t_i)} e_k \bigwedge_{k \in K_0(t_i)} \neg e_k \end{aligned}$$

Finalement, on arrive à la définition par compréhension de $*$:

$$\begin{array}{rccl} * \ : & \mathcal{B}^N & \to & \mathcal{B} \\ & (e_{N-1}, \ldots, e_0) & \mapsto & \displaystyle\bigvee_{i=1}^{Card(T)} \left(\bigwedge_{k \in K_1(t_i)} e_k \bigwedge_{k \in K_1(t_i)} \neg e_k \right) \end{array}$$

Les informations de la fonction sont contenues dans T.

Remarque : $\mathcal{B}^N$ est en bijection avec $[\![0, 2^N - 1]\!]$. Les opérations $\vee$, $\wedge$ et $\neg$ permettent donc de définir n'importe quelle fonction sur $[\![0, 2^N - 1]\!]$ d'arité finie, donc par exemple l'addition, la multiplication, etc... [2]

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 propositionnels 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 $V = (v_{N-1}, \ldots, v_0)$ appelés *interprétations* tels que $*(V) = Vrai$. 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 nommés respectivement *somme de produits* et *produit de sommes* en raison des expressions trouvés à l' **Exercice 1** .

Solution 3 (Porte NAND). La table de vérité de NAND est :

A	B	$A \uparrow B$
0	0	1
0	1	1
1	0	1
1	1	0

On a ensuite pour tout $a, b \in \mathcal{B}$:

- $\neg(a) = a \uparrow a$
- $a \wedge b = \neg a \uparrow b = (a \uparrow b) \uparrow (a \uparrow b)$
- $a \vee b = \neg(\neg a \wedge \neg b) = \neg a \uparrow \neg b = (a \uparrow a) \uparrow (b \uparrow b)$

Solution 4 (Porte NOR). La table de vérité de NOR est :

A	B	$A \uparrow B$
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 $0 \oplus 0 = x$ et $0 \oplus 1 = 1$ et $1 \oplus 0 = 1$. Donc 0 est un élément neutre par $\oplus$. Comme $1 \oplus 1 = 0$, 0 est le seul élément neutre.
- $\oplus$ est associative

- Pour tout $x \in \mathcal{B}$, $x \oplus x = 0$ donc $x^{-1} = x$
- $\oplus$ est commutative

Donc G est bien un groupe commutatif.

Solution 6 (Inversibilité des décalages). Les décalages ne sont pas inversibles. En effet, quand des bits sont supprimés par un décalage de trop de bits, l'information est perdue. Un décalage dans l'autre sens ne produit que des 0. Par exemple : $(01000 \ll 2) \gg 2 = 00000 \neq 01000$

8.1.2 Opérations arithmétiques

Solution 7 (Du décimal au binaire).

- $(01001100)_2$
- $(10111100)_2$
- $(00100001)_2$
- $(01101101)_2$
- $(01011100)_2$
- $(11010011)_2$
- $(00000100)_2$
- $(11101110)_2$
- $(10100001)_2$
- $(01111110)_2$

8.1.3 Opérations sur les nombres à virgules

Solution 8 (Écriture en base 2 de nombres non entiers).

- $(1110.1001)_2$
- $(1.01101)_2$
- $(111.0111)_2$
- $(0.0001001\underline{001}\dots)_2$: il existe un nombre infini de décimales. On a le même phénomène avec la représentation de $\frac{1}{3}$ en décimal.

Solution 9 (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 :

$$log_2\left(\frac{1}{\sqrt{P}}\right) = -\frac{1}{2}log_2(P) \tag{8.2}$$

$$\forall x \in \mathbb{R}_{f32},\ x = itof\left(A^{-1}(log_2(x) - B)\right) \tag{8.3}$$

$$\forall x \in \mathbb{R}_{f32},\ log_2(x) = Aftoi(x) + B \tag{8.4}$$

En utilisant l'égalité (8.2) dans (8.3) en $x = \frac{1}{\sqrt{P}}$, on obtient :

$$\frac{1}{\sqrt{P}} = itof\left(-\frac{1}{2A}(log_2(P) + 2B)\right)$$

En injectant (8.4), on a alors :

$$\frac{1}{\sqrt{P}} = itof\left(-\frac{1}{2}ftoi(P) - \frac{3B}{2A}\right)$$

8.1.4 Représentation hexadécimale

Solution 10 (Conversion binaire-hexadécimale).

- $713705 = (1010\ 1110\ 0011\ 1110\ 1001)_2 = 0\text{xAE3E9}$
- $8.8 = (1.00011001001001001001001)_2 \times 2^3 = 0100\ 0001\ 1000\ 1100\ 1001\ 0010\ 0100\ 1001 = 0\text{x418C9249}$
- $42 = (0001\ 1010)_2 = 0\text{x1A}$
- $-1.1 = -(1.00010010010010010010010)_2 \times 2^1 = 1100\ 0000\ 0000\ 1001\ 0010\ 0100\ 1001\ 0010 = 0\text{xC0092492}$
- $101 = (0110\ 0101)_2 = 0\text{x65}$

8.1.5 Caractères ASCII

Solution 11 (Traduction ASCII 1). La chaîne de caractères décrite est :

"Minitel = GOAT"

Solution 12 (Traduction ASCII 2). Le mot binaire qui décrit cette chaîne est :

0x4F6820210A5175692065732D74752021083F

Deuxième partie

8.2.1 Bases du langage

Variables

Solution 13 (Interversion de variables par effet de bord).

```c
#include <stdio.h>
#include <stdlib.h>

int main() {
  int a = 8;
  int b = 6;

  // l'échange :
  int tmp = a;
  a = b;
  b = tmp;

  return EXIT_SUCCESS;
}
```

Formatage de chaînes de caractères

Solution 14 (Taille des types).

```c
#include <stdio.h>
#include <stdlib.h>

int main() {
  printf("sizeof(char) = %ld\n", sizeof(char));
  printf("sizeof(short int) = %ld\n", sizeof(short int));
  printf("sizeof(int) = %ld\n", sizeof(int));
  printf("sizeof(long int) = %ld\n", sizeof(long int));
  printf("sizeof(long long int) = %ld\n", sizeof(long long int));
  printf("sizeof(float) = %ld\n", sizeof(float));
  printf("sizeof(double) = %ld\n", sizeof(double));
  printf("sizeof(long double) = %ld\n", sizeof(long double));
  printf("sizeof(pointer) = %ld\n", sizeof(void*));

  return EXIT_SUCCESS;
}
```

Opérateurs sur les variables

Solution 15 (Valeur).

Le premier code est strictement équivalent au code suivant :

```c
int i = 10;
i = i - i;
i--;
```

D'où $i = -1$ en fin d'exécution. Le second code est strictement équivalent au code suivant :

```c
int i = 10;
i--;
i = i - i;
```

D'où $i = 0$ en fin d'exécution.

Solution 16 (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 17 (Priorité des opérateurs).

```c
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;
```

Solution 18 (Interversion sans effet de bord (1)).

```c
#include <stdio.h>
#include <stdlib.h>

int main() {
  int a = ...; // valeur quelconque
  int b = ...; // valeur quelconque

  // l'échange :
  a = a + b;
  b = a - b; // b = a + b - b = a
  a = a - b; // a = a + b - a = b

  return EXIT_SUCCESS;
}
```

On note a_p et b_p les valeurs de a et b précédent 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 19 (Interversion sans effet de bord (2)).

```c
#include <stdio.h>
#include <stdlib.h>

int main() {
  int a = ...;
  int b = ...;

  a = a ^ b;
  b = a ^ b;
  a = a ^ b;

  return EXIT_SUCCESS;
}
```

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 20 (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 $14 = 16 - 2 = 2^4 - 2^1 = (1 \ll 4) - (1 \ll 1)$

```c
#include <stdio.h>
#include <stdlib.h>

int main() {
  int a = 57;
```

```
  int r = (a << 4) - (a << 1);
  printf("57x14 = %d", r);
  return EXIT_SUCCESS;
}
```

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 21 (Valeur absolue (1)). On utilise la particularité qu'un décalage logique (effectué sur `unsigned int`) n'effectue pas d'extension du signe :

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

int main() {
  int x = -5; // valeur quelconque positive ou négative
  unsigned int abs_x = x;
  unsigned int positive = (abs_x >> 31) - 1; // -1 si positif, 0 si négative
  abs_x = ((~positive) & (-x)) | (positive & x);
  printf("|%d| = %u", x, abs_x);
  return EXIT_SUCCESS;
}
```

Par ailleurs, $-1 =$ `0xFFFFFFFF`, donc `-1 & x == x` est toujours vrai et `0 & x == 0` est toujours vrai.

Projection de type

Solution 22 (Quelques évaluations entières).

1. *vraie* car `0xFFFFFFFF` $+ 1 = 0$ ($e1$ est sur 32 bits)
2. *faux* car `(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 ∧ e3) % 8) - 1)` $= 1$.
   Alors : $e4 \equiv$ `(unsigned char)(257) - 2` $= -1$

Structures de contrôle du flot d'exécution

Solution 23 ().

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

#define N ...

int main() {
  int u = 3;
  unsigned int n = 0;
  while (n < N) {
```

```
    u = 3*u*u - 5*u + 2;
    n++;
    printf("u(%u) = %d\n", n, u);
  }
  return EXIT_SUCCESS;
}
```

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 24 (Question d'âge).

```c
#include <stdio.h>
#include <stdlib.h>

int main() {
  signed char age = 18;
  if (age < 0) {
    printf("Soyez patient(e), votre tour arrive");
    return EXIT_SUCCESS;
  } else if (age < 18) {
    printf("Mineur");
    return EXIT_SUCCESS;
  } else if (age == 18) {
    printf("Tout juste majeur");
    return EXIT_SUCCESS;
  } else {
    printf("Majeur");
    return EXIT_SUCCESS;
  }
  printf("Et la surpopulation alors ?");
  return EXIT_SUCCESS;
}
```

Solution 25 (Suite de Fibonacci).

```c
#include <stdio.h>
#include <stdlib.h>

#define N ...

int main() {
  unsigned int n = N;
  unsigned int f_m_1 = 0;
  unsigned int f_m_2 = 1;
  unsigned int tmp;
  for (; n > 0; n--) {
    tmp = f_m_1;
    f_m_1 = f_m_1 + f_m_2;
    f_m_2 = tmp;
  }
  printf("f(%u) = %u", N, f_m_1);
```

```
17    return EXIT_SUCCESS;
18  }
```

Solution 26 (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 [3] est donc le suivant :

```
1   #include <stdio.h>
2   #include <stdlib.h>
3
4   #define N ...
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 $\frac{n}{\sqrt{n}} \underset{n\to\infty}{\to} \infty$, l'algorithme est donc plus efficace (il ne s'agit pas d'une constante).

Routines

Solution 27 (Encore Fibonacci).

```
1   #include <stdio.h>
2   #include <stdlib.h>
3
```

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).

```c
unsigned int fibo(int n) {
  unsigned int f_m_1 = 0;
  unsigned int f_m_2 = 1;
  unsigned int tmp;
  for (; n > 0; n--) {
    tmp = f_m_1;
    f_m_1 = f_m_1 + f_m_2;
    f_m_2 = tmp;
  }
  return f_m_1;
}
int main() {
  for (unsigned int n = 0; n <= 100; n++) {
    printf("fibo(%u) = %u\n", n, fibo(n));
  }
  return EXIT_SUCCESS;
}
```

Solution 28 (Puissances entières).

```c
long int pow(int x, unsigned int n) {
  long int p = 1;
  for (; n > 0; n--) {
    p *= x;
  }
  return p;
}
```

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 29 (Quelques procédures inutiles pour devenir un bot efficace).

```c
void mov(int *x, int val) {
  *x = val;
}

void add(int *a, int *b) {
  *a += *b;
}

void mul(int *y, int *z) {
  *y *= *b;
}

void pow(int *x, int n) {
  int p = 1;
  for (; n > 0; n--) {
    p *= *x;
```

```
  }
  *x = p;
}
```

Solution 30 (Interversion sans effet de bord (3)).

```
void swap(int *x, int *y) {
  if (x == y) {
    return;
  }
  *x = *x ^ *y;
  *y = *x ^ *y;
  *x = *x ^ *y;
}
```

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 31 (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 :

```
double x; // 8 octets
float *x_left = (float*)&x; // pointeur sur 4 octets de gauche
float *x_right = (float*)&x + 1; // pointeur sur 4 octets de droite
```

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

double coords_to_point(float x, float y) {
  double p;
  float *ptr = (float *)&p;
  ptr[0] = x; // *ptr = x;
  ptr[1] = y; // *(ptr + 1) = y;
  return p;
}

double substract(double pA, double pB) {
  float *ptrA = (float *)&pA;
  float *ptrB = (float *)&pB;
  float x = ptrA[0] - ptrB[0]; // *ptrA - *ptrB
  float y = ptrA[1] - ptrB[1]; // *(ptrA + 1) - *(ptrB + 1)
  return coords_to_point(x, y);
}

float absolute(float x) {
```

```c
  if (x < 0) {
    return -x;
  }
  return x;
}

float norme1(double p) {
  float *ptr = (float *)&p;
  return absolute(ptr[0]) + absolute(ptr[1]);
}

float distance(double pA, double pB) {
  return norme1(substract(pA, pB));
}

int main() {
  double pA = coords_to_point(3.0, 4.0);
  double pB = coords_to_point(-1.0, -1.0);
  printf("%f\n", distance(pA, pB)); // -> 9.000000
  return EXIT_SUCCESS;
}
```

Solution 32 (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 31** :

```c
float abs(float x) {
  int int_x = *((int*)&x); // réinterprétation entière
  int_x &= 0x7FFFFFFF;
  float abs_x = *((float*)&int_x); // réinterprétation flottante
  return abs_x;
}
```

Solution 33 (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 32** :

```c
float fast_inverse_square_root(float x) {
  int n = *((int*)&x);
  // transformation
  float y = *((float*)&n);
  return y;
}
```

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 $-\frac{3B}{2A}$:

```c
int binary = 0x4ebe7a62;
float float_interpretation = *(float*)&binary;
int projection = (int)(float_interpretation);
printf("%x\n", projection);
```

donne le mot binaire 0x5f3d3100. Finalement :

```c
float fast_inverse_square_root(float x) {
  int n = *((int*)&x);
  n = 0x5f3d3100 - (n >> 1);
  float y = *((float*)&n);
  return y;
}
```

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[4], 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 34 (Distance Euclidienne).

```c
#include <stdio.h>
#include <stdlib.h>

int main() {
  double xA, xB, yA, yB;
  printf("Entrer xA : ");
  scanf("%lf", &xA);
  printf("Entrer yA : ");
  scanf("%lf", &yA);
  printf("Entrer xB : ");
  scanf("%lf", &xB);
  printf("Entrer yB : ");
  scanf("%lf", &yB);
  printf("||A - B||^2 = %lf", (xA - xB)*(xA - xB) + (yA - yB)*(yA - yB));
  return EXIT_SUCCESS;
}
```

Solution 35 (Somme d'entiers).

```c
#include <stdio.h>
#include <stdlib.h>

#define N 10

int main() {
```

4. Le jeu *Quake III utilise la constante 0x5f3759df.*

```c
  int somme = 0;
  int x;
  for (int i = 0; i < N; i++) {
    printf("Entrer l'entier %d : ", i + 1);
    if (scanf("%d", &x) == 1) {
      somme += x;
    } else {
      fprintf(stderr, "Il fallait entrer un entier !!!");
      return EXIT_FAILURE;
    }
  }
  printf("Somme : %d\n", somme);
  return EXIT_SUCCESS;
}
```

Solution 36 (Jeu du plus ou moins).

```c
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

#define N ...

int main() {
  srand(time(NULL));

  unsigned int mystery = rand() % N;
  unsigned int guess = -1;
  unsigned int tentatives = 0;
  printf("Un nombre mystere a ete tire entre 0 et %u inclu !\n", N-1);
  for (; guess != mystery; ++tentatives) {
    printf("Quel est le nombre mystere ?\n> ");
    fscanf(stdin, "%u", &guess); // strictement équivalent à scanf("%u", &guess);
    if (guess > mystery) {
      printf("Le nombre mystere est plus petit\n");
    }
    if (guess < mystery) {
      printf("Le nombre mystere est plus grand\n");
    }
  }
  printf("Youpiii tu as trouve le nombre mystere en %u tentatives !\n", tentatives);
  return EXIT_SUCCESS;
}
```

La stratégie optimale est identique à celle de l' **Exercice 39** pour les mêmes raisons. Il faudra au maximum $\lceil log_2(N) \rceil$ tentatives pour trouver le nombre mystère.

Tableaux statiques

Solution 37 (Les routines, direction la seconde classe !). Une fonction ne peut pas être construite pendant l'exécution du programme en C [5].

5. Enfin, en assembleur on peut, mais là n'est pas le sujet, ce n'est plus du C.

Solution 38 (Routines classiques de manipulation de tableaux).

```c
#include <stdio.h>

int somme(int array[], unsigned int length) {
  int sum = 0;
  for (unsigned int i = 0; i < length; i++) {
    sum += array[i];
  }
  return sum;
}

void display(int array[], unsigned int length) {
  for (unsigned int i = 0; i < length; i++) {
    printf("|%d|", array[i]);
  }
  printf("\n");
}

unsigned int max(int array[], unsigned int length) {
  unsigned int i_max = 0;
  for (int i = 1; i < length; i++) {
    i_max = array[i] > array[i_max] ? i : i_max;
  }
  return i_max;
}

unsigned int min(int array[], unsigned int length) {
  unsigned int i_min = 0;
  for (int i = 1; i < length; i++) {
    i_min = array[i] < array[i_min] ? i : i_min;
  }
  return i_min;
}

void swap(int array[], int i, int j) {
  int tmp = array[i];
  array[i] = array[j];
  array[j] = tmp;
}

void selection_sort(int array[], unsigned int length) {
  for (unsigned int i = 0; i < length; i++) {
    unsigned int i_min = min(array + i, length - i);
    swap(array, i_min + i, i);
  }
}
```

On pose pour $i \in [\![0; l(T)]\!]$ 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 trivialement vraies.

Hérédité : Soit $i \in [\![0; l(T)-1]\!]$. Supposons $P(i)$. Au $(i+1)^e$ tour de boucle, on a i_{min} l'indice de l'élément minimal dans $T[i:l(A)]$. Comme 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 $i \in [\![0; l(T)]\!]$, $P(i)$ 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 39 (Recherche dichotomique).

Question 1 : La démonstration est presque triviale. Supposons que $x \in T$. Il existe donc $i_x \in [\![0; l(T)-1]\!]$ 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 $l(T) \geq 1$ et le programme termine.

```c
#include <stdio.h>
#include <stdlib.h>

char is_in(int x, int array[], unsigned int length) {
  unsigned int i_left = 0;
  unsigned int i_right = length;
  unsigned int i_mid = i_left + (i_right - i_left) / 2;
  while (i_mid > i_left) {
    if (array[i_mid] > x) { // x à gauche
      i_right = i_mid;
    } else if (array[i_mid] < x) {
      i_left = i_mid;
    } else {
      return 1;
    }
    i_mid = i_left + (i_right - i_left) / 2;
  }
  return array[i_mid] == x; // traite i_mid == i_left
}
```

Solution 40 (Liste des nombres premiers). TODO

Tableaux dynamiques

Solution 41 (Conversion en binaire).

```c
#include <stdlib.h>
#include <stdio.h>

char* bin_from_nat(int n) {
  char* b = (char*)malloc(8*sizeof(int)*sizeof(char)); // 8 car sizeof(int) est en octets

  for (unsigned int i = 0; n > 0; i++) {
    b[i] = n%2;
    n /= 2;
  }
  return b;
}

void bin_display(char* b) {
  for (unsigned int i = 0; i < 8*sizeof(int); i++) {
    printf("%d", b[8*sizeof(int) - i - 1]);
  }
  printf("\n");
}

int main() {
  char* tab = bin_from_nat(5);
  bin_display(tab);
  free(tab);
  return EXIT_SUCCESS;
}
```

Tableaux multidimensionnels

Solution 42 (Afficher un tableau *2d*).

```c
#include <stdio.h>
#include <stdlib.h>

#define N 10
#define M 20

int main() {
  int array[N][M];

  for (int i = 0; i < N; i++) {
    for (int j = 0; j < M; j++) {
      printf("|%d|", array[i][j]);
    }
    printf("\n");
  }

  return EXIT_SUCCESS;
}
```

Solution 43 (Affichage du triangle).
Version généralisée pour N_LINES quelconque :

```c
#include <stdio.h>
#include <stdlib.h>

#define N_LINES 5 // hypothèse : toujours impair

int min(int a, int b) {
  return (a > b) ? b : a;
}

void print_array(int** triangle, unsigned int n_lines) {
  for (int i = 0; i < n_lines; i++) {
    for (int j = 0; j < (1 + min(N_LINES - i - 1, i)); j++) {
      printf("|%d|", triangle[i][j]);
    }
    printf("\n");
  }
}

int main() {
  int** triangle = (int**)(malloc(sizeof(int*) * N_LINES));
  for (int i = 0; i < N_LINES; i++) {
    triangle[i] = (int*)(malloc(sizeof(int) * (1 + min(N_LINES - i - 1, i))));
  }

  int cnt = 0;
  for (int i = 0; i < N_LINES; i++) {
    for (int j = 0; j <  (1 + min(N_LINES - i - 1, i)); j++) {
      triangle[i][j] = ++cnt;
    }
  }

  print_array(triangle, N_LINES);
  return EXIT_SUCCESS;
}
```

Solution 44 (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 :

$$mat[i][j] \equiv \ ^*(^*(mat + i) + j) \equiv (^*((char^*)mat + i \times sizeof(double^*)) + j \times sizeof(double))$$

```c
#include <stdio.h>
#include <stdlib.h>

double** matrix_new(unsigned int n, unsigned int m) {
  double** mat = malloc(n * sizeof(double*));
  for (unsigned int i = 0; i < n; i++) {
    mat[i] = malloc(m * sizeof(double));
  }
```

```c
  return mat;
}

void matrix_destroy(double** m, unsigned int n) {
  for (unsigned int i = 0; i < n; i++) {
    free(m[i]);
  }
  free(m);
}

void matrix_display(double** mat, unsigned int n, unsigned int m) {
  for (unsigned int i = 0; i < n; i++) {
    for (unsigned int j = 0; j < m; j++) {
      printf("|%.2lf|", mat[i][j]);
    }
    printf("\n");
  }
}

double** matrix_mul(double** a, double** b, unsigned n, unsigned m, unsigned o) {
  double** c = matrix_new(n, o);
  for (unsigned int i = 0; i < n; i++) {
    for (unsigned int j = 0; j < o; j++) {
      double x = 0;
      for (unsigned int k = 0; k < m; k++) {
        x += a[i][k]*b[k][j];
      }
      c[i][j] = x;
    }
  }
  return c;
}
```

Structures

Solution 45 (Matrices (2)).
Il suffit d'utiliser l'interface suivante, les modifications des routines sont assez triviales :

```c
#ifndef MATRIX_H_INCLUDED
#define MATRIX_H_INCLUDED

struct Matrix {
  double** mat;
  unsigned int n;
  unsigned int m;
};

struct Matrix matrix_new(unsigned int n, unsigned int m);
void matrix_destroy(struct Matrix m);
void matrix_display(struct Matrix m);
struct Matrix matrix_mul(struct Matrix a, struct Matrix b);

#endif
```

Solution 46 (Listes chaînées).

```c
#include <stdio.h>
#include <stdlib.h>

struct Node {
  struct Node* next;
  struct Node* previous;
  int value;
};

struct Node* node_new(int value) {
  struct Node* nd = malloc(sizeof(struct Node));
  if (nd == NULL) {
    fprintf(stderr, "Erreur d'allocation d'un noeud de valeur %d\n", value);
  }
  nd->next = NULL;
  nd->previous = NULL;
  nd->value = value;
  return nd;
}

struct LinkedList {
  struct Node *head;
  struct Node *tail;
  unsigned int length;
};

struct LinkedList* linkedlist_new() {
  struct LinkedList* lst = malloc(sizeof(struct LinkedList));
  if (lst == NULL) {
    fprintf(stderr, "Erreur d'allocation de la liste\n");
  }
  lst->length = 0;
  lst->head = NULL;
  lst->tail = NULL;
  return lst;
}

char linkedlist_error_access(struct LinkedList *l) {
  if (l == NULL) {
    fprintf(stderr, "Erreur : acces a un pointeur de liste NULL.\n");
    return -1;
  }
  return 0;
}

void linkedlist_destroy(struct LinkedList* l) {
  if (linkedlist_error_access(l)) {return;}
  while (!linkedlist_is_empty(l)) {
    linkedlist_pop_from_head(l);
  }
  free(l);
}

char linkedlist_is_empty(struct LinkedList* l) {
```

```
55    if (linkedlist_error_access(l)) {return;}
56    return l->length == 0;
57  }
58
59  void linkedlist_push_on_head(struct LinkedList *l, int value) {
60    if (linkedlist_error_access(l)) {return;}
61    struct Node *nd = node_new(value);
62    if (linkedlist_is_empty(l)) {
63      l->head = l->tail = nd;
64    } else {
65      l->head->previous = nd;
66      nd->next = l->head;
67      l->head = nd;
68    }
69    l->length++;
70  }
71
72  void linkedlist_push_on_tail(struct LinkedList *l, int value) {
73    if (linkedlist_error_access(l)) {return;}
74    struct Node *nd = node_new(value);
75    if (linkedlist_is_empty(l)) {
76      l->head = l->tail = nd;
77    } else {
78      l->tail->next = nd;
79      nd->previous = l->tail;
80      l->tail = nd;
81    }
82    l->length++;
83  }
84
85  int linkedlist_pop_from_head(struct LinkedList *l) {
86    if (linkedlist_error_access(l)) {return;}
87    int to_return = -1;
88    if (!linkedlist_is_empty(l)) {
89      struct Node *to_delete = l->head;
90      l->head = to_delete->next;
91      if (l->head != NULL) { // if l->length == 1
92        l->head->previous = NULL;
93      }
94      to_return = to_delete->value;
95      free(to_delete);
96      l->length--;
97    } else {
98      fprintf(stderr, "Erreur : tentative de suppression d'un element d'une liste vide.\n");
99    }
100   return to_return;
101 }
102
103 int linkedlist_pop_from_tail(struct LinkedList *l) {
104   if (linkedlist_error_access(l)) {return;}
105   int to_return = -1;
106   if (!linkedlist_is_empty(l)) {
107     struct Node *to_delete = l->tail;
108     l->tail = to_delete->previous;
109     if (l->tail != NULL) { // if l->length == 1
110       l->tail->next = NULL;
```

```
    }
    to_return = to_delete->value;
    free(to_delete);
    l->length--;
  } else {
    fprintf(stderr, "Erreur : tentative de suppression d'un element d'une liste vide.\n");
  }
  return to_return;
}

void linkedlist_display(struct LinkedList *l) {
  if (linkedlist_error_access(l)) {return;}
  struct Node *tmp = l->head;
  while (tmp != NULL) {
    printf("->%d", tmp->value);
    tmp = tmp->next;
  }
  printf("\n");
}

int main() {
  struct LinkedList *l = linkedlist_new();

  linkedlist_push_on_head(l, 5);
  linkedlist_push_on_head(l, 4);
  linkedlist_push_on_tail(l, 6);

  linkedlist_display(l);

  linkedlist_destroy(l);
  l = NULL;
  return EXIT_SUCCESS;
}
```

Modulation et entêtes

Solution 47 (Un module de listes chaînées).
On écrit le code des fonctions de l'exercice précédent dans un fichier *"linkedlist.c"*[6] que l'on fait débuter 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 :

```
#ifndef LINKEDLIST_H_INCLUDED
#define LINKEDLIST_H_INCLUDED

#include <stdlib.h>
#include <stdio.h>

struct Node {
  struct Node* next;
  struct Node* previous;
  int value;
};
```

6. Dans le répertoire de travail du fichier *"main.c"*

```c

struct Node* node_new(int value);

struct LinkedList {
  struct Node *head;
  struct Node *tail;
  unsigned int length;
};

struct LinkedList* linkedlist_new();
void linkedlist_destroy(struct LinkedList* l);
char linkedlist_error_access(struct LinkedList *l);
char linkedlist_is_empty(struct LinkedList* l);
void linkedlist_push_on_head(struct LinkedList *l, int value);
void linkedlist_push_on_tail(struct LinkedList *l, int value);
int linkedlist_pop_from_head(struct LinkedList *l);
int linkedlist_pop_from_tail(struct LinkedList *l);
void linkedlist_display(struct LinkedList *l);

#endif
```

Il conserve la même fonction main du fichier *"main.c"* avec une inclusion du module :

```c
// stdio est non nécessaire ici
#include <stdlib.h> // juste pour EXIT_SUCCESS

#include "linkedlist.h"

int main() {

  struct LinkedList *l = linkedlist_new();

  linkedlist_push_on_head(l, 5);
  linkedlist_push_on_head(l, 4);
  linkedlist_push_on_tail(l, 6);

  linkedlist_display(l);

  return EXIT_SUCCESS;
}
```

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 48 (Calculatrice).

```c
#include <stdio.h>
#include <stdlib.h>

```

```c
int main() {
  char op;
  double x;
  double y;
  while (1) {
    printf("> ");
    if (scanf("%lf %c %lf", &x, &op, &y) == 3) {
      switch (op) {
      case '+':
        printf("%lf\n", x + y);
        break;
      case '-':
        printf("%lf\n", x - y);
        break;
      case '/':
        printf("%lf\n", x / y);
        break;
      case '*':
        printf("%lf\n", x * y);
        break;
      }
    } else {
      fprintf(stderr, "Erreur !\n");
      return EXIT_FAILURE;
    }
  }

  return EXIT_SUCCESS;
}
```

Solution 49 (AtoI).

```c
int atoi(char* string) {
  if (string == NULL) {
    return 0;
  }
  int x = 0;
  while (*string == ' ') {
    string++;
  }
  char sign = (*string == '-');
  if (*string == '-' || *string == '+') {
    string++;
  }
  while (*string <= '9' && *string >= '0') {
    x = 10*x + (*string++ - '0');
  }
  return (sign) ? -x : x;
}
```

Interagir avec les flux de fichiers

8.2.2 Concepts avancés

Passage d'arguments au programme

Solution 50 (Liste des arguments).

```c
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char **argv) {
  for (int i = 0; i < argc; i++) {
    printf("%s\n", argv[i]);
  }
  return EXIT_SUCCESS;
}
```

Solution 51 (Un cat minimaliste).

```c
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char **argv) {
  if (argc < 2) {
    fprintf(stderr, "Il faut donner en argument du programme le nom du fichier à afficher.\n");
  }
  FILE *f = fopen(argv[1], "r");
  if (f == NULL) {
    fprintf(stderr, "Erreur d'ouverture du fichier %s\n", argv[1]);
  }
  char buffer[1024] = {0};
  while (fread(buffer, 1, 1023, f) == 1023) {
    printf("%s", buffer);
  }
  printf("%s", buffer);
  return EXIT_SUCCESS;
}
```

Pointeurs de routines

Solution 52 (Mapping).

```c
void array_map(int array[], unsigned int length, void (*const f)(int*)) {
  for (unsigned int i = 0; i < length; i++) {
    f(&array[i]);
  }
}

void linkedlist_map(struct LinkedList *l, void (*const f)(int*)) {
  if (linkedlist_error_access(l)) {return;}
  struct Node *tmp = l->head;
  while (tmp != NULL) {
```

```
    f(tmp->value);
    tmp = tmp->next;
  }
}
```

Routines variadiques

Solution 53 (À un doigt du zéro).

```
unsigned int mult(unsigned int first, ...) {
  unsigned int product = first;
  va_list args;
  va_start(args, first);
  for (int x = va_arg(args, int); x != 0; x = va_arg(args, int)) {
    product *= x;
  }
  va_end(args);
  return product;
}
```

BIBLIOGRAPHIE

Livres

[1] A. V. Aho et al. *Compilers : Principles, Techniques, and Tools*. Pearson Education Inc., 2006. ISBN : 0201100886. URL : https://drive.google.com/file/d/11t2AZMkYnSqhaufvIU1orxafpR0mca2X/view?usp=drive_link.

[3] Olivier Bournez. *Fondements de l'informatique : Logique, modèles et calculs*. École Polytechnique, juill. 2024. URL : https://drive.google.com/file/d/11t2AZMkYnSqhaufvIU1orxafpR0mca2X/%20view?usp=drive_link.

[4] 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/1bkLb3OByL_UaBNz-ksr03WliZ6mnuiy-/view?usp=drive_link.

[7] 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.

[8] M. Kerrisk. *The Linux Programming Interface*. learning. 2010. ISBN : 9781593272203. URL : https://drive.google.com/file/d/1CDfK3cz0xKj-E0bJOFTiKaDFwi2f3RgI/view?usp=drive_link.

[9] Donald Knuth. *The Art of Computer Programming*. Addison-Wesley, 2022. ISBN : 0201038013. URL : https://drive.google.com/drive/folders/1n0Wh2rfAy49rzrFhdS3OOZ8hPma6WN8P?usp=drive_link.

[10] John R. Levine. *Linkers and Loaders*. Morgan-Kauffman, oct. 1999. ISBN : 1558604960. URL : https://drive.google.com/file/d/1EtAHWMljpFzL8Y6jlJ5H32-1L2AJ_eF9/view?usp=drive_link.

[11] 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.

[12] D. Salomon. *Assemblers and Loaders*. Ellis Horwood Ltd, fév. 1993. ISBN : 0130525642. URL : https://drive.google.com/file/d/1IAtRVoKwOkhZLkjWorFNVQpquWi5zTHR/view?usp=drive_link.

[13] Alexander Stepanov et Paul McJones. *Elements of Programming*. Pearson Education Inc., juin 2019. ISBN : 0578222141. URL : https://drive.google.com/file/d/1LNMhVp_q9z5KnmLCC05Q1eo3rQ17OSlS/view?usp=drive_link.

Manuels et documentation

[5] *C11 Specificatioon*. Avr. 2011. URL : https://drive.google.com/file/d/1nUx1JETBBm7zyQR0oviLO2veJtb5MpLD/view?usp=drive_link.

[6] *GNU Make*. Fév. 2023. URL : 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.

[14] Tyler Whitney et al. *Documentation sur le langage C*. 2023. URL : https://learn.microsoft.com/fr-fr/cpp/c-language/.