Le C pour l'ingénieur

Prof. Yves Chevallier

août 02, 2020

Table des matières

1 Introduction **1**
- 1.1 Historique . . . 1
- 1.2 Standardisation . . . 2
- 1.3 Environnement de développement . . . 3
- 1.4 L'Anglais . . . 4
- 1.5 Apprendre à pêcher . . . 5
- 1.6 Programmation texte structurée . . . 6
- 1.7 Les paradigmes de programmation . . . 7
- 1.8 Cycle de développement . . . 7
- 1.9 Cycle de compilation . . . 8
- 1.10 Une affaire de consensus . . . 11
- 1.11 Hello World ! . . . 13

2 La programmation **17**
- 2.1 Algorithmique . . . 18
- 2.2 Programmation . . . 19
- 2.3 Calculateur . . . 21
- 2.4 Ordinateur . . . 21
- 2.5 Historique . . . 22
- 2.6 Fonctionnement de l'ordinateur . . . 23

3 Généralités du langage **25**
- 3.1 L'alphabet . . . 25
- 3.2 Fin de lignes (EOL) . . . 26
- 3.3 Mots clés . . . 27
- 3.4 Identificateurs . . . 28
- 3.5 Variables . . . 30
- 3.6 Les constantes . . . 32
- 3.7 Constantes littérales . . . 32
- 3.8 Operateur d'affectation . . . 33
- 3.9 Commentaires . . . 35

4 Numération **39**
- 4.1 Bases . . . 39
- 4.2 Entiers relatifs . . . 46
- 4.3 Opérations logiques . . . 49

5 Opérateurs **55**
- 5.1 Opérateurs relationnels . . . 56

5.2 Opérateurs arithmétiques . . . 56
5.3 Opérateurs bit à bit . . . 57
5.4 Opérateurs d'affectation . . . 57
5.5 Opérateurs logiques . . . 57
5.6 Opérateurs d'incrémentation . . . 58
5.7 Opérateur ternaire . . . 58
5.8 Opérateur de transtypage . . . 58
5.9 Opérateur séquentiel . . . 58
5.10 Opérateur sizeof . . . 59
5.11 Les opérateurs logiques . . . 59
5.12 Les opérateurs bit-à-bit . . . 60
5.13 Les opérateurs d'incrémentation (++) et de décrémentation (–) . . . 62
5.14 Priorité des opérateurs . . . 63
5.15 Valeurs gauche . . . 65
5.16 Optimisation . . . 65

6 Types de données **69**
6.1 Typage . . . 69
6.2 Stockage et interprétation . . . 70
6.3 Boutisme . . . 71
6.4 Les nombres entiers . . . 72
6.5 Les nombres réels . . . 78
6.6 Les caractères . . . 82
6.7 Chaîne de caractères . . . 85
6.8 Les booléens . . . 87
6.9 Énumérations . . . 87
6.10 Type incomplet . . . 89
6.11 Type vide (*void*) . . . 89
6.12 Promotion implicite . . . 89

7 Structures de contrôle **99**
7.1 Séquences . . . 99
7.2 Les embranchements . . . 100
7.3 Les boucles . . . 105
7.4 Les sauts . . . 109

8 Programmes et Processus **115**
8.1 Qu'est-ce qu'un programme ? . . . 115
8.2 Arguments et options . . . 121
8.3 Fonction main . . . 124
8.4 Entrées sorties standards . . . 125
8.5 Boucle d'attente . . . 127

9 Entrées Sorties **129**
9.1 Sorties non formatées . . . 129
9.2 Sorties formatées . . . 130
9.3 printf . . . 132
9.4 Entrées formatées . . . 134

10 Fonctions **149**
10.1 Conventions d'appel . . . 151
10.2 Prototype . . . 155
10.3 Syntaxe . . . 156
10.4 Paramètres . . . 157
10.5 Récursion . . . 158
10.6 Mémoïsation . . . 160

11 Types composites **165**
11.1 Tableaux . . . 165
11.2 Chaînes de caractères . . . 174
11.3 Structures . . . 175
11.4 Champs de bits . . . 184
11.5 Unions . . . 185
11.6 Création de type . . . 186
11.7 Compound Literals . . . 186

12 Les fichiers **189**
12.1 Système de fichiers . . . 189
12.2 Format d'un fichier . . . 190
12.3 Ouverture d'un fichier . . . 191
12.4 Navigation dans un fichier . . . 194
12.5 Lecture / Écriture . . . 195
12.6 Buffer de fichier . . . 195
12.7 Fichiers et Flux . . . 197
12.8 Formats de sérialisation . . . 198

13 Gestion de la mémoire **203**
13.1 Allocation statique . . . 203
13.2 Allocation dynamique . . . 204
13.3 Mémoire de programme . . . 204
13.4 La pile . . . 205
13.5 Allocation dynamique sur le tas . . . 207
13.6 Variables automatiques . . . 209
13.7 Fragmentation mémoire . . . 210

14 Pointeurs **213**
14.1 Pointeur simple . . . 215
14.2 Arithmétique de pointeurs . . . 216
14.3 Pointeur et chaînes de caractères . . . 217
14.4 Structures et pointeurs . . . 217
14.5 Transtypage de pointeurs (cast) . . . 220
14.6 Pointeurs de fonctions . . . 223
14.7 La règle gauche-droite . . . 224
14.8 Initialisation par transtypage . . . 225
14.9 Enchevêtrement ou *Aliasing* . . . 225

15 Bibliothèques **229**
15.1 Exemple : libgmp . . . 230

15.2 Exemple : ncurses . . . 233
15.3 Bibliothèques statiques . . . 234
15.4 Bibliothèques dynamiques . . . 235
15.5 Bibliothèques standard . . . 236
15.6 Autres bibliothèques . . . 239
15.7 MacOS X Library . . . 241

16 Préprocesseur **243**
16.1 Phases de traduction . . . 245
16.2 Extensions des fichiers . . . 245
16.3 Inclusion de fichiers . . . 245
16.4 Définitions . . . 246
16.5 Macros . . . 248
16.6 Debogage . . . 248
16.7 Caractère d'échappement . . . 248
16.8 Macros . . . 249
16.9 Directives conditionnelles . . . 249
16.10 Commentaires . . . 251

17 Algorithmes et conception **253**
17.1 Complexité d'un algorithme . . . 254
17.2 Machines d'états . . . 256
17.3 Diagrammes visuels . . . 256
17.4 Récursivité . . . 256
17.5 Programmation dynamique . . . 257
17.6 Algorithmes de tris . . . 257
17.7 Type d'algorithmes . . . 257

18 Compilation séparée **261**
18.1 Translation unit . . . 261
18.2 Diviser pour mieux régner . . . 261
18.3 Module logiciel . . . 263
18.4 Compilation avec assemblage différé . . . 263
18.5 Fichiers d'en-tête (*header*) . . . 264

19 Portée et visibilité **269**
19.1 Espace de nommage . . . 269
19.2 Portée . . . 269
19.3 Visibilité . . . 271
19.4 Qualificatif de type . . . 273

20 Qualité et Testabilité **277**
20.1 Hacking . . . 277
20.2 Tests unitaires . . . 279
20.3 Tests fonctionnels . . . 279
20.4 Framework de tests . . . 279

21 Structures de données **281**
21.1 Types de données abstraits . . . 281

21.2 Tableau dynamique . . . 283
21.3 Buffer circulaire . . . 287
21.4 Listes chaînées . . . 289
21.5 Liste doublement chaînée . . . 296
21.6 Liste chaînée déroulée (Unrolled linked list) . . . 297
21.7 Arbre binaire de recherche . . . 297
21.8 Heap . . . 298
21.9 Queue prioritaire . . . 300
21.10 Tableau de Hachage . . . 301
21.11 Piles ou LIFO (*Last In First Out*) . . . 307
21.12 Queues ou FIFO (*First In First Out*) . . . 307
21.13 Performances . . . 308

22 Avancé **311**
22.1 Points de séquences . . . 311
22.2 Complément sur les variables initialisées . . . 311
22.3 Binutils . . . 312
22.4 Format Q . . . 313
22.5 Mémoire partagée . . . 316
22.6 Collecteur de déchets (*garbage collector*) . . . 318

23 Pièges **321**
23.1 Préprocesseur . . . 321
23.2 Erreurs de syntaxe . . . 323

24 Philosophie **325**
24.1 Rasoir d'Ockham . . . 325
24.2 Principes de programmation . . . 326
24.3 Zen de Python . . . 327
24.4 The code taste . . . 328
24.5 L'odeur du code . . . 329

A Visual Studio Code **333**

B Grammaire C **335**

C Ligne de commande **347**
C.1 Pipe . . . 347

D Environnement de développement **349**

E Fiches d'unités **351**
E.1 Informatique 1 . . . 351
E.2 Informatique 2 . . . 356
E.3 Modalités d'évalutation et de validation . . . 360

F Laboratoires **363**
F.1 Protocole . . . 363
F.2 Evaluation . . . 363
F.3 Directives . . . 363

F.4 Format de rendu 364
F.5 Anatomie d'un travail pratique 364

G Résumé **369**
G.1 Introduction 369
G.2 Cycle de développement 369
G.3 Cycle de compilation 369
G.4 Make 370
G.5 Linux/POSIX 371
G.6 Diagramme de flux 371
G.7 Caractères non imprimables 371
G.8 Fin de lignes 373
G.9 Identificateurs 373
G.10 Variable 374
G.11 Constantes littérales 375
G.12 Commentaires 375
G.13 Fonction main 375
G.14 Valeur gauche 377
G.15 Caractères 379
G.16 Chaîne de caractère 379
G.17 Booléens 380
G.18 Promotion implicite 380
G.19 Transtypage 380
G.20 Séquence 381
G.21 Si, sinon 381
G.22 Si, sinon si, sinon 381
G.23 Boucle For 382
G.24 Boucle While 382
G.25 `printf` 382
G.26 Masque binaire 384
G.27 Permuter deux variables sans valeur intermédiaire 384

Bibliographie **391**

Colophon **393**

Préambule

Cet ouvrage est destiné aux étudiants de première année Bachelor HEIG-VD, département TIN et filières Génie électrique. Il est une introduction à la programmation en C. Il couvre la matière vue durant le cycle des cours *Info1* et *Info2*.

Le contenu de ce cours est calqué sur les fiches d'unités de cours et de modules suivantes :

- Module **InfoMicro** (`InfoMicro`)
- Unité **Informatique 1** (*Info1*)
- Unité **Informatique 2** (*Info2*)

La version actuelle : v0.2.7-3-g16f703f

Date : août 02, 2020

Chapitre 1

Introduction

1.1 Historique

Le langage de programmation **C** est la suite naturelle du langage **B** créé dans la toute fin des années soixante par un grand pionnier de l'informatique moderne : Ken Thompson.

Le langage C a été inventé en 1972 par Brian Kernighan et Dennis Ritchie. Ils sont les concepteurs du système d'exploitation UNIX et ont créé ce nouveau langage pour faciliter leurs travaux de développement logiciel. La saga continue avec Bjarne Stroustrup qui décide d'étendre C en apportant une saveur nouvelle : la programmation orientée objet (OOP), qui fera l'objet d'un cours à part entière. Ce C amélioré voit le jour en 1985.

Il faut attendre 1989 pour que le langage C fasse l'objet d'une normalisation par l'ANSI. L'année suivante le comité ISO ratifie le standard *ISO/IEC 9899 :1990* communément appelé **C90**.

Les années se succèdent et le standard évolue pour soit corriger certaines de ses faiblesses soit pour apporter de nouvelles fonctionnalités.

Cinquante ans plus tard, C est toujours l'un des langages de programmation les plus utilisés, car il allie une bonne vision de haut niveau tout en permettant des manipulations de très bas niveau, de fait il est un langage de choix pour les applications embarquées à microcontrôleurs, ou lorsque l'optimisation du code est nécessaire pour obtenir de bonnes performances tels que les noyaux des systèmes d'exploitation comme le noyau Linux (Kernel) ou le noyau Windows.

Il faut retenir que C est un langage simple et efficace. Votre machine à café, votre voiture, vos écouteurs Bluetooth ont très probablement été programmés en C.

Fig. 1.1 – Les pères fondateurs du C

1.2 Standardisation

Vous l'aurez compris à lecture de cette introduction, le langage C possède un grand historique, et il a fallu attendre près de 20 ans après sa création pour voir apparaître la première standardisation internationale.

Le standard le plus couramment utilisé en 2019 est encore C99.

Notation courte	Standard international	Date
C	n/a	1972
K&R C	n/a	1978
C89 (ANSI C)	ANSI X3.159-1989	1989
C90	ISO/IEC 9899 :1990	1990
C99	ISO/IEC 9899 :1999	1999
C11	ISO/IEC 9899 :2011	2011
C17/C18	ISO/IEC 9899 :2018	2018

En substance, **C18** n'apporte pas de nouvelles fonctionnalités au langage, mais vise à clarifier de nombreuses zones d'ombres laissées par **C11**.

C11 apporte peu de grands changements fondamentaux pour le développement sur microcontrôleur par rapport à **C99** et ce dernier reste de facto le standard qu'il est souhaité de respecter dans l'industrie.

Note : Vous entendrez ou lirez souvent des références à **ANSI C** ou **K&R**, préférez plutôt une compatibilité avec **C99** au minimum.

Le standard est lourd, difficile à lire et avec 552 pages pour **C99**, vous n'aurez probablement jamais le moindre plaisir à y plonger les yeux. L'investissement est pourtant parfois nécessaire pour comprendre certaines subtilités du langage qui sont rarement expliquées dans les livres. Pourquoi diable écrire un livre qui détaille l'implémentation C alors qu'il existe déjà ? Exercice

Ouvrez le standard C99 et cherchez la valeur maximale possible de la constante `ULLONG_MAX`. Que vaut-elle ?

Au paragraphe §5.2.4.2.1-1 on peut lire que `ULLONG_MAX` est encodé sur 64-bits et donc que sa valeur est $2^{64} - 1$ donc *18'446'744'073'709'551'615*.

1.3 Environnement de développement

Un développeur logiciel passe son temps devant son écran à étudier, et écrire du code et bien qu'il pourrait utiliser un éditeur de texte tel que Microsoft Word ou Notepad, il préfèrera des outils apportant davantage d'interactivité et d'aide au développement. Les *smartphones* disposent aujourd'hui d'une fonctionnalité de suggestion automatique de mots ; les éditeurs de texte orienté programmation disposent de fonctionnalités similaires qui complètent automatiquement le code selon le contexte.

Un autre composant essentiel de l'environnement de développement est le **compilateur**. Il s'agit généralement d'un ensemble de programmes qui permettent de convertir le **code** écrit en un programme exécutable. Ce programme peut-être par la suite intégré dans un *smartphone*, dans un système embarqué sur un satellite, sur des cartes de prototypage comme un Raspberry PI, ou encore sur un ordinateur personnel.

L'ensemble des outils nécessaire à créer un produit logiciel est appelé chaîne de compilation, plus communément appelée **toolchain**.

Un environnement de développement intégré, ou IDE pour *Integrated development environment* comporte généralement un éditeur de code ainsi que la toolchain associée.

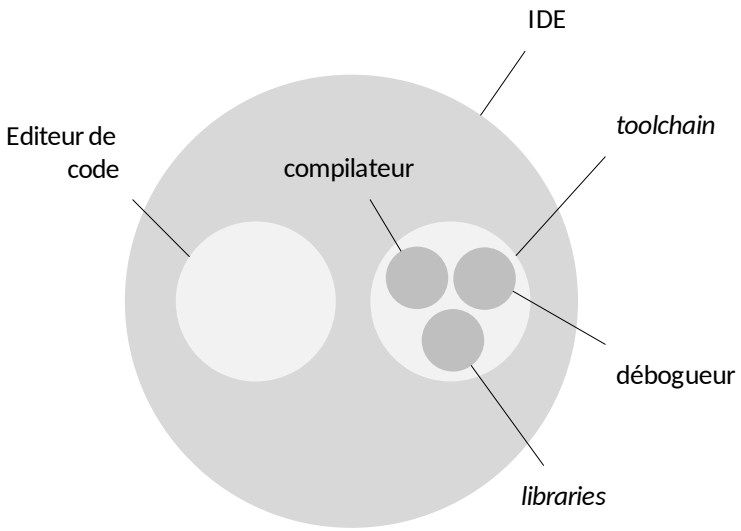


Fig. 1.2 – Représentation graphique des notions de compilateur, IDE, toolchain, ...

À titre d'exemple on peut citer quelques outils bien connus des développeurs. Choisissez celui que vous pensez être le plus adapté à vos besoins, consultez l'internet, trouvez votre optimal :

Microsoft Visual Studio Un **IDE** très puissant disponible sous Microsoft Windows exclusivement. Il supporte de nombreux langages de programmation comme C, C++, C# ou Python.

Code ::Blocks Un **IDE** libre et multi plate-forme pour C et C++, une solution simple pour développer rapidement.

Microsoft Visual Studio Code (VsCode) Un **éditeur de code** *open-source* multi plates-formes disponible sur Windows, MacOS et Linux.

GCC Un **compilateur** *open-source* utilisé sous Linux et MacOS.

CLANG Un **compilateur** *open-source* gagnant en popularité, une alternative à GCC.

Vim Un **éditeur de code** *open-source* multi-usage à la courbe d'apprentissage très raide et installé par défaut sur la plupart des distributions Unix/Linux. Il est l'évolution de *ed*, puis *ex* puis *vi* puis *vim*.

Ed Prononcé /i di / (hidi), il s'agit du tout premier éditeur de texte développé en 1969 faisant parti des trois premiers éléments du système UNIX : l'assembleur, l'éditeur et le *shell*. Il n'est pas interactif, il n'a pas de coloration syntaxique, il est absolument obscure dans son fonctionnement mais bientôt 50 ans après, il fait toujours parti de la norme POSIX et donc disponible sur tout système compatible. Bref, ne l'utilisez pas...

Exercice

Un ami vous parle d'un outil utilisé pour le développement logiciel nommé **Eclipse**. De quel type d'outil s'agit-il ?

Eclipse est un IDE. Il n'intègre donc pas de chaîne de compilation et donc aucun compilateur.

1.4 L'Anglais

En programmation, quel que soit le langage utilisé, la langue **anglaise** est omniprésente. D'une part les mots clés des langages de programmation sont majoritairement empruntés à l'anglais, mais souvent les outils de développement ne sont disponibles qu'en anglais. Il existe une raison à cela. Un article de journal publié dans une revue locale sera certainement lu par madame Machin et monsieur Bidule, mais n'aura aucun intérêt pour les habitants de l'antipode néo-zélandais. En programmation, le code se veut **réutilisable** pour économiser des coûts de développement. On réutilise ainsi volontiers des algorithmes écrits par un vénérable japonais, ou une bibliothèque de calcul matriciel développée en Amérique du Sud. Pour faciliter la mise en commun de ces différents blocs logiciels et

surtout pour que chacun puisse dépanner le code des autres, il est essentiel qu'une langue commune soit choisie et l'anglais est le choix le plus naturel.

Aussi dans cet ouvrage, l'anglais sera privilégié dans les exemples de code et les noms des symboles (variables, constantes ...), les termes techniques seront traduits lorsqu'il existe un consensus établi sinon l'anglicisme sera préféré. Il m'est d'ailleurs difficile, bien que ce cours soit écrit en français de parler de *feu d'alerte* en lieu et place de *warning* car si l'un est la traduction ad-hoc de l'autre, la terminologie n'a rien à voir et préfère, au risque d'un affront avec l'Académie, préserver les us et coutumes des développeurs logiciels.

Un autre point méritant d'être mentionné est la constante interaction d'un développeur avec internet pour y piocher des exemples, chercher des conseils, ou de l'aide pour utiliser des outils développés par d'autres. De nombreux sites internet, la vaste majorité en anglais, sont d'une aide précieuse pour le développeur. On peut ainsi citer :

https://stackoverflow.com/ Aujourd'hui le plus grand portail de questions/réponses dédié à la programmation logicielle

https://github.com/ Un portail de partage de code

https://scholar.google.ch/ Un point d'entrée essentiel pour la recherche d'articles scientifiques

https://linux.die.net/man/ La documentation (*man pages*) des commandes et outils les plus utilisés dans les environnements MacOS/Linux/Unix et POSIX compatibles.

Exercice

Combien y-a-t-il eu de questions posées en C sur le site Stack Overflow ?

Il suffit pour cela de se rendre sur le site de Stackoverflow et d'accéder à la liste des tags. En 2019/07 il y eut 307'669 questions posées.

Seriez-vous capable de répondre à une question posée ?

1.5 Apprendre à pêcher

Un jeune homme s'en va à la mer avec son père et lui demande : papa, j'ai faim, comment ramènes-tu du poisson ? Le père fier, lance sa ligne à la mer et lui ramène un beau poisson. Plus tard, alors que le jeune homme revient d'une balade sur les estrans, il demande à son père : papa, j'ai faim, me ramènerais-tu du poisson ? Le père, sort de son étui sa plus belle canne et l'équipant d'un bel hameçon lance sa ligne à la mer et ramène un gros poisson. Durant longtemps, le jeune homme mange ainsi à sa faim cependant que le père ramène du poisson pour son fils.

Un jour, alors que le fils invective son père l'estomac vide, le père annonce. Fils, il est temps pour toi d'apprendre à pêcher, je peux te montrer encore longtemps comment je ramène du poisson, mais ce ne serait pas t'aider, voici donc cette canne et cet hameçon.

Le jeune homme tente de répéter les gestes de son père, mais il ne parvient pas à ramener le poisson qui le rassasierait. Il demande à son père de l'aide que ce dernier refuse. Fils, c'est par la pratique et avec la faim au ventre que tu parviendras à prendre du poisson,

persévère et tu deviendras meilleur pêcheur que moi, la lignée ainsi assurée de toujours manger à sa faim.

La morale de cette histoire est plus que jamais applicable en programmation, confier aux expérimentés l'écriture d'algorithmes compliqués, ou se contenter d'observer les réponses des exercices pour se dire : j'ai compris ce n'est pas si compliqué, est une erreur, car pêcher ou expliquer comment pêcher n'est pas la même chose.

Aussi, cet ouvrage se veut être un guide pour apprendre à apprendre le développement logiciel et non un guide exhaustif du langage car le standard C99/C11 est disponible sur internet ainsi que le K&R qui reste l'ouvrage de référence pour apprendre C. Il est donc inutile de paraphraser les exemples donnés quand internet apporte toutes les réponses, pour tous les publics du profane réservé au hacker passionné.

1.6 Programmation texte structurée

Le C comme la plupart des langages de programmation utilise du texte structuré, c'est-à-dire que le langage peut être défini par un **vocabulaire**, une **grammaire** et se compose d'un **alphabet**.

À l'inverse des langages naturels comme le Français, un langage de programmation est un langage formel et se veut exact dans sa grammaire et son vocabulaire, il n'y a pas de cas particuliers ni d'ambiguïtés possibles dans l'écriture.

Les **compilateurs**, sont ainsi construits autour d'une grammaire du langage qui est réduite au minimum par souci d'économie de mémoire, pour taire les ambiguïtés et accroître la productivité du développeur.

L'exemple suivant est un pseudo-code utilisant une grammaire simple :

```
POUR CHAQUE oeuf DANS le panier :
    jaune, blanc ← CASSER(oeuf)
    omelette ← MELANGER(jaune, blanc)
    omelette_cuite ← CUIRE(omelette)

SERVIR(omelette_cuite)
```

La structure de la phrase permettant de traiter tous les éléments d'un ensemble d'éléments peut alors s'écrire :

```
POUR CHAQUE <> DANS <>:
    <>
```

Où les **<>** sont des marques substitutives (placeholder) qui seront remplacées par le développeur par ce qui convient.

Les grammaires des langages de programmation sont souvent formalisées à l'aide d'un méta-langage, c'est-à-dire un langage qui permet de décrire un langage. La grammaire du langage C utilisé dans ce cours peu ainsi s'exprimer en utilisant la forme Backus-Naur ou **BNF** disponible en annexe.

1.7 Les paradigmes de programmation

Chaque langage de programmation que ce soit C, C++, Python, ADA, COBOL, Lisp et sont d'une manière générale assez proche les uns des autres. Nous citions par exemple le langage B, précurseur du C (c.f. Section 1.1). Ces deux langages, et bien que leurs syntaxes soient différentes, ils demeurent assez proches, comme l'espagnol et l'italien qui partagent des racines latines. En programmation on dit que ces langages partagent le même paradigme de programmation.

Certains paradigmes sont plus adaptés que d'autres à la résolution de certains problèmes et de nombreux langages de programmation sont dit **multi-paradigmes**, c'est-à-dire qu'ils supportent différents paradigmes.

Nous citions plus haut le C++ qui permet la programmation orientée objet, laquelle est un paradigme de programmation qui n'existe pas en C.

Ce qu'il est essentiel de retenir c'est qu'un langage de programmation peut aisément être substitué par un autre pour autant qu'ils s'appuient sur les mêmes paradigmes.

Le langage C répond aux paradigmes suivants :
- Impératif : programmation en séquences de commandes
- Structuré : programmation impérative avec des structures de contrôle imbriquées
- Procédural : programmation impérative avec appels de procédures

Le C++ quant à lui apporte les paradigmes suivants à C :
- Fonctionnel
- Orienté objet

Des langages de plus haut niveau comme Python ou C# apportent d'avantages de paradigmes comme la programmation réflective.

1.8 Cycle de développement

Le cycle de développement logiciel comprend la suite des étapes menant de l'étude et l'analyse d'un problème jusqu'à la réalisation d'un programme informatique exécutable. Dans l'industrie, il existe de nombreux modèles comme le Cycle en V ou le modèle en cascade. Quel que soit le modèle utilisé, il comprendra les étapes suivantes :

1. **Étude** et analyse du problème
2. Écriture d'un **cahier des charges** (spécifications)
3. Écriture de **tests** à réaliser pour tester le fonctionnement du programme
4. **Conception** d'un algorithme
5. **Transcription** de cet algorithme en utilisant le langage C
6. **Compilation** du code et génération d'un exécutable
7. **Test** de fonctionnement
8. **Vérification** que le cahier des charges est respecté
9. **Livraison** du programme

Mises à part la dernière étape où il n'y a pas de retour en arrière possible, les autres étapes sont **itératives**. Il est très rare d'écrire un programme juste du premier coup. Durant tout le cycle de développement logiciel, des itérations successives sont faites pour permettre d'optimiser le programme, de résoudre des bogues, d'affiner les spécifications, d'écrire davantage de tests pour renforcer l'assurance d'un bon fonctionnement du programme et éviter une *coulée de lave*.

Le modèle en cascade suivant résume le cycle de développement d'un programme. Il s'agit d'un modèle simple, mais qu'il faut garder à l'esprit que ce soit pour le développement d'un produit logiciel que durant les travaux pratiques liés à ce cours.

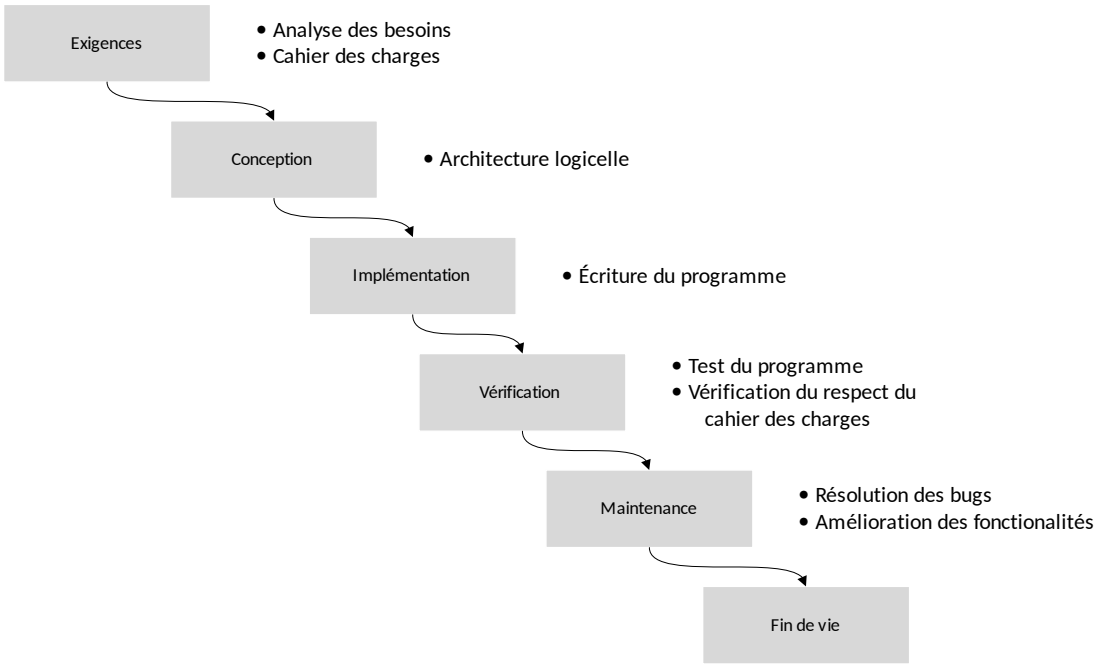


Fig. 1.3 – Modèle en cascade

1.9 Cycle de compilation

Le langage C à une particularité que d'autres langages n'ont pas, c'est-à-dire qu'il comporte une double grammaire. Le processus de compilation s'effectue donc en deux passes.

1. Préprocesseur
2. Compilation du code

Vient ensuite la phase d'édition des liens ou *linkage* lors de laquelle l'exécutable binaire est créé.

Voyons plus en détail chacune de ces étapes.

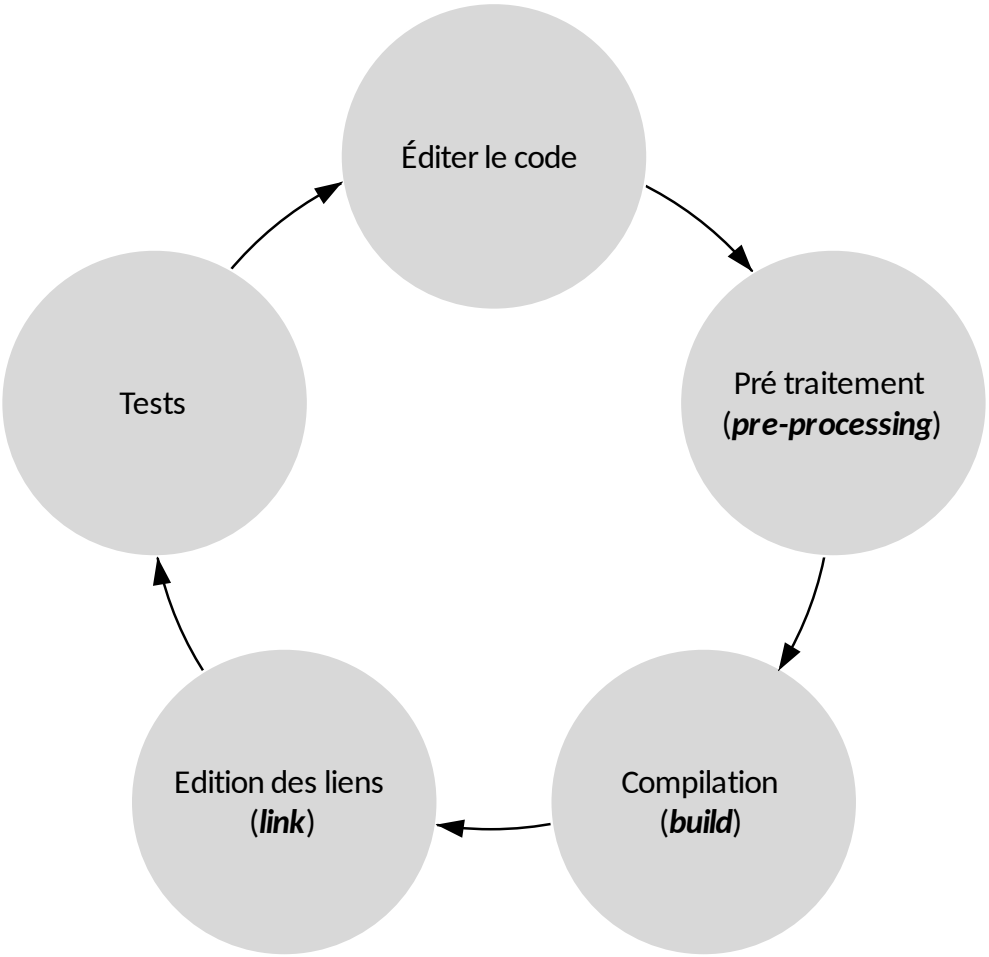


Fig. 1.4 – Cycle de compilation illustré

1.9.1 Préprocesseur (*pre-processing*)

La phase de *preprocessing* permet de générer un fichier intermédiaire en langage C dans lequel toutes les instructions nécessaires à la phase suivante sont présentes. Le *preprocessing* réalise :

- Le remplacement des définitions par leurs valeurs (`#define`),
- Le remplacement des fichiers inclus par leurs contenus (`#include`),
- **La conservation ou la suppression des zones de compilation** conditionnelles (`#if/#ifdef/#elif/#else/#endif`).
- La suppression des commentaires (`/* ... */`, `// ...`)

Avec `gcc` il est possible de demander que l'exécution du préprocesseur en utilisant l'option `-E`.

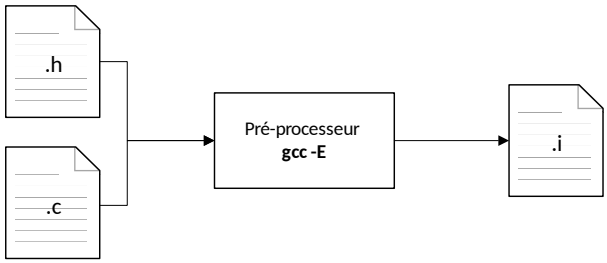
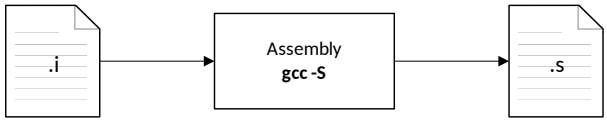


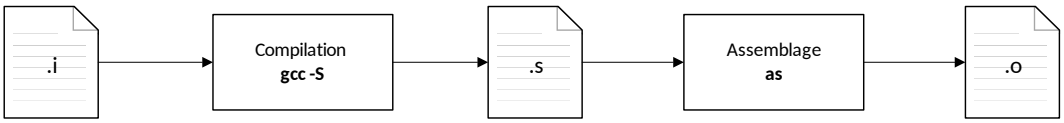
Fig. 1.5 – Processus de pré-processing

1.9.2 Compilation (*build*)

La phase de compilation consiste en une analyse syntaxique du fichier à compiler puis en sa traduction en langage assembleur pour le processeur cible. Le fichier généré est un fichier binaire (extension *.o* ou *.obj*) qui sera utilisé pour la phase suivante. Lors de la *compilation*, des erreurs peuvent survenir et empêcher le déroulement complet de la génération de l'exécutable final. Là encore, la correction des erreurs passe toujours par un examen minutieux des messages d'erreur, en commençant toujours par le premier.

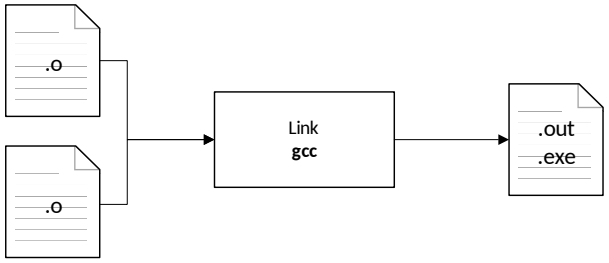
Avec `gcc` il est possible de ne demander que l'assemblage d'un code avec l'option `-S`.





1.9.3 Édition de liens (*link*)

La phase d'édition de liens permet de rassembler le fichier binaire issu de la compilation et les autres fichiers binaires nécessaires au programme pour former un exécutable complet. Les autres fichiers binaires sont appelés des **librairies**. Elles peuvent appartenir au système (installée avec l'environnement de développement) ou provenir d'autres applications avec lesquelles votre programme doit interagir. Lors de l'édition de liens, des erreurs peuvent survenir et empêcher le déroulement complet de génération de l'exécutable final. Là encore, la correction des erreurs passe toujours par un examen minutieux des messages d'erreur, en commençant toujours par le premier.



1.10 Une affaire de consensus

En informatique comme dans la société humaine, il y a les religieux, les prosélytes, les dogmatiques, les fanatiques, les contestataires et les maximalistes. Le plus souvent les motifs de fâcheries concernent les outils que ces derniers utilisent et ceux dont on doit taire le nom. Ils se portent parfois sur les conventions de codage à respecter, l'encodage des fichiers, le choix de l'EOL, l'interdiction du `goto`, le respect inconditionnel des règles MISRA. Il existe ainsi de longues guerres de croyances, parfois vielles de plusieurs générations et qui perdurent souvent par manque d'ouverture d'esprit et surtout parce que la bonne attitude à adopter n'est pas enseignée dans les écoles supérieures là où les dogmes s'établissent et pénètrent les esprits dociles, faute au biais d'ancrage mental. L'enseignant devrait être sensible à ces aspects fondamentaux et devrait viser l'impartialité en visant l'ouverture l'esprit et le culte du bon sens de l'ingénieur.

Citons par exemple les guerres d'éditeurs qui date des années 1970 et qui opposent les

défenseurs de l'éditeur `vi` aux inconditionnels d'`emacs`. Il s'agit de deux éditeurs de texte très puissants et à la courbe d'apprentissage raide qui séparent les opinions tant leur paradigme de fonctionnement est aporétique. Ces guerres sont d'abord entretenues par plaisir de l'amusement, mais les foules de convertis ne s'aperçoivent pas toujours de l'envergure émotionnelle que prend l'affaire dans son ensemble et force est de constater qu'avec le temps ils ne parviennent plus à percevoir le monde tel qu'il est, à force d'habitudes.

S'enterrer dans une zone de confort renforce le biais du Marteau de Maslow, car lorsque l'on est un marteau, on ne voit plus les problèmes qu'en forme de clou. Cette zone de confort devient un ennemi et barre l'accès au regard critique et au pragmatisme qui devrait prévaloir. Car accepter l'existence de différentes approches possibles d'un problème donné est, essentiel, car plus que dans tout autre domaine technique, le développement logiciel est avant tout une aventure collaborative qui ne devrait jamais être sous le joug d'une quelconque emprise émotionnelle.

Un programme se doit d'être le plus neutre possible, impartial et minimaliste. Il n'est pas important de se préoccuper des affaires cosmétiques telles que la position des accolades dans un programme, le choix d'utiliser des espaces versus des tabulations horizontales, ou le besoin d'utiliser tel ou tel outil de développement parce qu'il est jugé meilleur qu'un autre.

La clé de la bonne attitude c'est d'être à l'écoute du consensus de ne pas sombrer au biais d'attention. Il faut non seulement être sensible au consensus local direct : son entreprise, son école, son équipe de travail, mais surtout au consensus planétaire dont l'accès ne peut se faire que par l'interaction directe avec la communauté de développeurs, soit par les forums de discussions (reddit, stackoverflow), soit par le code lui-même. Vous avez un doute sur la bonne méthode pour écrire tel algorithme ou sur la façon dont votre programme devrait être structuré ? Plongez-vous dans le code des autres, multipliez vos expériences, observez les disparités et les oppositions, et apprenez à ne pas y être sensible.

Vous verrez qu'au début, un programme ne vous semble lisible que s'il respecte vos habitudes, la taille de vos indentations préférées, la police de caractère qui vous sied le mieux, l'éditeur qui supporte les ligatures, car admettez-le `fi` est infiniment plus lisible que `fi`. Par la suite, et à la relecture de cette section, vous apprendrez à faire fi de cette zone de confort qui vous était si chère et que l'important n'est plus la forme, mais le fond. Vous aurez comme Néo, libéré votre esprit et serez capable de voir la matrice sans filtre, sans biais.

En somme, restez ouvert aux autres points de vues, cherchez à adopter le consensus majoritaire qui dynamise au mieux votre équipe de développement, qui s'encadre le mieux dans votre stratégie de croissance et de collaboration et surtout, abreuvez-vous de code, faites-en des indigestions, rêvez-en la nuit. Vous tradez du Bitcoin, allez lire le code source, vous aimez Linux, plongez-vous dans le code source du kernel, certains collègues ou amis vous ont parlé de Git, allez voir ses entrailles... Oui, tous ces projets sont écrits en C, n'est-ce pas merveilleux ?

1.11 Hello World !

Il est traditionnellement coutume depuis la publication en 1978 du livre The C Programming Language de reprendre l'exemple de Brian Kernighan comme premier programme.

```c
#include <stdio.h>

int main(void)
{
    printf("hello, world\n");
    return 0;
}
```

Ce programme est composé de deux parties. L'inclusion de la *library* standard d'entrées sorties (*STandarD Inputs Outputs*) qui définit la fonction `printf`, et le programme principal nommé `main`. Tout ce qui se situe à l'intérieur des accolades `{ }` appartient au programme `main`.

L'ensemble que définit `main` et ses accolades est appelé une fonction, et la tâche de cette fonction est ici d'appeler une autre fonction *printf* dont le nom vient de *print formatted*.

L'appel de `printf` prend en **paramètre** le texte `Hello world!\n` dont le `\n` représente un retour à la ligne.

Une fois le code écrit, il faut le compiler. Pour bien comprendre ce que l'on fait, utilisons la ligne de commande ; plus tard, l'IDE se chargera de l'opération automatiquement.

Une console lancée ressemble à ceci, c'est intimidant si l'on en a pas l'habitude mais vraiment puissant.

```
$
```

La première étape est de s'assurer que le fichier `test.c` contient bien notre programme. Pour ce faire on utilise un autre programme cat qui ne fait rien d'autre que lire le fichier passé en argument et de l'afficher sur la console :

```
$ cat hello.c
#include <stdio.h>

int main(void)
{
    printf("hello, world\n");
    return 0;
}
```

A présent on peut utiliser notre compilateur par défaut : `cc` pour *C Compiler*. Ce compilateur prends en argument un fichier C et sans autre option, il génèrera un fichier a.out pour *assembler output*. C'est un fichier exécutable que l'on peut donc exécuter.

```
$ gcc hello.c
```

Il ne s'est rien passé, c'est une bonne nouvelle. La philosophie Unix est qu'un programme soit le plus discret possible, comme tout s'est bien passé, inutile d'informer l'utilisateur.

On s'attends donc à trouver dans le répertoire courant, notre fichier source ainsi que le résultat de la compilation. Utilisons le programme ls pour le vérifier

```
$ ls
hello.c       a.out
```

Très bien ! A présent, exécutons le programme en prenant soin de préfixer le nom par `./` car étant un programme local `a.out` ne peut pas être accédé directement. Imaginons qu'un fourbe hacker ait décidé de créer dans ce répertoire un programme nommé `ls` qui efface toutes vos données. La ligne de commande ci-dessus aurait eu un effet désastreux. Pour remédier à ce problème de sécurité tout programme local doit être explicitement nommé.

```
$ ./a.out
hello, world
```

Félicitations, le programme s'est exécuté.

Pouvons nous en savoir plus sur ce programme ? On pourrait s'intéresser à la date de création de ce programme ainsi qu'à sa taille sur le disque. Une fois de plus `ls` nous sera utile mais cette fois-ci avec l'option `l` :

```
$ ls -l a.out
-rwxr-xr-- 1 ycr iai 8.2K Jul 24 09:50 a.out*
```

Décortiquons tout cela :

```
-             Il s'agit d'un fichier
rwx           Lisible (r), Éditable (w) et Exécutable (x) par le␣
↪propriétaire
r-x           Lisible (r) et Exécutable (x) par le groupe
r--           Lisible (r) par les autres utilisateurs
1             Nombre de liens matériels pour ce fichier
ycr           Nom du propriétaire
iai           Nom du groupe
8.2K          Taille du fichier, soit 8200 bytes soit 65'600 bits
Jul 24 09:50  Date de création du fichier
a.out         Nom du fichier
```

Exercice

Qui a inventé le C ?

Brian Kernighan et Dennis Ritchie en 1972 Exercice

Quel est le standard C à utiliser en 2019 et pourquoi ?

Le standard industriel, malgré que nous soyons en 2019 est toujours **ISO/IEC 9899 :1999** car peu de changements majeurs ont été apporté au langage depuis et les

entreprises préfèrent migrer sur C++ plutôt que d'adopter un standard plus récent qui n'apporte que peu de changements. Exercice

Quels sont les paradigmes de programmation supportés par C ?

C supporte les paradigmes impératifs, structurés et procédural. Exercice

Pourriez-vous définir ce qu'est la programmation impérative ?

La programmation impérative consiste en des séquences de commandes ordonnées. C'est à dire que les séquences sont exécutées dans un ordre défini les unes à la suite de autres. Exercice

Qu'est ce qu'une coulée de lave en informatique ?

Lorsqu'un code immature est mis en production, l'industriel qui le publie risque un retour de flamme dû aux bogues et mécontentement des clients. Afin d'éviter une *coulée de lave* il est important qu'un programme soit testé et soumis à une équipe de *beta-testing* qui s'assure qu'outre le respect des spécifications initiales, le programme soit utilisable facilement par le public cible. Il s'agit aussi d'étudier l'ergonomie du programme.

Un programme peut respecter le cahier des charges, être convenablement testé, fonctionner parfaitement mais être difficile à l'utilisation car certaines fonctionnalités sont peu ou pas documentées. La surcharge du service de support par des clients perdus peut également être assimilée à une coulée de lave. Exercice

Qu'est-ce que `cat` ?

`cat` est un programme normalisé POSIX prenant en entrée un fichier et l'affichant à l'écran. Il est utilisé notament dans cet ouvrage pour montrer que le contenu du fichier `hello.c` est bel et bien celui attendu.

Chapitre 2

La programmation

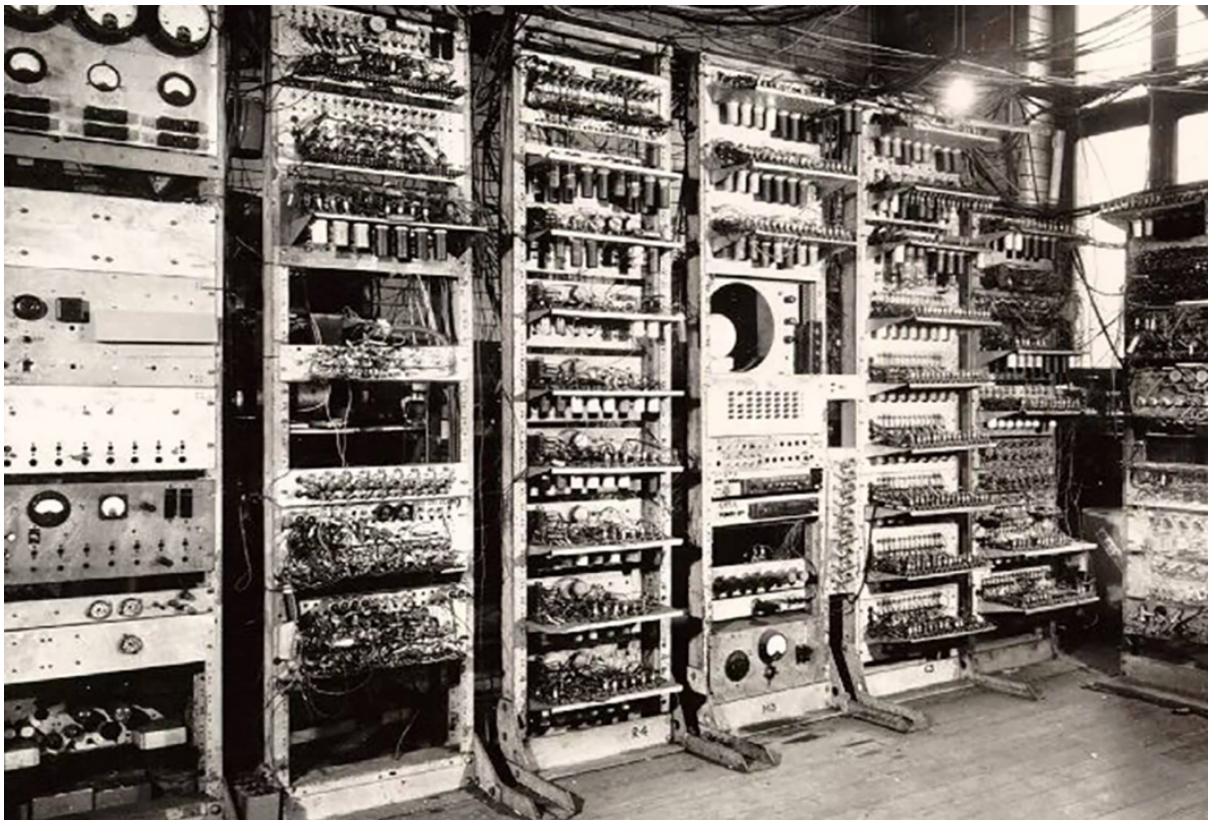


Fig. 2.1 – L'un des premiers ordinateurs : l'Eniac

Il ne serait pas raisonnable de vous enseigner la programmation C sans au préalable définir ce qu'est la programmation et quelle est son origine. La programmation intervient après une étape plus générale impliquant un ou plusieurs algorithmes.

Algorithmique et Programmation, il y donc deux questions à éclairer :

- Qu'est-ce que l'algorithmique ?
- Qu'est-ce que la programmation ?

2.1 Algorithmique

L'algorithmique et non l'*algorithmie*, est la science qui étudie la production de règles et techniques impliquées dans la définiton et la conception d'algorithmes. Nous verrons l'algorithmique plus en détail dans le chapitre Section 17. Retenons pour l'heure que l'algorithmique intervient tous les jours :

- dans une recette de cuisine,
- le tissage de tapis persans,
- les casse-tête (Rubik's Cube),
- les tactiques sportives,
- les procédures administratives.

Dans le contexte mathématique et scientifique qui nous intéresse ici, citons l'algorithme d'Euclide datant probablement de 300 av. J.-C. est un algorithme permettant de déterminer le plus grand commun diviseur (PGCD). Voici la description de l'algorithme :

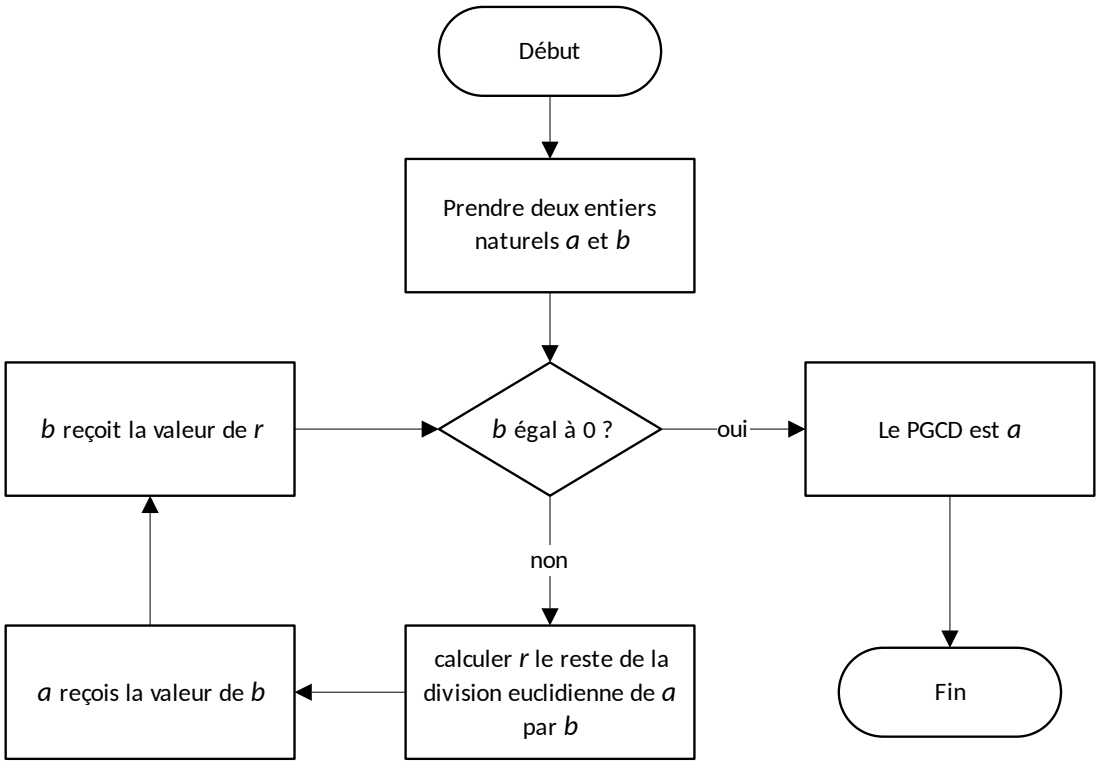


Fig. 2.2 – Algorithme de calcul du PGCD d'Euclide.

Exercice

Appliquer l'algorithme d'Euclide aux entrées suivantes. Que vaut a , b et r ?

$$a = 1260, b = 630$$

2.2 Programmation

La machine Jacquard est un métier à tisser mis au point par Joseph Marie Jacquard en 1801. Il constitue le premier système mécanique programmable avec cartes perforées.

Les cartes perforées contiennent donc la suite des actions guidant les crochets permettant de tisser des motifs complexes. L'automatisation d'un travail qui jadis était effectué manuellement causa une vague de chômage menant à la Révolte des canuts en 1831.

La programmation définit toute activité menant à l'écriture de programmes. En informatique, un programme est un ensemble ordonné d'instructions codées avec un langage donné et décrivant les étapes menant à la solution d'un problème. Il s'agit le plus souvent d'une écriture formelle d'un algorithme.

Les informaticiens-tisserands responsables de la création des cartes perforées auraient pu se poser la question de comment simplifier leur travail en créant un langage formel pour créer des motifs complexes et dont les composants de base se répètent d'un travail à l'autre. Prenons l'exemple d'un ouvrier spécialisé en héraldique et devant créer des motifs complexes de blasons. Nul n'est sans savoir que l'héraldique à son langage parfois obscure et celui qui le maitrise voudrait par exemple l'utiliser au lieu de manuellement percer les cartes pour chaque point de couture. Ainsi l'anachronique informaticien-tisserand souhaitant tisser le motif des armoiries duc de Mayenne (c.f. figure ci-dessous) aurait sans doute rédigé un programme informatique en utilisant sa langue. Le programme aurait pu ressembler à ceci :

```
Écartelé, en 1 et 4 :
    coupé et parti en 3,
        au premier fascé de gueules et d'argent,
        au deuxième d'azur semé de lys d'or
            et au lambel de gueules,
        au troisième d'argent à la croix potencée d'or,
            cantonnée de quatre croisettes du même,
        au quatrième d'or aux quatre pals de gueules,
        au cinquième d'azur semé de lys d'or
            et à la bordure de gueules,
        au sixième d'azur au lion contourné d'or,
            armé,
            lampassé et couronné de gueules,
        au septième d'or au lion de sable,
            armé,
            lampassé de gueules,
        au huitième d'azur semé de croisettes d'or
            et aux deux bar d'or.
    Sur le tout d'or à la bande de gueules
        chargé de trois alérions d'argent
    le tout brisé d'un lambel de gueules ;
        en 2 et 3 contre-écartelé en 1 et 4 d'azur,
        à l'aigle d'argent, becquée,
        languée et couronnée d'or et en 2 et 3 d'azur,
        à trois fleurs de lys d'or,
```

(suite sur la page suivante)

Fig. 2.3 – Mécanisme Jacquard au Musée des arts et métiers de Paris.

(suite de la page précédente)

```
        à la bordure endentée de gueules et d'or.
```

Notons que *de gueules* signifie *rouge*. Le drapeau suisse est donc *de gueules, à la croix alésée d'argent*.

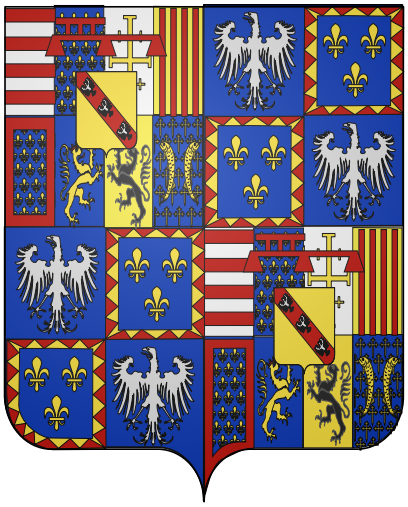


Fig. 2.4 – Armoiries des ducs de Mayenne

2.3 Calculateur

Un calculateur du latin *calculare* : calculer avec des cailloux, originellement appelé abaque était un dispositif permettant de faciliter les calculs mathématiques.

Les os d'Ishango datés de 20'000 ans sont des artéfacts archéologiques attestant la pratique de l'arithmétique dans l'histoire de l'humanité.

Si les anglophones ont détourné le verbe *compute* (calculer) en un nom *computer*, un ordinateur est généralement plus qu'un simple calculateur car même une calculatrice de poche doit gérer en plus des calculs :

— l'interface de saisie (pavé numérique) ;
— l'affichage du résultat (écran à cristaux liquide).

2.4 Ordinateur

Le terme ordinateur est très récent, il daterait de 1955, créé par Jacques Perret à la demande d'IBM France (c.f 2014 : 100 ans d'IBM en France).

> « Le 16 IV 1955 Cher Monsieur, Que diriez-vous d'ordinateur ? C'est un mot correctement formé, qui se trouve même dans le Littré comme adjectif désignant Dieu qui met de l'ordre dans le monde. Un mot de ce genre a l'avantage de donner aisément un verbe ordiner, un nom d'action ordination. L'inconvénient est que ordination désigne une cérémonie religieuse ; mais les deux champs de signification (religion et comptabilité) sont si éloignés et la

cérémonie d'ordination connue, je crois, de si peu de personnes que l'inconvénient est peut-être mineur. D'ailleurs votre machine serait ordinateur (et non-ordination) et ce mot est tout à fait sorti de l'usage théologique. Systématateur serait un néologisme, mais qui ne me paraît pas offensant ; il permet systématisé ; — mais système ne me semble guère utilisable — Combinateur a l'inconvénient du sens péjoratif de combine ; combiner est usuel donc peu capable de devenir technique ; combination ne me paraît guère viable à cause de la proximité de combinaison. Mais les Allemands ont bien leurs combinats (sorte de trusts, je crois), si bien que le mot aurait peut-être des possibilités autres que celles qu'évoque combine.

Congesteur, digesteur évoquent trop congestion et digestion. Synthétiseur ne me paraît pas un mot assez neuf pour désigner un objet spécifique, déterminé comme votre machine.

En relisant les brochures que vous m'avez données, je vois que plusieurs de vos appareils sont désignés par des noms d'agent féminins (trieuse, tabulatrice). Ordinatrice serait parfaitement possible et aurait même l'avantage de séparer plus encore votre machine du vocabulaire de la théologie. Il y a possibilité aussi d'ajouter à un nom d'agent un complément : ordinatrice d'éléments complexes ou un élément de composition, par exemple : sélecto-systématateur. Sélecto-ordinateur a l'inconvénient de deux o en hiatus, comme électro-ordonnatrice.

Il me semble que je pencherais pour ordonnatrice électronique. Je souhaite que ces suggestions stimulent, orientent vos propres facultés d'invention. N'hésitez pas à me donner un coup de téléphone si vous avez une idée qui vous paraisse requérir l'avis d'un philologue.

Vôtre Jacques Perret »

2.5 Historique

87 av. J.-C. La machine d'Anticythère considéré comme le premier calculateur analogique pour positions astronomiques.

1642 La pascaline : machine d'arithmétique de Blaise Pascal, première machine à calculer

1834 Machine à calculer programmable de Charles Babbage

1937 l'Automatic Sequence Controlled Calculator Mark I d'IBM, le premier grand calculateur numérique.
- 4500 kg
- 6 secondes par multiplication à 23 chiffres décimaux
- Cartes perforées

1950 L'ENIAC, de Presper Eckert et John William Mauchly
- 160 kW
- 100 kHz

— Tubes à vide
— 100'000 additions/seconde
— 357 multiplications/seconde

1965 Premier ordinateur à circuits intégrés, le PDP-8
— 12 bits
— mémoire de 4096 mots
— Temps de cycle de 1.5 µs
— Fortran et BASIC

2018 Le Behold Summit est un superordinateur construit par IBM.
— 200'000'000'000'000'000 multiplications par seconde
— simple ou double précision
— 14.668 GFlops/watt
— 600 GiB de mémoire RAM

2.6 Fonctionnement de l'ordinateur

2.6.1 Machine de Turing

Exercice

Comment est mort Alain Turing et pourquoi est-il connu ?

Chapitre 3

Généralités du langage

Ce chapitre traite des éléments constitufs et fondamentaux du langage C. Il traite des généralités propres au langage mais aussi des notions élémentaires permettant d'interpréter du code source.

Notons que ce chapitre est transveral, à la sa première lecture, le profane ne pourra tout comprendre sans savoir lu et maitrisé les chapitres suivants, néanmoins il retrouvera ici les aspects fondamentaux du langage.

3.1 L'alphabet

Fort heureusement pour nous occidentaux, l'alphabet de C est composé des 52 caractères latins et de 10 chiffres indo-arabes :

```
A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
a b c d e f g h i j k l m n o p q r s t u v w x y z
0 1 2 3 4 5 6 7 8 9
```

La séparation des symboles est assurée par une espace, une tabulation horizontale, une tabulation verticale, et un caractère de retour à la ligne. Ces caractères ne sont pas imprimables, c'est à dire qu'ils ne sont pas directement visible ni à l'écran, ni à l'impression. Microsoft Word et d'autres éditeurs utilisent généralement le pied-de-mouche ¶ pour indiquer les fin de paragraphes qui sont également des caractères non-imprimables.

On nomme les caractères non-imprimables soit par leur acronyme `LF` pour *Line Feed* ou soit par leur convention C échappée par un *backslash* `\n` :

```
LF    \n   Line feed (retour à la ligne)
VT    \v   Vertical tab
FF    \f   New page
TAB   \t   Horizontal tab
CR    \r   Carriage return (retour charriot)
SPACE \040 Space
```

La ponctuation utilise les 29 symboles graphiques suivants :

```
! # % ^ & * ( _ ) - + = ~ [ ] ' | \ ; : " { } , . < > / ?
```

Un fait historique intéressant est que les premiers ordinateurs ne disposaient pas d'un clavier ayant tous ces symboles et la comission responsable de standardiser C a intégré au standard les **trigraphes** et plus tard les **digraphes** qui sont des combinaisons de caractères de base qui remplacent les caractères impossibles à saisir directement. Ainsi `<:` est le digraphe de `[` et `??<` est le trigraphe de `{`. Néanmoins vous conviendrez cher lecteur que ces alternatives ne devraient être utilisées que dans des cas extrêmes et justifiables.

Retenez que C peut être un langage extrèmement cryptique tant il est permissif sur sa syntaxe. Il existe d'ailleurs un concours international d'obfuscation, le The International Obfuscated C Code Contest qui prime des codes les plus subtils et illisibles comme le code suivant écrit par Chris Mills. Il s'agit d'ailleurs d'un exemple qui compile parfaitement sur la plupart des compilateurs.

```
    int I=256,l,c, o,O=3; void e(
int L){ o=0; for( l=8; L>>++l&&
16>l;                           o+=l
<<l-                            1) ;
o+=l                        *L-(l<<l-1); { ; }
if (                       pread(3,&L,3,O+o/8)<
2)/*                       */exit(0);  L>>=7&o;
L%=1                        <<l; L>>8?256-L?e(
L-1)                               ,c||
(e(c                               =L),
c=0)                               :( O
+=(-I&7)*l+o+l>>3,I=L):putchar(
    L); }int main(int l,char**o){
                for(
            /*          ////      */
            open(1[o],0); ; e(I++
            ))                     ;}
```

3.2 Fin de lignes (EOL)

À l'instar des premières machines à écrire, les téléscripteurs possédaient de nombreux caractères de déplacement qui sont depuis tombés en désuétude et prêtent aujourd'hui à confusion même pour le plus aguerri des programmeurs. Maintenant que les ordinateurs possèdent des écrans, la notion originale du terme retour chariot est compromise et comme il y a autant d'avis que d'ingénieurs, les premiers PC IBM compatibles ont choisi qu'une nouvelle ligne devait toujours se composer de deux caractères : un retour chariot (`CR`) et une nouvelle ligne (`LF`) ou en C `\r\n`. Les premiers Macintosh d'Apple jugaient inutile de gaspiller deux caractères pour chaque nouvelle ligne dans un fichier et ont décidé d'associer le retour chariot et la nouvelle ligne dans le caractère `\r`. Enfin, les ordinateurs UNIX ont eu le même raisonnement mais ils ont choisi de ne garder que `\n`.

Fort heureusement depuis que Apple a migré son système sur une base BSD (UNIX), il

n'existe aujourd'hui plus que deux standards de retour à la ligne :

— `LF` ou `\n` sur les ordinateurs POSIX comme Linux, Unix ou MacOS

— `CRLF` ou `\r\n` sur les ordinateurs Windows.

Il n'y a pas de consensus établi sur lesquels des deux types de fin de ligne (`EOL` : *End Of Line*) il faut utiliser, faite preuve de bon sens et surtout, soyez cohérent.

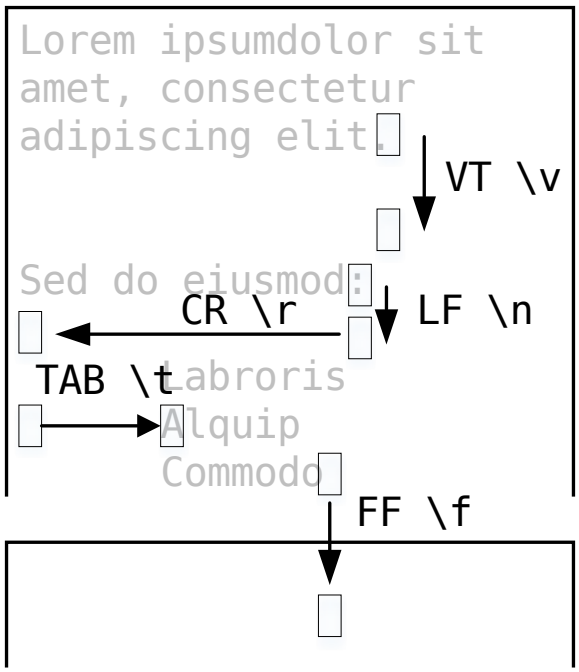


Fig. 3.1 – Distinction de différents caractères non-imprimables

3.3 Mots clés

Le langage de programmation C tel que défini par C11 comporte environ 37 mots clés :

```
auto        do          goto        return      typedef     _
 ↪Complex
break       double      if          short       union       _
 ↪Imaginary
case        else        inline      signed      unsigned
char        enum        int         sizeof      void
const       extern      long        static      volatile
continue    float       register    struct      while
default     for         restrict    switch      _Bool
```

Dans ce cours l'usage des mots clés suivants est découragé car leur utilisation pourrait prêter à confusion ou mener à des inélégances d'écriture.

```
auto        restrict    short       inline
_Bool       register    goto        _imaginary
long
```

Notons que les mots clés `true` et `false` décrits à la Section 6.8 ne sont pas standardisés en C mais ils le sont en C++.

3.4 Identificateurs

Un identificateur est une séquence de caractères représentant une entité du programme et à laquelle il est possible de se référer. Un identificateur est défini par :

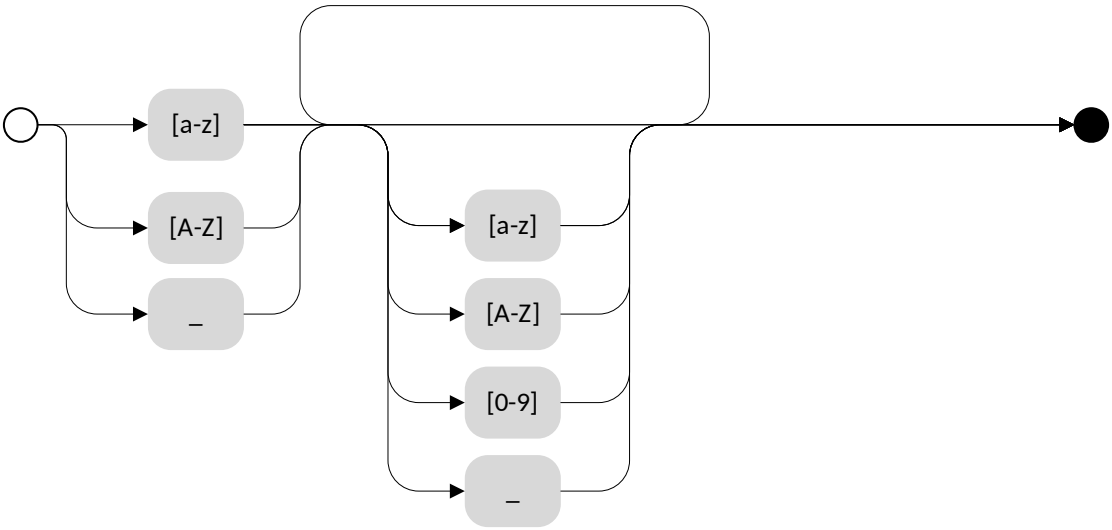


Fig. 3.2 – Grammaire d'un identificateur C

En addition de ceci, voici quelques règles :

- Un identificateur ne peut pas être l'un des mots clés du langage.
- Les identificateurs sont sensible à la casse.
- Le standard C99, se réserve l'usage de tous les identificateurs débutant par `_` suivi d'une lettre majuscule ou un autre *underscore* `_`.
- Le standard POSIX, se réserve l'usage de tous les identificateurs finissant par `_t`.

Indication : Expression régulière

Il est possible d'exprimer la syntaxe d'un identificateur à l'aide de l'expression régulière suivante :

```
^[a-zA-Z_][a-zA-Z0-9_]*$
```

Exercice

Pour chacune des suites de caractères ci-dessous, indiquez s'il s'agit d'un identificateur valide et utilisable en C. Justifier votre réponse.

1. `2_pi`
2. `x_2`
3. `x___3`
4. `x 2`
5. `positionRobot`
6. `piece_presente`
7. `_commande_vanne`
8. `-courant_sortie`
9. `_alarme_`
10. `panne#2`
11. `int`
12. `défaillance`
13. `f'`
14. `INT`

Une excellente approche serait d'utiliser directement l'expression régulière fournie et d'utiliser l'outil en ligne regex101.com.

1. `2_pi` **invalide** car commence par un chiffre
2. `x_2` **valide**
3. `x___3` **valide**
4. `x 2` **invalide** car comporte un espace
5. `positionRobot` **valide**, notation *camelCase*
6. `piece_presente` **valide**, notation *snake_case*
7. `_commande_vanne` **valide**
8. `-courant_sortie` **invalide**, un identificateur ne peut pas commencer par le signe `-`
9. `_alarme_` **valide**
10. `panne#2` **invalide**, le caractère `#` n'est pas autorisé
11. `int` **invalide**, `int` est un mot réservé du langage
12. `défaillance` **invalide**, uniquement les caractères imprimable ASCII sont autorisés
13. `f'` **invalide** l'apostrophe n'est pas autorisée
14. `INT` **valide**

3.5 Variables

Une variable est un symbole qui associe un nom **identificateur** à une **valeur**. Comme son nom l'indique, une variable peut voir son contenu varier au cours du temps.

Une variable est définie par :

— Son **nom** (*name*), c'est à dire l'identificateur associé au symbole.
— Son **type** (*type*), qui est la convention d'interprétation du contenu binaire en mémoire.
— Sa **valeur** (*value*), qui est le contenu interprêté connaissant son type.
— Son **adresse** (*address*) qui est l'emplacement mémoire ou la représentation binaire sera enregistrée
— Sa **portée** (*scope*) qui est la portion de code ou le symbole est défini et accessible.
— Sa **visibilité** (*visibility*) qui ne peut être que *public* en C.

3.5.1 Déclaration

Avant de pouvoir être utilisée, une variable doit être déclarée afin que le compilateur puisse réserver un emplacement en mémoire pour stocker sa valeur. Voici quelques déclarations valides en C :

```
char c = '€';
int temperature = 37;
float neptune_stone_height = 376.86;
char message[] = "Jarvis, il faut parfois savoir "
    "courir avant de savoir marcher.";
```

Il n'est pas nécessaire d'associer une valeur initiale à une variable, une déclaration peut se faire sans initialisation comme montré dans l'exemple suivant dans lequel on réserver trois variables `i`, `j`, `k`.

```
int i, j, k;
```

Exercice

Considérons les déclarations suivantes :

```
int a, b, c;
float x;
```

Notez après chaque affectation, le contenu des différentes variables :

Ligne	Instruction	a	b	c	x
1	`a = 5;`				
2	`b = c;`				
3	`c = a;`				
4	`a = a + 1;`				
5	`x = a - ++c;`				
6	`b = c = x;`				
7	`x + 2. = 7.;`				

Ligne	Instruction	a	b	c	x
1	`a = 5;`	5	?	?	?
2	`b = c;`	5	?	?	?
3	`c = a;`	5	?	5	?
4	`a = a + 1;`	6	?	5	?
5	`x = a - ++c;`	6	?	6	12
6	`b = c = x;`	6	12	12	12
7	`x + 2. = 7.;`	—	—	—	—

3.5.2 Convention de nommage

Il existe autant de conventions de nommage qu'il y a de développeurs mais un consensus majoritaire, que l'on retrouve dans d'autres langages de programmation exprime que :

- La longueur du nom d'une variable est généralement proportionnelle à sa portée et donc il est d'autant plus court que l'utilisation d'une variable est localisée.
- Le nom doit être concis et précis et ne pas laisser place à une quelconque ambiguité.
- Le nom doit participer à l'auto-documentation du code et permettre à un lecteur de comprendre facilement le programme qu'il lit.

Selon les standards adoptés chaque société on trouve ceux qui préfèrent nommer les variables en utilisant un *underscore* (`_`) comme séparateur et ceux qui préfèrent nommer une variable en utilisant des majuscules comme séparateurs de mots.

Convention	Nom français	Exemple
camelcase	Casse de chameau	`userLoginCount`
snakecase	Casse de serpent	`user_login_count`
pascalcase	Casse de Pascal	`UserLoginCount`
kebabcase	Casse de kebab	`user-login-count`

3.5.3 Variable métasyntaxiques

Souvent lors d'exemples donnés en programmation on utilise des variables génériques dites métasyntaxiques. En français les valeurs `toto`, `titi`, `tata` et `tutu` sont régulièrement utilisées tandis qu'en anglais `foo`, `bar`, `baz` et `qux` sont régulièrement utilisés. Les valeurs `spam`, `ham` et `eggs` sont quant à elles souvent utilisée en Python, en référence au sketch Spam des Monthy Python.

Leur usage est conseillé pour appuyer le cadre générique d'un exemple sans lui donner la consonnance d'un problème plus spécifique.

On touvera une table des différents noms les plus courants utilisés dans différentes langues.

3.6 Les constantes

Une constante par opposition à une variable voit son contenu fixe et immutable.

Formellement, une constante se déclare comme une variable mais préfixée du mot-clé `const`.

```
const double scale_factor = 12.67;
```

Note : Il ne faut pas confondre la **constante** qui est une variable immutable, stockée en mémoire et une **macro** qui appartient au pré-processeur. Le fichier d'en-tête `math.h` définit par exemple la constante `M_PI` sous forme de macro.

```
#define M_PI 3.14159265358979323846
```

3.7 Constantes littérales

Les constantes littérales représentent des grandeurs scalaires numériques ou de caractères et initialisées lors de la phase de compilation.

```
6      // Grandeur valant le nombre d'heures sur l'horloge du␣
↪Palais du Quirinal à Rome
12u    // Grandeur non signée
6l     // Grandeur entière signée codée sur un entier long
42ul   // Grandeur entière non signée codée sur un entier long
010    // Grandeur octale valant 8 en décimal
0xa    // Grandeur hexadécimale valant 10 en décimal
0b111  // Grandeur binaire valant 7 en décimal
33.    // Grandeur réelle exprimée en virgule flottante
'0'    // Grandeur caractère vallant 48 en décimal
```

Exercice

Pour les entrées suivantes, indiquez lesquelles sont correctes.

1. `12.3`
2. `12E03`
3. `12u`
4. `12.0u`
5. `1L`
6. `1.0L`
7. `.9`
8. `9.`
9. `.`
10. `0x33`
11. `0xefg`
12. `0xef`
13. `0xeF`
14. `0x0.2`
15. `09`
16. `02`

3.8 Operateur d'affectation

Dans les exemples ci-dessus on utilise l'opérateur d'affectation pour associer une valeur à une variable.

Historiquement, et fort malheureusement, le symbole choisi pour cet opérateur est le signe égal `=` or, l'égalité est une notion mathématique qui n'est en aucun cas reliée à l'affectation.

Pour mieux saisir la nuance, considérons le programme suivant :

```
a = 42;
a = b;
```

Mathématiquement, la valeur de `b` devrait être égale à 42 ce qui n'est pas le cas en C où il faut lire, séquentiellement l'exécution du code car oui, C est un langage impératif (c.f. Section 1.7). Ainsi, dans l'ordre on lit :

1. J'assigne la valeur 42 à la variable symbolisée par `a`
2. Puis, j'assigne la valeur de la variable `b` au contenu de `a`.

Comme on ne connaît pas la valeur de `b`, avec cet exemple, on ne peut pas connaître la valeur de `a`.

Certaines langages de programmation ont été sensibilisé à l'importance de cette distinction et dans les langages **F#**, **OCaml**, **R** ou **S**, l'opérateur d'affectation est `<-` et une affectation pourrait s'écrire par exemple : `a <- 42` ou `42 -> a`.

En C, l'opérateur d'égalité que nous verrons plus loin s'écrit `==` (deux `=` concaténés).

Remarquez ici que l'opérateur d'affectation de C agit toujours de droite à gauche c'est à dire que la valeur à **droite** de l'opérateur est affectée à la variable située à **gauche** de l'opérateur.

S'agissant d'un opérateur il est possible de chaîner les opérations, comme on le ferait avec l'opérateur `+` et dans l'exemple suivant il faut lire que `42` est assigné à `c`, que la valeur de `c` est ensuite assignée à `b` et enfin la valeur de `b` est assignée à `a`.

```
a = b = c = 42;
```

Nous verrons Section 5.14 que l'associativité de chaque opérateur détermine s'il agit de gauche à droite ou de droite à gauche. Exercice

Donnez les valeurs de `x`, `n`, `p` après l'exécution des instructions ci-dessous :

```
float x;
int n, p;

p = 2;
x = 15 / p;
n = x + 0.5;
```

```
p ≡ 2
x ≡ 7
n ≡ 7
```

Exercice

On considère les déclarations suivantes :

```
int i, j, k;
```

Donnez les valeurs des variabels `i`, `j` et `k` après l'exécution de chacune des expressions ci-dessous. Qu'en pensez-vous ?

```
/* 1 */ i = (k = 2) + (j = 3);
/* 2 */ i = (k = 2) + (j = 2) + j * 3 + k * 4;
/* 3 */ i = (i = 3) + (k = 2) + (j = i + 1) + (k = j + 2) + (j = k -
 ↪ 1);
```

Selon la table de priorité des opérateurs, on note :

- `()` priorité 1 associativité à droite
- `*` priorité 3 associativité à gauche
- `+` priorité 4 associativité à droite
- `=` priorité 14 associativité à gauche

En revanche rien n'est dit sur les point de séquences. L'opérateur d'affectation n'est pas un point de séquence, autrement dit le standard C99 (Annexe C) ne définit pas l'ordre dans lequel les assignations sont effectuées.

Ainsi, seul le premier point possède une solution, les deux autres sont indéterminés

1. `i = (k = 2) + (j = 3)`
   - `i = 5`
   - `j = 3`
   - `k = 2`
2. `i = (k = 2) + (j = 2) + j * 3 + k * 4`
   - Résultat indéterminé
3. `i = (i = 3) + (k = 2) + (j = i + 1) + (k = j + 2) + (j = k - 1)`

   - Résultat indéterminé

3.9 Commentaires

Comme en français et ainsi qu'illustré par la Fig. 3.3, il est possible d'annoter un programme avec des **commentaires**. Les commentaires n'ont pas d'incidence sur le fonctionnement d'un programme et ne peuvent être lu que par le développeur qui possède le code source.

Je m'amusais à regarder les carafes que les gamins mettaient dans la Vivonne pour prendre les petits poissons, et qui, remplies par la rivière, où elles sont à leur tour encloses, à la fois "contenant" aux flancs transparents comme une eau durcie, et "contenu" plongé dans un plus grand contenant de cristal liquide et courant, évoquaient l'image de la fraîcheur d'une façon plus délicieuse et plus irritante qu'elles n'eussent fait sur une table servie, en ne la montrant qu'en fuite dans cette allitération perpétuelle entre l'eau sans consistance où les mains ne pouvaient la capter et le verre sans fluidité où le palais ne pourrait en jouir.

- Marcl Proust

Fig. 3.3 – Les carafes dans la Vivonne

Il existe deux manière d'écrire un commentaire en C :

- Les commentaires de lignes (depuis C99)

```
// This is a single line comment.
```

- Les commentaires de block

```
/* This is a
   Multi-line comment */
```

Les commentaires sont parsés par le pré-processeur, aussi ils n'influencent pas le fonctionnement d'un programme mais seulement sa lecture. Rappelons qu'un code est plus

souvent lu qu'écrit, car on ne l'écrit qu'une seule fois mais comme tout développement doit être si possible **réutilisable**, il est plus probable qu'il soit lu part d'autres développeurs.

En conséquence, il est important de clarifier toute zone d'ombre lorsque que l'on s'éloigne des consensus établis, ou lorsque le code seul n'est pas suffisant pour bien comprendre son fonctionnement.

D'une façon générale, les commentaires servent à expliquer **pourquoi** et non **comment**. Un bon programme devrait pouvoir se passer de commentaires mais un programme sans commentaires n'est pas nécessairement un bon programme.

Notons que l'on ne commente jamais des portions de code et ce pour plusieurs raisons :

1. Les outils de *refactoring* ne pourront pas accéder du code commenté
2. La syntaxe ne pourra plus être vérifiée par l'IDE
3. Les outils de gestion de configuration (e.g. Git) devraient être utilisés à cette fin

Si d'aventure vous souhaitez exclure temporairement du code de la compilation, utilisez la directive de pré-processeur suivante, et n'oubliez pas d'expliquer pourquoi vous avez souhaité désactiver cette portion de code.

```
#if 0 // TODO: Check if divisor could still be null at this point.
if (divisor == 0) {
    return -1; // Error
}
#endif
```

D'une manière générale l'utilisaton des commentaires ne devrait pas être utilisée pour :

- Désactiver temporairement une portion de code sans l'effacer.
- Expliquer le **comment** du fonctionnement du code.
- Faire dans le dythyrambique pompeux et notarial, des phrases à rallonge bien trop romanesques.
- Créer de jolies séparations telles que `/**************************/`.

Exemple d'entête de fichier :

```
/**
 * Short description of the translation unit.
 *
 * Author: John Doe <john@doe.com>
 *
 * Long description of the translation unit.
 *
 * NOTE: Important notes about this code
 */
```

Le format des commentaires est par essence libre au développeur mais il est généralement souhaité que :

- Les commentaires soient concis et précis.
- Les commentaires soient écrits en anglais.

Exercice

Comment récririez-vous ce programme ?

```
for (register unsigned int the_element_index = 0; the_element_index
↪< number_of_elements; the_element_index += 1)
    array_of_elements[the_element_index] =  the_element_index;
```

Une règle de programmation : le nom identifieurs doit être proportionnel à leur contexte. Plus le contexte de la variable est réduit, plus le nom peut être court. Le même programme pourrait être écrit comme suit :

```
for (size_t i; i < nelems; i++)
    elem[i] = i;
```

Un consensus assez bien établi est qu'une variable commencant par `n` peut signifier *number of*.

Chapitre 4

Numération

La numération désigne le mode de représentation des nombres (e.g. cardinaux, ordinaux), leur base (système binaire, ternaire, quinaire, décimale ou vicésimale), ainsi que leur codification, IEEE 754, complément à un, complément à deux. Bien comprendre les bases de la numération est importante pour l'ingénieur développeur car il est souvent amené à effectuer des opérations de bas niveau sur les nombres.

Ce chapitre n'est essentiel qu'au programmeur de bas niveau, l'électronicien ou l'informaticien technique. Bien comprendre la numération permet de mieux se représenter la manière dont l'ordinateur traite les données au niveau le plus fondamental : le bit.

4.1 Bases

Une base désigne la valeur dont les puissances successives interviennent dans l'écriture des nombres dans la numération positionnelle, laquelle est un procédé par lequel l'écriture des nombres est composé de chiffres ou symboles reliés à leur position voisine par un multiplicateur, appelé base du système de numération.

Sans cette connaissance à priori du système de numération utilisé, il vous est impossible d'interprêter ces nombres :

```
69128
11027
j4b12
>>!!0
□□□□
□□ □□□
```

4.1.1 Système décimal

Le système décimal est le système de numération utilisant la base dix et le plus utilisé par les humains au vingt et unième siècle, ce qui n'a pas toujours été le cas, par exemple les anciennes civilisations de Mésopotamie (Sumer ou Babylone) utilisaient un système positionnel de base sexagésimale (60), la civilisation Maya utilisait un système de base 20 de même que certaines langues celtiques dont il reste aujourd'hui quelques trace en français avec la dénomination *quatre-vingt*.

L'exemple suivant montre l'écriture de 1506 en écriture hiéroglyphique `(1000+100+100+100+100+100+1+1+1+1+1+1)`. Il s'agit d'une numération additive.

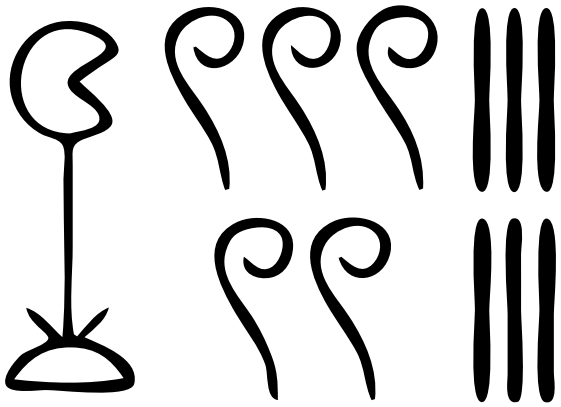


Fig. 4.1 – 1506 en écriture hiéroglyphique

Notre système de représentation des nombres est le système de numération indo-arabe qui employe une notation positionnelle et dix chiffres allant de zéro à neuf :

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

Un nombre peut être décomposé en puissances successives :

$$1506_{10} = 1 \cdot 10^3 + 5 \cdot 10^2 + 0 \cdot 10^1 + 6 \cdot 10^0$$

La base dix n'est pas utilsée dans les ordinateurs car elle nécessite la manipulation de dix états ce qui est difficile avec les systèmes logiques à deux états ; le stockage d'un bit en mémoire étant généralement assuré par des transistors.

4.1.2 Système binaire

Le système binaire est similaire au système décimal mais utilise la base deux. Les symboles utilisés pour exprimer ces deux états possibles sont d'ailleurs emprunté au système indo-arabe :

```
0, 1 = false, true = F, T
```

En termes technique ces états sont le plus souvent représentés par des signaux électriques dont souvent l'un des deux états est dit récessif tandis que l'autre est dit dominant.

Un nombre binaire peut être également décomposé en puissances successives :

$$1101_2 = 1 \cdot 2^3 + 1 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0$$

Exercice

Combien de valeurs décimales peuvent être représentées avec 10-bits ?

Avec une base binaire 2 et 10 bits, le total représentable est :

$$2^10 = 1024$$

Soit les nombres de 0 à 1023.

4.1.3 Système octal

Inventé par Charles XII de Suède, le système de numération octal utilise 8 symboles emprunté au système indo-arabe. Il pourrait avoir été utilisé par l'homme en comptant soit les jointures des phalanges proximales (trous entre les doigts), ou les doigts différents des pouces.

```
0 1 2 3 4 5 6 7
```

Un nombre octal peut également être décomposé en puissances successives :

$$1607_8 = 1 \cdot 8^3 + 6 \cdot 8^2 + 0 \cdot 8^1 + 7 \cdot 8^0$$

Au début de l'informatique la base octale fut très utilisée car il est très facile de la construire à partir de la numération binaire, en regroupant les chiffres par triplets :

```
010'111'100'001_2 = 2741_8
```

En C, un nombre octal est écrit en préfixant la valeur à représenter d'un zéro. Attention donc à ne pas confondre :

```c
int octal = 042;
int decimal = 42;

assert(octal != decimal);
```

Il est également possible de faire référence à un caractère en utilsant l'échappement octal :

```c
char cr = '\015';
char msg = "Hell\0157\040World";
```

4.1.4 Système hexadécimal

Ce système de numération positionnel en base 16 est le plus utilisé en informatique pour exprimer des grandeurs binaires. Il utilise les dix symboles du système indo-arabe, plus les lettres de A à F. Il n'y a pas de réel consensus quant à la casse des lettres.

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

L'écriture peut également être décomposée en puissances successives :

$$1AC7_{16} = (1 \cdot 16^3 + 10 \cdot 16^2 + 12 \cdot 16^1 + 7 \cdot 16^0)_{10} = 41415_{10}$$

Il est très pratique en électronique et en informatique d'utiliser ce système de représentation ou chaque chiffre hexadécimal représente un quadruplet, soit deux caractères hexadécimaux par octet (n'est-ce pas élégant ?) :

```
0101'1110'0001₂ = 5E1₁₆
```

L'ingénieur doit connaître la correspondance hexadécimale de tous les quadruplets aussi bien que ses tables de multiplications :

Binaire	Hexadécimal	Octal	Décimal
0b0000	0x0	00	0
0b0001	0x1	01	1
0b0010	0x2	02	2
0b0011	0x3	03	3
0b0100	0x4	04	4
0b0101	0x5	05	5
0b0110	0x6	06	6
0b0111	0x7	07	7
0b1000	0x8	10	8
0b1001	0x9	11	0
0b1010	0xA	12	10
0b1011	0xB	13	11
0b1100	0xC	14	12
0b1101	0xD	15	13
0b1110	0xE	16	14
0b1111	0xF	17	15

Le fichier *albatros.txt* contient un extrait du poème de Baudelaire, l'ingénieur en proie à un bogue lié à de l'encodage de caractère cherche à comprendre et utilise le programme `hexdump` pour lister le contenu hexadécimal de son fichier :

```
$ hexdump -C albatros.txt
00000000  53 6f 75 76 65 6e 74 2c  20 70 6f 75 72 20 73 27 ␣
↪|Souvent, pour s'|
00000010  61 6d 75 73 65 72 2c 20  6c 65 73 20 68 6f 6d 6d  |amuser,
↪ les homm|
```

(suite sur la page suivante)

(suite de la page précédente)

```
00000020  65 73 20 64 27 c3 a9 71  75 69 70 61 67 65 0d 0a  |es d'..
 ↪quipage..|
00000030  50 72 65 6e 6e 65 6e 74  20 64 65 73 20 61 6c 62 ␣
 ↪|Prennent des alb|
00000040  61 74 72 6f 73 2c 20 76  61 73 74 65 73 20 6f 69  |atros,␣
 ↪vastes oi|
00000050  73 65 61 75 78 20 64 65  73 20 6d 65 72 73 2c 0d  |seaux␣
 ↪des mers,.|
00000060  0a 51 75 69 20 73 75 69  76 65 6e 74 2c 20 69 6e  |.Qui␣
 ↪suivent, in|
00000070  64 6f 6c 65 6e 74 73 20  63 6f 6d 70 61 67 6e 6f ␣
 ↪|dolents compagno|
00000080  6e 73 20 64 65 20 76 6f  79 61 67 65 2c 0d 0a 4c  |ns de␣
 ↪voyage,..L|
00000090  65 20 6e 61 76 69 72 65  20 67 6c 69 73 73 61 6e  |e␣
 ↪navire glissan|
000000a0  74 20 73 75 72 20 6c 65  73 20 67 6f 75 66 66 72  |t sur␣
 ↪les gouffr|
000000b0  65 73 20 61 6d 65 72 73  2e 0d 0a 0d 0a 2e 2e 2e  |es␣
 ↪amers.........|
000000c0  0d 0a 0d 0a 43 65 20 76  6f 79 61 67 65 75 72 20  |....Ce␣
 ↪voyageur |
000000d0  61 69 6c 65 cc 81 2c 20  63 6f 6d 6d 65 20 69 6c  |aile..,
 ↪ comme il|
000000e0  20 65 73 74 20 67 61 75  63 68 65 20 65 74 20 76  | est␣
 ↪gauche et v|
000000f0  65 75 6c 65 e2 80 af 21  0d 0a 4c 75 69 2c 20 6e  |eule...
 ↪!..Lui, n|
00000100  61 67 75 c3 a8 72 65 20  73 69 20 62 65 61 75 2c  |agu..
 ↪re si beau,|
00000110  20 71 75 27 69 6c 20 65  73 74 20 63 6f 6d 69 71  | qu'il␣
 ↪est comiq|
00000120  75 65 20 65 74 20 6c 61  69 64 e2 80 af 21 0d 0a  |ue et␣
 ↪laid...!..|
00000130  4c 27 75 6e 20 61 67 61  63 65 20 73 6f 6e 20 62  |L'un␣
 ↪agace son b|
00000140  65 63 20 61 76 65 63 20  75 6e 20 62 72 c3 bb 6c  |ec␣
 ↪avec un br..l|
00000150  65 2d 67 75 65 75 6c 65  2c 0d 0a 4c 27 61 75 74  |e-
 ↪gueule,..L'aut|
00000160  72 65 20 6d 69 6d 65 2c  20 65 6e 20 62 6f 69 74  |re␣
 ↪mime, en boit|
00000170  61 6e 74 2c 20 6c 27 69  6e 66 69 72 6d 65 20 71  |ant, l
 ↪'infirme q|
00000180  75 69 20 76 6f 6c 61 69  74 e2 80 af 21           |ui␣
 ↪volait...!|
0000018d
```

Il lit à gauche l'offset mémoire de chaque ligne, au milieu le contenu hexadécimal, chaque caractère encodé sur 8 bits étant symbolisé par deux caractères hexadécimaux, et à droite

le texte ou chaque caractère non-imprimable est remplacé par un point. On observe notament ici que :

— é de équipage est encodé avec `\xc3\xa9` ce qui est le caractère unicode U+0065
— é de ailé est encodé avec *exccx81*, soit le caractère e suivi du diacritique ´ U+0301
— Une espace fine insécable `\xe2\x80\xaf` est utilisée avant les `!`, ce qui est le caractère unicode U+202F, ainsi que recommandé par l'académie Française.

Ce fichier est donc convenablement encodé en UTF-8 quant au bogue de notre ami ingénieur il concerne probablement les deux manières distinctes utilisées pour encoder le é.
Exercice

Calculer la valeur décimale des nombres suivants et donnez le détail du calcul :

```
0xaaaa
0b1100101
0x1010
129
0216
```

```
0xaaaa    ≡ 43690
0b1100101 ≡   101
0x1010    ≡  4112
129       ≡   129 (n'est-ce pas ?)
0216      ≡   142
```

4.1.5 Conversions de bases

La conversion d'une base quelconque en système décimal utilise la relation suivante :

$$\sum_{i=0}^{n-1} h_i \cdot b^i$$

où :

n Le nombre de chiffres

b La base du système d'entrée

h_i La valeur du chiffre à la position i

Ainsi, la valeur AP7 exprimé en base tritrigesimale (base 33) et utilisée pour représenter les plaques des véhicules à Hong Kong peut se convertir en décimal après avoir pris connaissance de la correspondance d'un symbole tritrigesimal vers le système décimal :

```
Tritrigesimal -> Décimal :

 0  1  2  3  4  5  6  7  8  9  A  B  C  D  E  F
 0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15

 G  H  I  K  L  M  N  P  R  S  T  U  V  W  X  Y  Z
16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32
```

(suite sur la page suivante)

(suite de la page précédente)

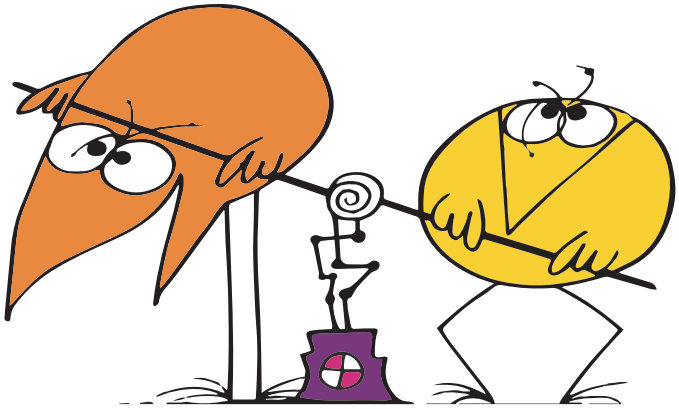
```
Conversion :

AP7 -> 10 * 33**2 + 23 * 33**1 + 7 * 33**0 -> 11'656
```

La conversion d'une grandeur décimale vers une base quelconque est plus compliquée. La conversion d'un nombre du système décimal au système binaire s'effectue simplement par une suite de divisions pour lesquelles on notera le reste.

Pour chaque division par 2, on note le reste et tant que le quotient n'est pas nul, on itère l'opération. Le résultat en binaire est la suite des restes lus dans le sens inverse :

```
n = 209

209 / 2 == 104, 209 % 2 == 1  ^ sens de lecture des restes
104 / 2 ==  52, 104 % 2 == 0  |
 52 / 2 ==  26,  52 % 2 == 0  |
 26 / 2 ==  13,  26 % 2 == 0  |
 13 / 2 ==   6,  13 % 2 == 1  |
  6 / 2 ==   3,   6 % 2 == 0  |
  3 / 2 ==   1,   3 % 2 == 1  |
  1 / 2 ==   0,   1 % 2 == 1  |

209 == 0b11010001
```

Exercice



Les Shadocks ne connaissent que quatre mots : GA, BU, ZO, MEU. La vidéo Comment compter comme les Shadocks en explique le principe.

Convertir −□○◿○ (BU ZO GA MEU GA) en décimal.

Le système Shadock est un système quaternaire similaire au système du génôme humain basé sur quatre bases nucléiques. Assignons donc aux symboles Shadocks les symboles du système indo-arabe que nous connaissons mieux :

```
0 ○ (GA)
1 − (BU)
```

(suite sur la page suivante)

(suite de la page précédente)

```
2 □ (ZO)
3 ◿ (MEU)
```

Le nombre d'entrée `-□0◿0` peut ainsi s'exprimer :

```
-□○◿○ ≡ 12030₄
```

En appliquant la méthode du cours on obtient :

$$1 \cdot 4^4 + 2 \cdot 4^3 + 0 \cdot 4^2 + 3 \cdot 4^1 + 0 \cdot 4^0 = 396_{10}$$

Indication : Depuis un terminal Python vous pouvez simplement utiliser `int("12030", 4)`

4.2 Entiers relatifs

Vous le savez maintenant, l'interprétation d'une valeur binaire n'est possible qu'en ayant connaissance de son encodage et s'agissant d'entiers, on peut se demander comment stocker des valeurs négatives.

Une approche naïve est de réserver une partie de la mémoire pour des entiers positifs et une autre pour des entiers négatifs et stocker la correspondance binaire/décimale simplement. L'ennui pour les variables c'est que le contenu peut changer et qu'il serait préférable de stocker le signe avec la valeur.

4.2.1 Bit de signe

On peut se réserver un bit de signe, par exemple le $8^{\text{ième}}$ bit d'un `char`.

```
┌─┐ ┌─┬─┬─┬─┬─┬─┬─┐
│0│ │1│0│1│0│0│1│1│ = (0 * (-1)) * 0b1010011 = 83
└─┘ └─┴─┴─┴─┴─┴─┴─┘

┌─┐ ┌─┬─┬─┬─┬─┬─┬─┐
│1│ │1│0│1│0│0│1│1│ = (1 * (-1)) * 0b1010011 = -83
└─┘ └─┴─┴─┴─┴─┴─┴─┘
```

Cette méthode impose le sacrifice d'un bit et donc l'intervalle représentable est ici de `[-127..127]`. On ajoutera qu'il existe alors deux zéros, le zéro négatif `0b00000000`, et le zéro positif `0b10000000` ce qui peut poser des problèmes pour les comparaisons.

```
000   001   010   011   100   101   110   111
-+-----+-----+-----+-----+-----+-----+-----+--->
```

(suite sur la page suivante)

(suite de la page précédente)

```
000   001   010   011   100   101   110   111
-+-----+-----+-----+->  -+-----+-----+-----+---> Méthode du bit de␣
↪signe
 0     1     2     3     0    -1    -2    -3
```

De plus les additions et soustractions sont difficile car il n'est pas possible d'effecuer des opérations simples :

```
  00000010 (2)
- 00000101 (5)
----------
  11111101 (-125)    2 - 5 != -125
```

En résumé, la solution utilsant un bit de signe pose deux problèmes :

— Les opérations ne sont plus triviales, et un algorithme particulier doit être mis en place

— Le double zéro (positif et négatif) est génant

4.2.2 Complément à un

Le **complément à un** est une methode plus maline utilisée dans les premiers ordinateurs comme le CDC 6600 (1964) ou le UNIVAC 1107 (1962). Il existe également un bit de signe mais il est implicite.

Le complément à un tire son nom de sa définition générique nommée *radix-complement* ou complément de base et s'exprime par :

$$b^n - y$$

où

b La base du système positionnel utilisé

n Le nombre de chiffres maximal du nombre considéré

y La valeur à complémenter.

Ainsi il est facile d'écrire le complément à neuf :

```
0 1 2 3 4 5 6 7 8 9
        |
        | Complément à 9
        v
9 8 7 6 5 4 3 2 1 0
```

On notera avec beaucoup d'intérêt qu'un calcul est possible avec cette méthode. A gauche on a une soustraction classique, à droite on remplace la soustraction par une addition ainsi que les valeurs négatives par leur complément à 9. Le résultat `939` correspond à `60`.

```
  150      150
- 210    + 789
-----    -----
  -60      939
```

Notons que le cas précis de l'inversion des chiffres correspond au complément de la base, moins un. L'inversion des bits binaire est donc le complément à $(2 - 1) = 1$.

```
000   001   010   011   100   101   110   111
-+-----+-----+-----+-----+-----+-----+-----+--->

000   001   010   011   100   101   110   111
-+-----+-----+-----+-> <-+-----+-----+-----+--- complément à un
 0     1     2     3    -3    -2    -1     0
```

Reprenons l'exemple précédant de soustraction, on notera que l'opération fonctionne en soustrayant 1 au résultat du calcul.

```
  00000010 (2)
+ 11111010 (5)
----------
  11111100 (-3)
-         1
----------
  11111100 (-3)
```

En résumé, la méthode du complément à 1 :

- Les opérations redeviennent presque triviale, mais il est nécessaire de soustraire 1 au résultat
- Le double zéro (positif et négatif) est génant

4.2.3 Complément à deux

Le complément à deux n'est rien d'autre que le complément à un **plus** un. C'est donc une amusante plaisanterie des informaticiens dans laquelle les étapes nécessaires sont :

1. Calculer le complément à un du nombre d'entrée.
2. Ajouter 1 au résultat.

Oui, et alors, quelle est la valeur ajoutée ? Surprenamment, on résouds tous les problèmes amenés par le complément à un :

```
000   001   010   011   100   101   110   111
-+-----+-----+-----+-----+-----+-----+-----+--->
 0     1     2     3     4     5     6     7     sans complément
 0     1     2     3    -3    -2    -1     0     complément à un
 0     1     2     3    -4    -3    -2    -1     complément à deux
```

Au niveau du calcul :

```
  2        00000010
- 5      + 11111011   (~0b101 + 1 == 0b11111011)
---      -----------
 -3        11111101   (~0b11111101 + 1 == 0b11 == 3)
```

Les avantages :

— Les opérations sont triviales.
— Le problème du double zéro est résolu.
— On gagne une valeur négative `[-128..+127]` contre `[-127..+127]` avec les méthodes précédamment étudiées.

4.3 Opérations logiques

4.3.1 Opérations bit à bit

..index :: bitwise

Les opérations bit-à-bit (*bitwise*) disponibles en C sont les suivantes :

Opérateur	Description	Exemple
&	ET binaire	`(0b1101 & 0b1010) == 0b1000`
|	OU binaire	`(0b1101 | 0b1010) == 0b1111`
^	XOR binaire	`(0b1101 ^ 0b1010) == 0b0111`
~	Complément à un	`~0b11011010 == 0b00100101`
<<	Décalage à gauche	`(0b1101 << 3) == 0b1101000`
>>	Décalage à droite	`(0b1101 >> 2) == 0b11`

Le ET logique est identique à la multiplication appliquée bit à bit et ne génère pas de retenue.

A	B	A B
0	0	0
0	1	0
1	0	0
1	1	1

OU logique

A	B	S
0	0	0
0	1	1
1	0	1
1	1	1

OU EXCLUSIF logique

A	B	A ⌒ B
0	0	0
0	1	1
1	0	1
1	1	0

Complément à un

Le complément à un est simplement la valeur qui permet d'obtenir 1, soit l'inverse de l'entrée en binaire :

A	¬ A
0	1
1	0

4.3.2 Opérateurs arithmétiques

Les opérations arithmétiques nécessitent le plus souvent d'une communication entre les bits. C'est à dire en utilisant une retenue (*carry*). En base décimale, on se souvent de l'addition :

```
  11  ← retenues
  123₁₀
+  89₁₀
-----
  212₁₀
```

En arithmétique binaire, c'est exactement la même chose :

A	B	A + B	C
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

```
 111  111
  11100101₂
+  1100111₂
----------
 101001100₂
```

Exercice

Une unité de calcul arithmétique (ALU) est capable d'effectuer les 4 opérations de bases comprenants additions et soustractions.

Traduisez les opérandes ci-dessous en binaire, puis poser l'addition en binaire.

1. 1 + 51
2. 51 - 7
3. 204 + 51
4. 204 + 204 (sur 8-bits)

1. **1 + 51**

```
       11
         1₂
+   110011₂  (2⁵ + 2⁴ + 2¹+ 2⁰ ≡ 51)
----------
    110100₂
```

2. **51 - 7**

```
  …111  11
  …000110011₂  (2⁵ + 2⁴ + 2¹ + 2⁰ ≡ 51)
+ …111111001₂  (complément à deux) 2³ + 2¹ + 2⁰ ≡ 111₂ → !7␣
↪+ 1 ≡ …111001₂)
 -----------
  …000101100₂  (2⁵ + 2³ + 2₂ ≡ 44)
```

3. **204 + 51**

```
    11001100₂
+     110011₂
 -----------
  …011111111₂  (2⁸ - 1 ≡ 255)
```

4. **204 + 204 (sur 8-bits)**

```
   1|1  11
    |11001100₂
+   |11001100₂
 ---+--------
   1|10011000₂  (152, le résultat complet devrait être 2⁸ +␣
↪152 ≡ 408)
```

4.3.3 Lois de De Morgan

Les lois de De Morgan sont des identités logiques formulées il y a près de deux siècles : sachant qu'en logique classique, la négation d'une conjonction implique la disjonction des négations et que la conjonction de négations implique la négation d'une disjonction, on peut alors eprimer que :

```
¬ (P ∧ Q) ⇒ ((¬ P) ∨ (¬ Q))
((¬ P) ∧ (¬ Q)) ⇒ ¬ (P ∨ Q)
```

Ces opérations logiques sont très utiles en programmation où elles permettent de simplifier certains algorithmes.

A titre d'exemple, les opérations suivantes sont donc équivalentes :

```
int a = 0b110010011;
int b = 0b001110101;

assert(a | b == ~a & ~b);
assert(~a & ~b == ~(a | b));
```

En logique booléenne on exprime la négation par une bar p.ex. $\bar{P}$. Exercice

Utiliser les relations de De Morgan pour simplifier l'expression suivante

$$D \cdot E + \bar{D} + \bar{E}$$

Si l'on applique De Morgan ($X\bar{Y} = \bar{X} + \bar{Y}$) :

$$D \cdot E + \bar{D} + \bar{E}$$

4.3.4 Arrondi

En programmation, la notion d'arrondi (rounding) est beaucoup plus complexe qu'imaginée. Un nombre réel peut être converti en un nombre entier de plusieurs manières dont voici une liste non exaustive :

- **tronqué** (*truncate*) lorsque la partie fractionnaire est simplement enlevée
- **arrondi à l'entier supérieur** (*rounding up*)
- **arrondi à l'entier inférieur** (*rounding down*)
- **arrondi en direction du zéro** (*rounding towards zero*)
- **arrondi loin du zéro** (*rounding away from zero*)
- **arrondi au plus proche entier** (*rounding to the nearest integer*)
- **arrondi la moitié en direction de l'infini** (*rounding half up*)

Selon le langage de programmation et la méthode utilisée, le mécanisme d'arrondi sera différent. En C, la bibliothèque mathématique offre les fonctions `ceil` pour l'arrondi au plafond (entier supérieur), `floor` pour arrondi au plancher (entier inférieur) et `round` pour l'arrondi au plus proche (*nearest*). Il existe également fonction `trunc` qui tronque la valeur en supprimant la partie fractionnaire.

Le fonctionnement de la fonction `round` n'est pas unanime entre les mathématiciens et les programmeurs. C utilise l'arrondi au plus proche, c'est à dire que -23.5 donne -24 et 23.5 donne 24.

Note : En Python ou en Java, c'est la méthode du *commercial rounding* qui a été choisie. Elle peut paraître contre intuitive car `round(3.5)` donne 4 mais `round(4.5)` donne 4 aussi.

Exercice

Soit deux variables entières `a` et `b`, chacune contenant une valeur différente. Écrivez les instructions permettant d'échanger les valeurs de a et de b sans utiliser de valeurs intermédiaire. Indice : utilisez l'opérateur XOR `^`.

Testez votre solution

```
a ^= b;
b ^= a;
a ^= b;
```

Chapitre 5

Opérateurs

Un opérateur applique une opération à une (opérateur unitaire), deux ou trois (ternaire) entrées.

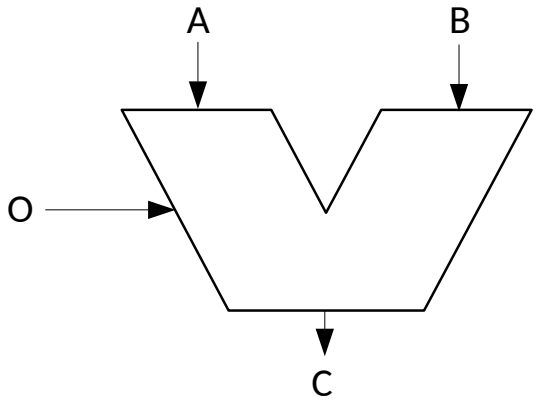


Fig. 5.1 – Unité de calcul arithmétique (ALU) composées de deux entrées `A` et `B`, d'une sortie `C` et d'un mode opératoire `O`.

```
c = a + b;
```

Un opérateur possède plusieurs propriétés :

Une priorité La multiplication * est plus prioritaire que l'addition `+`

Une associativité L'opérateur d'affectation possède une associativité à droite, c'est à dire que l'opérande à droite de l'opérateur sera évalué en premier

Un point de séquence Certains opérateurs comme `&&`, `||`, `?` ou `,` possèdent un point de séquence garantissant que l'exécution séquentielle du programme sera respectée avant et après ce point. Par exemple si dans l'expression `i < 12 && j > 2` la valeur de `i` est plus grande que 12, le test `j > 2` ne sera jamais effectué.

L'opérateur `&&` garanti l'ordre des choses ce qui n'est pas le cas avec l'affectation `=`.

5.1 Opérateurs relationnels

Les opérateurs relationnels permettent de comparer deux entités. Le résultat d'un opérateur relationnel est toujours un **boolean** c'est à dire que le résultat d'une comparaison est soit **vrai**, soit **faux**.

Opérateur	Description	Exemple vrai
`==`	Egal	`42 == 0x101010`
`>=`	Supérieur ou égal	`9 >= 9`
`<=`	Inférieur ou égal	`-8 <= 8`
`>`	Strictement supérieur	`0x31 > '0'`
`<`	Inférieur	`8 < 12.33`
`!=`	Différent	`'a' != 'c'`

Un opérateur relationnel est plus prioritaire que l'opérateur d'affectation et donc l'écriture suivante applique le test d'égalité entre `a` et `b` et le résultat de ce test `1` ou `0` sera affecté à la variable `c` :

```
int a = 2, b = 3;
int c = a == b;
```

Les opérateurs relationnels sont le plus souvent utilisés dans des structures de contrôles :

```
if (a == b) {
    printf("Egaux");
} else {
    printf("Pas égaux");
}
```

5.2 Opérateurs arithmétiques

Aux 4 opérations de base, le C ajoute l'opération modulo, qui est le reste d'une division entière.

- `+` Addition
- `-` Soustraction
- `*` Multiplication
- `/` Division
- `%` Modulo

Attention néanmoins aux types des variables impliquées. La division `5 / 2` donnera `2` et non `2.5` car les deux valeurs fournies sont entières.

Le modulo est le reste de la division entière. L'assertion suivante est donc vraie : `13 % 4 == 1`, car 13 divisé par 4 égal 3 et il reste 1.

Les opérateurs arithmétiques sont tributaires des types sur lesquels ils s'appliquent. L'addition de deux entiers 8 bits `120 + 120` ne fera pas `240` car le type ne permet pas de stoquer des valeurs plus grandes que `127`.

5.3 Opérateurs bit à bit

Les opérations binaires agissent directement sur les bits d'une entrée :

- `&` ET arithmétique
- `|` OU arithmétique
- `^` XOR arithmétique
- `<<` Décalage à gauche
- `>>` Décalage à droite
- `~` Inversion binaire

5.4 Opérateurs d'affectation

- `=` Affectation simple
- `+=` Affectation par addition
- `-=` Affectation par soustraction
- `*=` Affectation par multiplication
- `/=` Affectation par division
- `%=` Affectation par modulo
- `&=` Affectation par ET arithmétique
- `|=` Affectation par OU arithmétique
- `^=` Affectation par XOR arithmétique
- `<<=` Affectation par décalage à gauche
- `>>=` Affectation par décalage à droite

Les opérateurs d'affectation combinés peuvent tous des sucres syntaxiques : `a += b` est strictement équivalent à `a = a + b`. De la même manière `a <<= b` est une autre manière d'écrire `a = a << b`.

5.5 Opérateurs logiques

- `&&` ET logique
- `||` OU logique

5.6 Opérateurs d'incrémentation

— `()++` Post-incrémentation
— `++()` Pré-incrémentation
— `()--` Post-décrémentation
— `--()` Pré-décrémentation

5.7 Opérateur ternaire

— `()?():()` Opérateur ternaire

Cet opérateur permet sur une seule ligne d'évaluer une expression et de renvoyer une valeur ou une autre selon que l'expression est vraie ou fausse. **valeur = (condition ? valeur si condition vraie : valeur si condition fausse) ;**

Important : seule la valeur utilisée pour le résultat est évaluée.

```
val_max = (a > b ? a : b);  // retourne la valeur max entre a et b
```

5.8 Opérateur de transtypage

— `()()`

5.9 Opérateur séquentiel

L'opérateur séquentiel (*comma operator*) permet l'exécution ordonné d'opérations, et retourne la dernière valeur. Son utilisation est courament limitée soit aux décalarations de variables, soit au boucles `for` :

```
for (size_t i = 0, j = 10; i != j; i++, j--) { /* ... */ }
```

Dans le cas ci-dessus, il n'est pas possible de séparer les instructions `i++` et `j--` par un point virgule, l'opérateur virgule permet alors de combiner plusieurs instructions en une seule.

Une particularité de cet opérateur est que seule la dernière valeur est retournée :

```
assert(3 == (1, 2, 3))
```

L'opérateur agit également comme un *Point de séquence*, c'est à dire que l'ordre des étapes sont respectés. Exercice

Que sera-t-il affiché à l'écran ?

```c
int i = 0;
printf("%d", (++i, i++, ++i));
```

5.10 Opérateur sizeof

— `sizeof`

5.11 Les opérateurs logiques

Ils permettent de coupler des opérateurs de comparaison entre eux pour effectuer des tests un peu plus complexe.

5.11.1 ET logique

Ecriture :

```c
resultat = condition1 && condition2;
```

Table de vérité

condition1	condition2	résultat
0	0	0
0	1	0
1	0	0
1	1	1

5.11.2 OU logique

Ecriture :

```c
resultat = condition1 || condition2;
```

Table de vérité

condition1	condition2	résultat
0	0	0
0	1	1
1	0	1
1	1	1

5.11.3 Inversion logique

Ecriture :

```
resultat = !condition1;
```

Table de vérité

condition1	résultat
0	1
1	0

5.12 Les opérateurs bit-à-bit

Ils permettent d'effectuer des opérations binaire bit à bit sur des types entiers.

5.12.1 Inversion logique ou complément à 1

C'est un opérateur unaire dont l'écriture est :

```
uint8_t a=0x55; // 0101 0101 (binaire)
uint8_t r=0x00;

r = ~a; // résultat r=0xAA (1010 1010)
```

5.12.2 ET logique

Ecriture :

```
uint8_t a=0x55; // 0101 0101 (binaire)
uint8_t b=0x0F; // 0000 1111
uint8_t r=0x00;

r = a & b;  // résultat r=0x05 (0000 0101)
```

5.12.3 OU logique

Ecriture :

```
uint8_t a=0x55; // 0101 0101 (binaire)
uint8_t b=0x0F; // 0000 1111
uint8_t r=0x00;
```

(suite sur la page suivante)

(suite de la page précédente)

```
r = a | b;  // résultat r=0x5F (0101 1111)
```

5.12.4 OU EXCLUSIF logique

Ecriture :

```
uint8_t a=0x55; // 0101 0101 (binaire)
uint8_t b=0x0F; // 0000 1111
uint8_t r=0x00;

r = a ^ b;  // résultat r=0x5A (0101 1010)
```

5.12.5 Décalage à droite

Ecriture :

```
uint8_t a=0xAA; // 1010 1010 (binaire)
uint8_t r=0x00;

r = a >> 1  // résultat r=0x55 (0101 0101)
```

Pour le décalage à droite de valeurs signées, le signe est conservé. Cette opération s'apparente à une division par 2.

5.12.6 Décalage à gauche

Ecriture :

```
uint8_t a=0xAA; // 1010 1010 (binaire)
uint8_t r=0x00;

r = a << 1  // résultat r=0x54 (0101 0100)
```

Cette opération s'apparente à une multiplication par 2.

5.13 Les opérateurs d'incrémentation (++) et de décrémentation (−)

Ces opérateurs, qui ne s'appliquent que sur des nombres entiers, permettent d'ajouter 1 ou de retrancher 1 à une variable, et ce de manière optimisée pour le processeur qui exécute le programme.

Ils peuvent, en outre, être exécutés avant ou après l'évaluation de l'opération. On parle alors de pré-incrémentation ou pré-décrémentation et post-incrémentation ou post-décrémentation.

5.13.1 pré-incrémentation

Ecriture :

```
int32_t i=0, j=0;

j = ++i;    // on obtient i=1 et j=1
```

5.13.2 post-incrémentation

Ecriture :

```
int32_t i=0, j=0;

j = i++;    // on obtient i=1 et j=0
```

5.13.3 pré-décrémentation

Ecriture :

```
int32_t i=0, j=0;

j = --i;    // on obtient i=-1 et j=-1
```

5.13.4 post-décrémentation

Ecriture :

```
int32_t i=0, j=0;

j = i--;    // on obtient i=-1 et j=0
```

5.14 Priorité des opérateurs

La **précédence** est un anglicisme de *precedence* (priorité) qui concerne la priorité des opérateurs, où l'ordre dans lequel les opérateurs sont exécutés. Chacuns connaît la priorité des quatre opérateurs de base (`+`, `-`, `*`, `/`) mais le C et ses nombreux opérateurs est bien plus complexe.

La table suivante indique les règles à suivre pour les précédences des opérateurs en C. La précédence

Priorité	Opérateur	Description	Associativité
1	`++`, `--`	Postfix incréments/décréments	Gauche à Droite
	`()`	Appel de fonction	
	`[]`	Indexage des tableaux	
	`.`	Element d'une structure	
	`->`	Element d'une structure	
2	`++`, `--`	Préfix incréments/décréments	Droite à Gauche
	`+`, `-`	Signe	
	`!`, `~`	NON logique et NON binaire	
	`(type)`	Cast (Transtypage)	
	`*`	Indirection, déréfrencement	
	`&`	Adresse de...	
	`sizeof`	Taille de...	
3	`*`, `/`, `%`	Multiplication, Division, Mod	Gauche à Droite
4	`+`, `-`	Addition, soustraction	
5	`<<`, `>>`	Décalages binaires	
6	`<`, `<=`	Comparaison plus petit que	
	`>`, `>=`	Comparaison plus grand que	
7	`==`, `!=`	Egalité, non égalité	
8	`&`	ET binaire	
9	`^`	OU exclusif binaire	
10	`|`	OU inclusif binaire	
11	`&&`	ET logique	
12	`||`	OU logique	
13	`?:`	Opérateur ternaire	Droite à Gauche
14	`=`	Assignation simple	
	`+=`, `-=`	Assignation par somme/diff	
	`*=`, `/=`, `%=`	Assignation par produit/quotient/modulo	
	`<<=`, `>>=`	Assignation par décalage binaire	
15	`,`	Virgule	Gauche à Droite

Considérons l'exemple suivant :

```c
int i[2] = {10, 20};
int y = 3;

x = 5 + 23 + 34 / ++i[0] & 0xFF << y;
```

Selon la précédance de chaque opérateur ainsi que son associativité on a :

```
[ ] 1
++  2
/   3
+   4
+   4
<<  5
&   8
=   14
```

L'écriture en notation polonaise inversée, donnerait alors

```
34, i, 0, [], ++,  /, 5, 23, +, +, 0xFF, y, <<, &, x, =
```

5.14.1 Associativité

L'associativité des opérateurs (operator associativity) décrit la manière dont sont évaluées les expressions.

Une associativité à gauche pour l'opérateur ~ signifie que l'expression `a ~ b ~ c` sera évaluée `((a) ~ b) ~ c` alors qu'une associativité à droite sera `a ~ (b ~ (c))`.

Il ne faut pas confondre l'associativité *évaluée de gauche à droite* qui est une associativité à *gauche*.

5.14.2 Représentation mémoire des types de données

Nous avons vu au chapitre sur les types de données que les types C définis par défaut sont représentés en mémoire sur 1, 2, 4 ou 8 octets. On comprend aisément que plus cette taille est importante, plus on gagne en précision ou en grandeur représentable. La promotion numérique régit les conversions effectuées implicitement par le langage C lorsqu'on convertit une donnée d'un type vers un autre. Cette promotion tend à conserver le maximum de précision lorsqu'on effectue des calcul entre types différents (ex : l'addition d'un *int* avec un *double* donne un type *double*). **Règles de base :**

- les opérateurs ne peuvent agir que sur des types identiques
- quand les types sont différents, il y a conversion automatique vers le type ayant le plus grand pouvoir de représentation
- les conversions ne sont faites qu'au fur et à mesure des besoins

La **promotion** est l'action de promouvoir un type de donnée en un autre type de donnée plus général. On parle de promotion implicite des entiers lorsqu'un type est promu en un type plus grand automatiquement par le compilateur.

5.15 Valeurs gauche

Une valeur gauche (*lvalue*) est une particularité de certains langage de programmation qui définissent ce qui peut se trouver à gauche d'une affectation. Ainsi dans `x = y`, `x` est une valeur gauche. Néanmoins, l'expression `x = y` est aussi une valeur gauche :

```c
int x, y, z;

x = y = z;    // (1)
(x = y) = z;  // (2)
```

1. L'associativité de `=` est à droite donc cette expression est équivalente à `x = (y = (z))` qui évite toute ambiguïté.
2. **En forcant l'associativité à gauche, on essaie d'assigner `z` à une *lvalue* et le compila**

```
4:8: error: lvalue required as left operand of assignment
    (x = y) = z;
            ^
```

Voici quelques exemples de valeurs gauche :
- `x /= y`
- `++x`
- `(x ? y : z)`

5.16 Optimisation

Le compilateur est en règle général plus malin que le développeur. L'optimiseur de code (lorsque compilé avec `-O2` sous `gcc`), va regrouper certaines instructions, modifier l'ordre de certaines déclarations pour réduire soit l'empreinte mémoire du code, soit accélérer son exécution.

Ainsi l'expression suivante, ne sera pas calculée à l'exécution, mais à la compilation :

```c
int num = (4 + 7 * 10) >> 2;
```

De même que ce test n'effectura pas une division mais testera simplement le dernier bit de `a` :

```c
if (a % 2) {
    puts("Pair");
} else {
    puts("Impair");
}
```

Exercice

Soit les déclarations suivantes :

```
char m, n = 2, d = 0x55, e = 0xAA;
```

Représenter en binaire et en hexadécimal la valeur de tous les bits de la variable `m` après exécution de chacune des instructions suivantes :

1. `m = 1 << n;`
2. `m = ~1 << n;`
3. `m = ~(1 << n);`
4. `m = d | (1 << n);`
5. `m = e | (1 << n);`
6. `m = d ^ (1 << n);`
7. `m = e ^ (1 << n);`
8. `m = d & ~(1 << n);`
9. `m = e & ~(1 << n);`

Exercice

Pour programmer les registres 16-bits d'un composant électronique chargé de gérer des sorties tout ou rien, on doit être capable d'effectuer les opérations suivantes :

— mettre à 1 le bit numéro `n`, `n` étant un entier entre 0 et 15 ;
— mettre à 0 le bit numéro `n`, `n` étant un entier entre 0 et 15 ;
— inverser le bit numéro `n`, `n` étant un entier entre 0 et 15 ;

Pour des questions d'efficacité, ces opérations ne doivent utiliser que les opérateurs bit à bit ou décalage. On appelle `r0` la vairable désignant le registre en mémoire et `n` la variable contenant le numéro du bit à modifier. Écriver les expressions permettant d'effectuer les opérations demandées. Exercice

Considérant les déclarations suivantes :

```
float a, b;
int m, n;
```

Traduire en C les expressions mathématiques ci-dessous ; pour chacune, proposer plusieurs écritures différentes lorsque c'est possible. Le symbole $\leftarrow$ signifie *assignation*

1. $n \leftarrow 8 \cdot n$
2. $a \leftarrow a + 2$
3. $n \leftarrow \begin{cases} m & : m > 0 \\ 0 & : \text{sinon} \end{cases}$
4. $a \leftarrow n$
5. $n \leftarrow \begin{cases} 0 & : m \text{ pair} \\ 1 & : m \text{ impair} \end{cases}$
6. $n \leftarrow \begin{cases} 1 & : m \text{ pair} \\ 0 & : m \text{ impair} \end{cases}$
7. $m \leftarrow 2 \cdot m + 2 \cdot n$
8. $n \leftarrow n + 1$

9. $a \leftarrow \begin{cases} -a & : b < 0 \\ a & : \text{sinon} \end{cases}$
10. $n \leftarrow$ la valeur des 4 bits de poids faible de n

Exercice

Un nombre narcissique ou nombre d'Amstrong est un entier naturel `n` non nul qui est égal à la somme des puissances `p`-ièmes de ses chiffres en base dix, où `p` désigne le nombre de chiffres de `n` :

$$n = \sum_{k=0}^{p-1} x_k 10^k = \sum_{k=0}^{p-1} (x_k)^p \quad \text{avec} \quad x_k \in \{0, \ldots, 9\} \quad \text{et} \quad x_{p-1} \neq 0$$

Par exemple :
- `9` est un nombre narcissique car $9 = 9^1 = 9$
- `153` est un nombre narcissique car $153 = 1^3 + 5^3 + 3^3 = 1 + 125 + 27 = 153$
- `10` n'est pas un nombre narcissique car $10 \neq 1^2 + 0^2 = 1$

Implanter un programme permettant de vérifier si un nombre d'entrée est narcissique ou non. L'exécution est la suivante :

```
$ ./armstrong 153
1

$ ./armstrong 154
0
```

Chapitre 6

Types de données

6.1 Typage

Inhérent au fonctionnement interne de l'ordinateur, un langage de programmation s'abstrait plus ou moins du mode de stockage interne des données telles qu'elles sont enregistrées dans la mémoire. De la même manière que dans la vie réelle, il n'est pas possible de rendre de la monnaie à un vendeur à moins d'un cinquième de centime près, il n'est pas possible pour un ordinateur de stocker des informations numériques avec une précision infinie.

Aussi, les langages de programmation sont dits **typés** lorsqu'ils confient au programmeur la responsabilité de choisir comment une information sera stockée en mémoire, et **non typés** lorsque ce choix est implicite. Chacun des langages à ses avantages et ses inconvénients et pour reprendre l'exemple du rendu de monnaie, il serait ennuyant d'autoriser d'enregistrer des informations financières avec une précision meilleure qu'une pièce de cinq centimes, car il serait alors impossible à un caissier de rendre la monnaie correctement. Dans cette situation on préférera les langages **typés** et heureusement C est un langage fortement typé.

Les types de données ne se bornent pas qu'aux informations numériques, il existe des types plus complexes qui permettent par exemple de traiter des caractères tels que `A` ou `B`. Ce chapitre à pour objectif de familiariser le lecteur aux différents types de données disponibles en C.

Note : Standard ISO
Les ingénieurs sont friands des standards et qui plus est lorsqu'ils sont internationaux. Ainsi afin d'éviter le crash malheureux d'une fusée causé par la mésentente de deux ingénieurs de différents pays, il existe la norme **ISO 80000-2** qui définit précisément ce qu'est un entier, s'il doit inclure ou non le zéro, que sont les nombres réels, etc. Bien entendu les compilateurs, s'ils sont bien faits, cherchent à respecter au mieux ces normes internationales, et vous ?

6.2 Stockage et interprétation

Rappelez-vous qu'un ordinateur ne peut stocker l'information que sous forme binaire et qu'il n'est à même de manipuler ces informations que par paquets de bytes. Aussi un ordinateur 64-bits manipulera avec aisance des paquets de 64-bits, mais plus difficilement des paquets de 32-bits. Ajoutons qu'il existe encore des microcontrôleurs 8-bits utilisés dans des dispositifs à faible consommation et qui peinent à manipuler des types de plus grande taille. Stocker une température avec une trop grande précision et effectuer des opérations mathématiques sur toute la précision serait une erreur, car le microcontrôleur n'est simplement pas adapté à manipuler ce type d'information.

Considérons le paquet de 32-bit suivant, êtes-vous à même d'en donner une signification ?

```
01000000 01001001 00001111 11011011
```

Il pourrait s'agir :

- **de 4 caractères de 8-bits :**
  - `01000000` `@`
  - `01001001` `I`
  - `00001111` `\x0f`
  - `11011011` `Û`
- ou de 4 nombres de 8-bits : `64`, `73`, `15`, `219`,
- ou de deux nombres de 16-bits `18752` et `56079`,
- ou alors un seul nombre de 32-bit `3675212096`.
- Peut-être est-ce le nombre `-40331460896358400.000000` lu en *little endian*,
- ou encore `3.141592` lu en *big endian*.

Qu'en pensez-vous ?

Lorsque l'on souhaite programmer à bas niveau, vous voyez que la notion de type de donnée est essentielle, car en dehors d'une interprétation subjective : "c'est forcément PI la bonne réponse", rien ne permet à l'ordinateur d'interpréter convenablement l'information enregistrée en mémoire.

Le typage permet de résoudre toute ambiguïté.

```c
int main() {
    union {
        uint8_t u8[4];
        uint16_t u16[2];
        uint32_t u32;
        float f32;
    } u = { 0b01000000, 0b01001001, 0b00001111, 0b11011011 };

    printf("'%c', '%c', '%c', '%c'\n", u.u8[0], u.u8[1], u.u8[2], u.
↪u8[3]);
    printf("%hhu, %hhu, %hhu, %hhu\n", u.u8[0], u.u8[1], u.u8[2], u.
↪u8[3]);
    printf("%hu, %hu\n", u.u16[0], u.u16[1]);
    printf("%u\n", u.u32);
```

(suite sur la page suivante)

(suite de la page précédente)

```
    printf("%f\n", u.f32);
    u.u32 = (
        ((u.u32 >> 24) & 0xff) | // move byte 3 to byte 0
        ((u.u32 << 8) & 0xff0000) | // move byte 1 to byte 2
        ((u.u32 >> 8) & 0xff00) | // move byte 2 to byte 1
        ((u.u32 << 24) & 0xff000000) // byte 0 to byte 3
    );
    printf("%f\n", u.f32);
}
```

6.3 Boutisme

La hantise de l'ingénieur bas-niveau c'est le boutisme aussi appelé *endianess*. Ce terme étrange a été popularisé par l'informaticien Dany Cohen en référence aux Voyages de Gulliver de Jonathan Swift. Dans ce conte les habitants de Lilliput refusent d'obéir à un décret obligeant à manger les oeufs à la coque par le petit bout (petit boutisme/*little endian*), la répression incite les rebelles à manger leurs oeufs par le gros bout (gros boutisme/*big endian*).

Aujourd'hui encore, il existe des microprocesseurs qui fonctionnent en *big endian* alors que d'autres sont en *little endian*. C'est à dire que si une information est stockée en mémoire comme suit :

```
[0x40, 0x49, 0xf, 0xdb]
```

Faut-il la lire de gauche à droite ou de droite à gauche ? Cela vous paraît trivial, mais si cet exemple était mentionné dans un livre rédigé en arabe, quelle serait alors votre réponse ?

Imaginons qu'un programme exécuté sur un microcontrôleur *big-endian* 8-bit envoie par Bluetooth la valeur `1'111'704'645`, qui correspond au nombre de photons ayant frappé un détecteur optique. Il transmet donc les 4 octets suivants : `0x42, 0x43, 0x44, 0x45`. L'ordinateur qui reçoit les informations décode `1'162'101'570`. Les deux or-

dinateurs n'interprètent pas les données de la même façon, et c'est un problème que la plupart des ingénieurs électroniciens rencontrent un jour dans leur carrière.

6.4 Les nombres entiers

Les nombres entiers sont des nombres sans virgule et incluant le zéro. Ils peuvent donc être négatifs, nuls ou positifs. Mathématiquement ils appartiennent à l'ensemble des entiers relatifs.

Comme aucun ordinateur ne dispose d'un espace de stockage infini, ces nombres excluent les infinis positifs et négatifs, et sont donc bornés, cela va de soi.

6.4.1 Les entiers naturels

En mathématiques, un entier naturel est un nombre positif ou nul. Chaque nombre à un successeur unique et peut s'écrire avec une suite finie de chiffres en notation décimale positionnelle, et donc sans signe et sans virgule. L'ensemble des entiers naturels est défini de la façon suivante :

$$\mathbb{N} = 0, 1, 2, 3, ...$$

En informatique, ces nombres sont par conséquent **non signés**, et peuvent prendre des valeurs comprises entre 0 et $2^N - 1$ où N correspond au nombre de bits avec lesquels la valeur numérique sera stockée en mémoire. Il faut naturellement que l'ordinateur sur lequel s'exécute le programme soit capable de supporter le nombre de bits demandé par le programmeur.

En C, on nomme ce type de donnée `unsigned int`, `int` étant le dénominatif du latin *integer* signifiant "entier".

Voici quelques exemples des valeurs minimales et maximales possibles selon le nombre de bits utilisés pour coder l'information numérique :

Profondeur	Minimum	Maximum
8 bits	0	255 ($2^8 - 1$)
16 bits	0	65'535 ($2^{16} - 1$)
32 bits	0	4'294'967'295 ($2^{32} - 1$)
64 bits	0	18'446'744'073'709'551'616 ($2^{64} - 1$)

Notez l'importance du -1 dans la définition du maximum, car la valeur minimum 0 fait partie de l'information même si elle représente une quantité nulle. Il y a donc 256 valeurs possibles pour un nombre entier non signé 8-bits, bien que la valeur maximale ne soit que de 255.

6.4.2 Les entiers relatifs

Mathématiquement un entier relatif appartient à l'ensemble $\mathbb{Z}$:

$$\mathbb{Z} = ..., -3, -2, -1, 0, 1, 2, 3, ...$$

Les entiers relatifs sont des nombres **signés** et donc ils peuvent être **négatifs**, **nuls** ou **positifs** et peuvent prendre des valeurs comprises entre -2^{N-1} et $+2^{N-1} - 1$ où N correspond au nombre de bits avec lesquels la valeur numérique sera stockée en mémoire. Notez l'asymétrie entre la borne positive et négative.

En C on dit que ces nombres sont `signed`. Il est par conséquent correct d'écrire `signed int` bien que le préfixe `signed` soit optionnel, car le standard définit qu'un entier est par défaut signé. La raison à cela relève plus du lourd historique de C qu'à des préceptes logiques et rationnels.

Voici quelques exemples de valeurs minimales et maximales selon le nombre de bits utilisés pour coder l'information :

Profondeur	Minimum	Maximum
8 bits	-128	+127
16 bits	-32'768	+32'767
32 bits	-2'147'483'648	+2'147'483'647

En mémoire ces nombres sont stockés en utilisant le *complément à deux* qui fait l'objet d'une section à part entière.

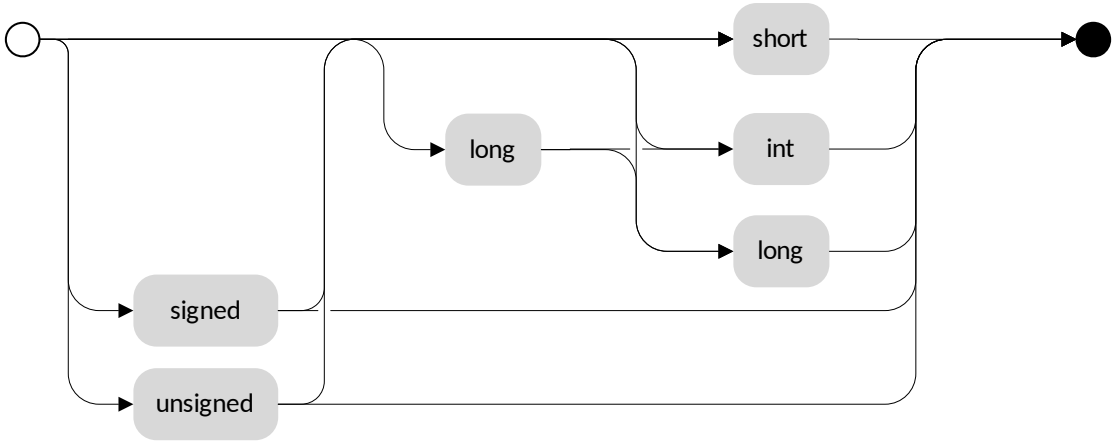
6.4.3 Les entiers bornés

Comme nous l'avons vu, les degrés de liberté pour définir un entier sont :
- Signé ou non signé
- Nombre de bits avec lesquels l'information est stockée en mémoire

À l'origine le standard C restait flou quant au nombre de bits utilisés pour chacun des types et aucune réelle cohérence n'existait pour la construction d'un type. Le modificateur `signed` était optionnel, le préfix `long` ne pouvait s'appliquer qu'au type `int` et `long` et la confusion entre `long` (préfixe) et `long` (type) restait possible. En fait, la plupart des développeurs s'y perdaient et s'y perd toujours ce qui menait à des problèmes de compatibilités des programmes entre eux.

Types standards

La construction d'un type entier C est la suivante :



Ce qu'il faut retenir c'est que chaque type de donnée offre une profondeur d'au moins N bits, ce qui est l'information minimale essentielle pour le programmeur. La liste des types de données standards en C pour les entiers est donnée au Tableau 6.1.

Tableau 6.1 – Types entiers standards

Type	Signe	Profondeur	Format
`char`	?	`CHAR_BIT`	`%c`
`signed char`	signed	au moins 8 bits	`%c`
`unsigned char`	unsigned	au moins 8 bits	`%c`

`short`
`short int`
`signed short`
`signed short int`
	signed	au moins 16 bits	`%hi`

`unsigned short` `unsigned short int`	unsigned	au moins 16 bits	`%hu`
`unsigned` `unsigned int`	unsigned	au moins 32 bits	`%u`

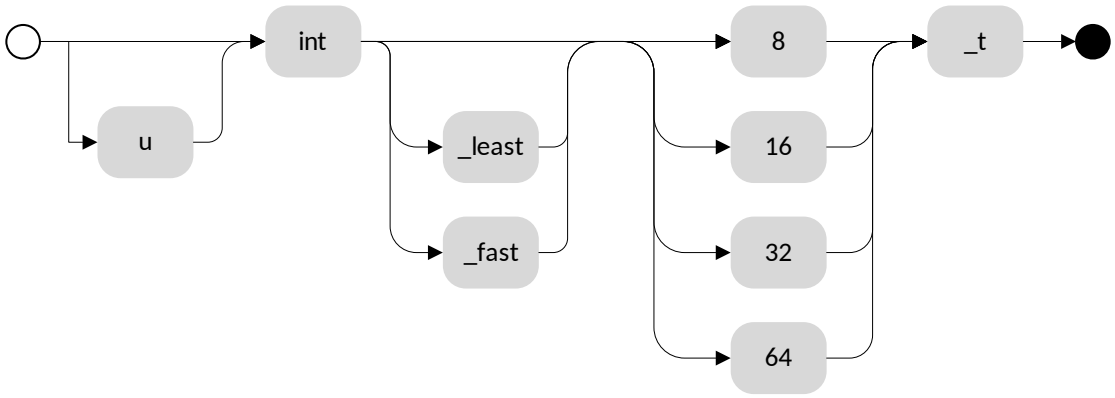
`int`
`signed`
`signed int`
	signed	au moins 32 bits	`%d`

`unsigned` `unsigned int`	unsigned	au moins 32 bits	`%u`

`long`
`long int`
`signed long`
`signed long int`
	signed	au moins 32 bits	`%li`

`unsigned long` `unsigned long int`	unsigned	au moins 32 bits	`%lu`
`long long`	signed	au moins 64 bits	`%lli`

Avec l'avènement de **C99**, une meilleure cohésion des types a été proposée dans le fichier d'en-tête `stdint.h`. Cette bibliothèque standard offre les types suivants :



Types réformés

Voici les types standards qu'il est recommandé d'utiliser lorsque le nombre de bits de l'entier doit être maîtrisé.

Tableau 6.2 – Entiers standard définis par `stdint`

Type	Signe	Profondeur	Format
`uint8_t`	unsigned	8 bits	`%c`
`int8_t`	signed	8 bits	`%c`
`uint16_t`	unsigned	16 bits	`%hu`
`int16_t`	signed	16 bits	`%hi`
`uint32_t`	unsigned	32 bits	`%u`
`int32_t`	signed	32 bits	`%d`
`uint64_t`	unsigned	64 bits	`%llu`
`int64_t`	signed	64 bits	`%lli`

À ces types s'ajoutent les types **rapides** (*fast*) et **minimums** (*least*). Un type nommé `uint_least32_t` garanti l'utilisation du type de donnée utilisant le moins de mémoire et garantissant une profondeur d'au minimum 32 bits. Il est strictement équivalent à `unsigned int`.

Les types rapides, moins utilisés vont automatiquement choisir le type adapté le plus rapide à l'exécution. Par exemple si l'architecture matérielle permet un calcul natif sur 48-bits, elle sera privilégiée par rapport au type 32-bits. Exercice

Donnez la valeur des expressions ci-dessous :

```
25 + 10 + 7 – 3
5 / 2
24 + 5 / 2
```

(suite sur la page suivante)

(suite de la page précédente)

```
(24 + 5) / 2
25 / 5 / 2
25 / (5 / 2)
72 % 5 - 5
72 / 5 - 5
8 % 3
-8 % 3
8 % -3
-8 % -3
```

Exercice

Quel sera le contenu de `j` après l'exécution de l'instruction suivante :

```
uint16_t j = 1024 * 64;
```

Modèle de donnée

Comme nous l'avons évoqué plus haut, la taille des entiers `short`, `int`, ... n'est pas précisément définie par le standard. On sait qu'un `int` contient **au moins** 16-bits mais il peut, selon l'architecture, et aussi le modèle de donnée, prendre n'importe quelle valeur supérieure. Ceci pose des problèmes de portabilité possibles si le développeur n'est pas suffisamment conscensieux et qu'il ne s'appuie pas sur une batterie de tests automatisés.

Admettons que ce développeur sans scrupule développe un programme complexe sur sa machine de guerre 64-bits en utilisant un `int` comme valeur de comptage allant au delà de dix milliards. Après tests, son programme fonctionne sur sa machine, ainsi que celle de son collègue. Mais lorsqu'il livre le programme à son client, le processus crash. En effet, la taille du `int` sur l'ordinateur du client est de 32-bits. Comment peut-on s'affranchir de ce type de problème ?

La première solution est de toujours utiliser les types proposés par `<stdint.h>` lorsque la taille du type nécessaire est supérieure à la valeur garantie. L'autre solution est de se fier au modèle de données :

Tableau 6.3 – Modèle de données

Modèle de donnée	`short`	`int`	`long`	`long long`	`size_t`	Système d'exploitation
LP32	16	16	32		32	Windows 16-bits, Apple Macintosh (très vieux)
ILP32	16	32	32	64	32	Windows x86, Linux/Unix 32-bits
LLP64	16	32	32	64	64	Microsoft Windows x86-64, MinGW
LP64	16	32	64	64	64	Unix, Linux, macOS, Cygwin
ILP64	16	64	64	64	64	HAL (SPARC)
SILP64	64	64	64	64	64	UNICOS (Super ordinateur)

6.5 Les nombres réels

Mathématiquement, les nombres réels $\mathbb{R}$, sont des nombres qui peuvent être représentés par une partie entière, et une liste finie ou infinie de décimales. En informatique, stocker une liste infinie de décimale demanderait une quantité infinie de mémoire et donc, la précision arithmétique est contrainte.

Au début de l'ère des ordinateurs, il n'était possible de stocker que des nombres entiers, mais le besoin de pouvoir stocker des nombres réels s'est rapidement fait sentir. La transition s'est faite progressivement, d'abord par l'apparition de la virgule fixe, puis par la virgule flottante.

Le premier ordinateur avec une capacité de calcul en virgule flottante date de 1942 (ni vous ni moi n'étions probablement né) avec le Zuse's Z4, du nom de son inventeur Konrad Zuse.

6.5.1 Virgule fixe

Prenons l'exemple d'un nombre entier exprimé sur 8-bits, on peut admettre facilement que bien qu'il s'agisse d'un nombre entier, une virgule pourrait être ajoutée au bit zéro sans en modifier sa signification.

```
┌─┬─┬─┬─┬─┬─┬─┬─┐
│0│1│0│1│0│0│1│1│ = 2^6 + 2^4 + 2^1 + 2^0 = 64 + 16 + 2 + 1 = 83
└─┴─┴─┴─┴─┴─┴─┴─┘
                , / 2^0     ----> 83 / 1 = 83
```

Imaginons à présent que nous déplacions cette virgule virtuelle de trois éléments sur la gauche. En admettant que deux ingénieurs se mettent d'accord pour considérer ce nombre `0b01010011` avec une virgule fixe positionnée au quatrième bit, l'interprétation de cette

grandeur serait alors la valeur entière divisé par 8 (2^3). On parviens alors à exprimer une grandeur réelle comportant une epartie décimale :

```
┌─┬─┬─┬─┬─┬─┬─┬─┐
|0|1|0|1|0|0|1|1| = 2^6 + 2^4 + 2^1 + 2^0 = 64 + 16 + 2 + 1 = 83
└─┴─┴─┴─┴─┴─┴─┴─┘
          ,       / 2^3     ----> 83 / 8 = 10.375
```

Cependant, il manque une information. Un ordinateur, sans yeux et sans bon sens, est incapable sans information additionnelle d'interpréter correctement la position de la virgule puisque sa position n'est encodée nulle part. Et puisque la position de cette virgule est dans l'intervalle `[0..7]`, il serait possible d'utiliser trois bits supplémentaires à cette fin :

```
┌─┬─┬─┬─┬─┬─┬─┬─┐
|0|1|0|1|0|0|1|1| = 2^6 + 2^4 + 2^1 + 2^0 = 64 + 16 + 2 + 1 = 83
└─┴─┴─┴─┴─┴─┴─┴─┘
          ┌─┬─┬─┐
          |0|1|1| / 2^3     ----> 83 / 8 = 10.375
          └─┴─┴─┘
```

Cette solution est élégante mais demande a présent 11-bits contre 8-bits initialement. Un ordinateur n'étant doué que pour manipuler des paquets de bits souvent supérieurs à 8, il faudrait ici soit étendre inutilement le nombre de bits utilisés pour la position de la virgule à 8, soit tenter d'intégrer cette information, dans les 8-bits initiaux.

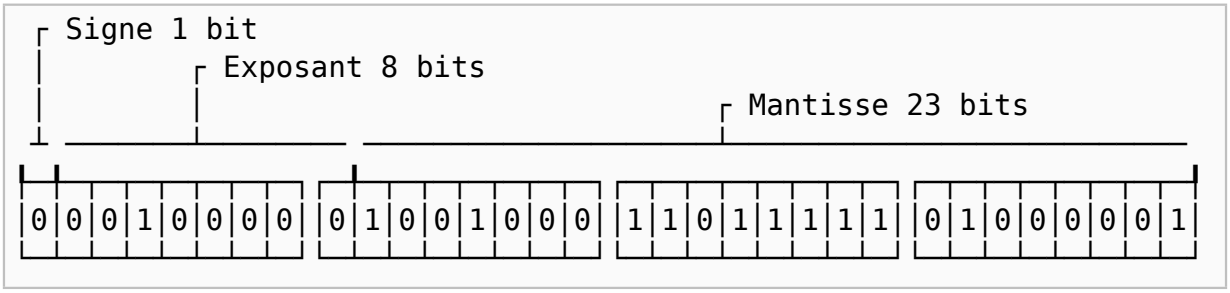
6.5.2 Virgule flottante

Imaginons alors que l'on sacrifie 3 bits sur les 8 pour encoder l'information de la position de la virgule. Appelons l'espace réservé pour positionner la virgule l' exposant et le reste de l'information la mantisse, qui en mathématique représente la partie décimale d'un logarithme (à ne pas confondre avec la mantis shrimp, une quille ou crevette mante boxeuse aux couleurs particulièrement chatoyantes).

```
  exp.  mantisse
┌─────┐┌───────┐
┌─┬─┬─┬─┬─┬─┬─┬─┐
|0|1|0|1|0|0|1|1| = 2^4 + 2^1 + 2^0 = 16 + 2 + 1 = 19
└─┴─┴─┴─┴─┴─┴─┴─┘
   └──────────────> / 2^1 ----> 19 / 2 = 9.5
```

Notre construction nous permet toujours d'exprimer des grandeurs réelles mais avec ce sacrifice, il n'est maintenant plus possible d'exprimer que les grandeurs comprises entre $1 \cdot 2^7 = 0.0078125$ et 63. Ce problème peut être aisément résolu en augmentant la profondeur mémoire à 16 ou 32-bits. Ajoutons par ailleurs que cette solution n'est pas à même d'exprimer des grandeurs négatives.

Dernière itération, choisissons d'étendre notre espace de stockage à ,4 octets. Réservons un bit de signe pour exprimer les grandeurs négatives, 8 bits pour l'exposant et 23 bits pour la mantisse :



Peu à peu, nous nous rapprochons du *Standard for Floating-Point Arithmetic* (IEEE 754). La formule de base est la suivante :

$$x = s \cdot b^e \sum_{k=1}^{p} f_k \cdot b^{-k},\ e_{\min} \leq e \leq e_{\max}$$

Avec :

- s Signe (± 1)
- b Base de l'exposant, un entier > 1.
- e Exposant, un entier entre $e_{\min}$ et $e_{\max}$
- p Précision, nombre de digits en base b de la mantisse
- f_k Entier non négatif plus petit que la base b.

Etant donné que les ordinateurs sont plus à l'aise à la manipulation d'entrées binaire, la base est 2 et la norme IEEE nomme ces nombres `binary16`, `binary32` ou `binary64`, selon le nombre de bits utilisé pour coder l'information. Les termes de *Single precision* ou *Double precision* sont aussi couramment utilisés.

Les formats supporté par un ordinateur ou qu'un microcontrôleur équipé d'une unité de calcul en virgule flottante (FPU pour *Floating point unit*) sont les suivants :

IEEE-754	Exposant	Mantisse	Signe
`binary32`	8 bits	23 bits	1 bit
`binary64`	11 bits	52 bits	1 bit

Prenons le temps de faire quelques observations.

- Une valeur encodée en virgule flottante sera toujours une approximation d'une grandeur réelle.
- La précision est d'autant plus grande que le nombre de bits de la mantisse est grand.
- La base ayant été fixée à 2, il est possible d'exprimer 1/1024 sans erreur de précision mais pas 1/1000.
- Un ordinateur qui n'est pas équipé d'une FPU sera beaucoup plus lent (10 à 100x) pour faire des calculs en virgule flottante.
- Bien que le standard **C99** définisse les types virgule flottante `float`, `double` et `long double`, ils ne definissent pas la précision avec lesquelles ces nombres sont exprimés car cela dépend de l'architecture du processeur utilisé.

6.5.3 Simple précision

Le type `float` aussi dit à précision simple utilise un espace de stockage de 32-bits organisé en 1 bit de signe, 8 bits pour l'exposant et 23 bits pour la mantisse. Les valeurs pouvant être exprimées sont de :

— $\pm$ inf lorsque l'exposant vaut `0xff`
— $(-1)^{\text{sign}} \cdot 2^{\text{exp}-127} \cdot 1.\text{significand}$
— 0 lorsque la mantisse vaut `0x00000`

La valeur de 1.0 est encodée :

$$\begin{aligned} 0\,01111111\,00000000000000000000000_2 &= 3\text{f}80\,0000_{16} \\ &= (-1)^0 \cdot 2^{127-127} \cdot \frac{(2^{23}+0)}{2^{23}} \\ &= 2^0 \cdot 1.0 = 1.0 \end{aligned}$$

La valeur maximale exprimable :

$$\begin{aligned} 0\,11111110\,11111111111111111111111_2 &= 7\text{f}7\text{f}\,\text{ffff}_{16} \\ &= (-1)^0 \cdot 2^{254-127} \cdot \frac{(2^{23}+838'607)}{2^{23}} \\ &2^{127} \cdot 1.9999998807 \\ &3.4028234664 \cdot 10^{38} \end{aligned}$$

La valeur de $-\pi$ (pi) est :

$$\begin{aligned} 1\,10000000\,10010010000111111011011_2 &= 4049\,0\text{fdb}_{16} \\ &= (-1)^1 \cdot 2^{128-127} \cdot \frac{(2^{23}+4'788'187)}{2^{23}} \\ &- 1 \cdot 2^1 \cdot 1.5707963 \\ &- 3.14159274101 \end{aligned}$$

Vient s'ajouter les valeurs particulières suivantes :

```
0 00000000 00000000000000000000000₂ ≡ 0000 0000₁₆ ≡ 0
0 11111111 00000000000000000000000₂ ≡ 7f80 0000₁₆ ≡ inf
1 11111111 00000000000000000000000₂ ≡ ff80 0000₁₆ ≡ -inf
```

6.5.4 Double précision

La double précision est similaire à la simple précision mais avec une mantisse à **52 bits** et **11 bits** d'exposants. Exercice

Donnez la valeur des expressions ci-dessous :

```
25. + 10. + 7. - 3.
5. / 2.
24. + 5. / 2.
```

(suite sur la page suivante)

(suite de la page précédente)

```
25. / 5. / 2.
25. / (5. / 2.)
2. * 13. % 7.
1.3E30 + 1.
```

6.6 Les caractères

Les caractères, ceux que vous voyez dans cet ouvrage, sont généralement représentés par des grandeurs exprimées sur 1 octet (8-bits) :

```
97 ≡ 0b1100001 ≡ 'a'
```

Historiquement, alors que les informations dans un ordinateur ne sont que des 1 et des 0, il a fallu établir une correspondance entre une grandeur binaire et le caractère associé. Un standard a été proposé en 1963 par l'ASA, l'*American Standards Association* aujourd'hui ANSI qui ne définissait alors que 63 caractères imprimables et comme la mémoire était en son temps très cher, un caractère n'était codé que sur 7 bits.

	000 0x00	001 0x01	002 0x02	003 0x03	004 0x04	005 0x05	006 0x06	007 0x07	010 0x08	011 0x09	012 0x0A	013 0x0B	014 0x0C	015 0x0D	016 0x0E	017 0x0F
000 0x00	NUL	SOM	EOA	EOM	EOT	WRU	RU	BEL	FEO	HT	LF	VT	FF	CR	SO	SI
020 0x10	DCO	DC1	DC2	DC3	DC4	ERR	SYN	LEM	SO	S1	S2	S3	S4	S5	S6	S7
040 0x20		!	"	#	$	%	&	'	(	)	*	+	,	-	.	/
060 0x30	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
0100 0x40	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
0120 0x50	P	Q	R	S	T	U	V	W	X	Y	Z	[	\	]	↑	←
0140 0x60																
0160 0x70													ACK		ESC	DEL

Fig. 6.1 – Table ASCII ASA X3.4 établie en 1963

Aujourd'hui la table ASCII de base défini 128 caractères qui n'incluent pas les caractères accentués.

Chaque pays et chaque langue utilise ses propres caractères et il a fallu trouver un moyen de satisfaire tout le monde. Il a été alors convenu d'encoder les caractères sur 8-bits au lieu de 7 et de profiter des 128 nouvelles positions pour ajouter les caractères manquants tels que les caractères accentués, le signe euro, la livre sterling et d'autres.

Le standard **ISO/IEC 8859** aussi appelé standard *Latin* défini 16 tables d'extension selon les besoins des pays. Les plus courantes en Europe occidentale sont les tables **ISO-8859-1** ou (**latin1**) et **ISO-8859-15** (**latin9**) :

	000 0x00	001 0x01	002 0x02	003 0x03	004 0x04	005 0x05	006 0x06	007 0x07	010 0x08	011 0x09	012 0x0A	013 0x0B	014 0x0C	015 0x0D	016 0x0E	017 0x0F
000 0x00	NUL \0	SOH	STX	ETX	EOT	ENQ	ACK	BEL \a	BS \b	TAB \t	LF \n	VT \v	FF \f	CR \r	SO	SI
020 0x10	DLE	DC1	DC2	DC3	DC4	NAK	SYN	ETB	CAN	EM	SUB	ESC	FS	GS	RS	US
040 0x20		!	"	#	$	%	&	'	(	)	*	+	,	-	.	/
060 0x30	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
0100 0x40	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
0120 0x50	P	Q	R	S	T	U	V	W	X	Y	Z	[	\ \\	]	^	_
0140 0x60	`	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
0160 0x70	p	q	r	s	t	u	v	w	x	y	z	{	|	}	~	DEL

Fig. 6.2 – Table ANSI INCITS 4-1986 (standard actuel)

	000 0x00	001 0x01	002 0x02	003 0x03	004 0x04	005 0x05	006 0x06	007 0x07	010 0x08	011 0x09	012 0x0A	013 0x0B	014 0x0C	015 0x0D	016 0x0E	017 0x0F
0x80																
0x90																
0xA0	NBSP	¡	¢	£	¤ €	¥	¦ Š	§	¨ š	©	ª	«	¬	SHY	®	¯
0xB0	°	±	²	³	´ Ž	µ	¶	·	¸ ž	¹	º	»	¼ Œ	½ œ	¾ Ÿ	¿
0xC0	À	Á	Â	Ã	Ä	Å	Æ	Ç	È	É	Ê	Ë	Ì	Í	Î	Ï
0xD0	Ð	Ñ	Ò	Ó	Ô	Õ	Ö	×	Ø	Ù	Ú	Û	Ü	Ý	Þ	ß
0xE0	à	á	â	ã	ä	å	æ	ç	è	é	ê	ë	ì	í	î	ï
0xF0	ð	ñ	ò	ó	ô	õ	ö	÷	ø	ù	ú	û	ü	ý	þ	ÿ

Fig. 6.3 – Table d'extension ISO-8859-1 (haut) et ISO-8859-15 (bas)

Ce standard a généré durant des décénies de grandes frustrations et de profondes incompréhensions chez les développeurs, et utilisateurs d'ordinateur. Ne vous est-il jamais arrivé d'ouvrir un fichier texte et de ne plus voir les accents convenablement ? C'est un problème typique d'encodage.

Pour tenter de remédier à ce standard incompatible entre les pays Microsoft à proposé un standard nommé Windows-1252 s'inspirant de ISO-8859-1. En voulant rassembler en proposant un standard plus général, Microsoft n'a contribué qu'à proposer un standard supplémentaire venant s'inscrire dans une liste déjà trop longue. Et l'histoire n'est pas terminée...

Avec l'arrivée d'internet et les échanges entre les arabes ,( ) les coréens ( ), les chinois avec le chinois simplifé ( ) et le chinois traditionel ( ), les japonais qui possèdent deux alphabets ainsi que des caractères chinois ( ), sans oublier l'ourdou ( ) pakistanais et tous ceux que l'on ne mentionnera pas, il a fallu bien plus que 256 caractères et quelques tables de correspondance. Ce présent ouvrage, ne pourrait d'ailleur par être écrit sans avoir pu résoudre, au préalable, ces problèmes d'encodage ; la preuve étant, vous parvenez à voir ces caractères qui ne vous sont pas familiers.

Un consensus planétaire a été atteint en 2008 avec l'adoption majoritaire du standard **Unicode** (*Universal Coded Character Set*) plus précisément nommé **UTF-8**.

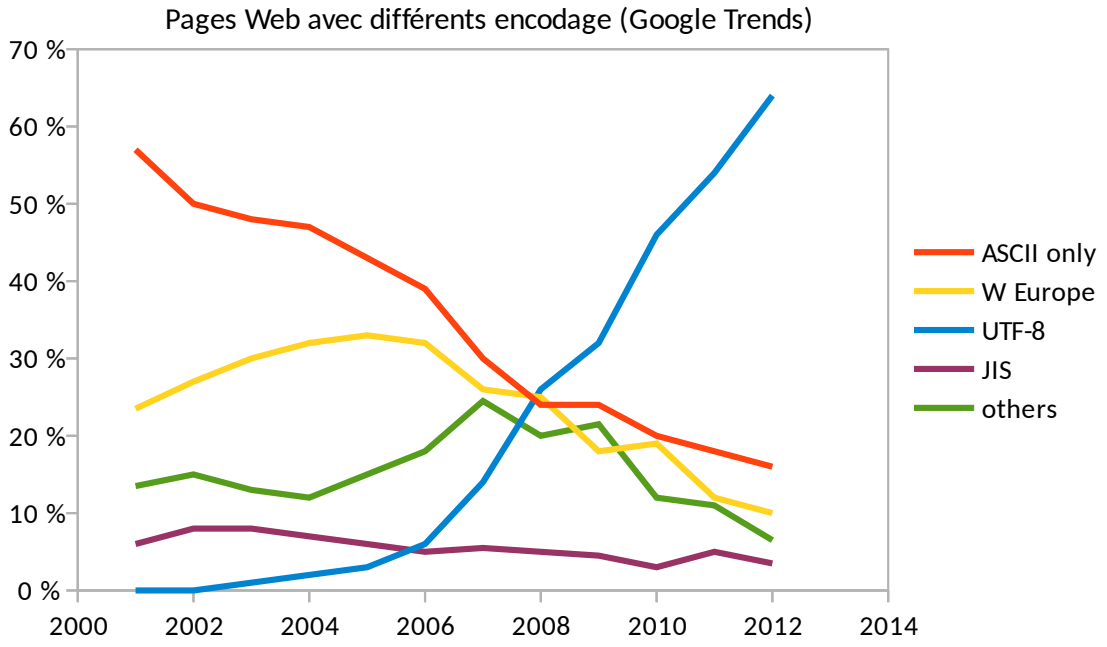


Fig. 6.4 – Tendances sur l'encodage des pages web en faveur de UTF-8 dès 2008

L'UTF-8 est capable d'encoder 11'112'064 caractères en utilisant de 1 à 4 octets. Ken Thompson, dont nous avons déjà parlé en *introduction* est à l'origine de ce standard. Par exemple le *devanagari* caractère ऩ utilisé en Sanskrit possède la dénomination unicode U+0939 et s'encode sur 3 octets : `0xE0 0xA4 0xB9`

En programmation C, un caractère `char` ne peut exprimer sans ambiguité que les 128 caractères de la table ASCII standard et selon les conventions locales, les 128 caractères

d'extension.

Voici par exemple comment déclarer une variable contenant le caractère dollar :

```
char c = '$';
```

Attention donc au caractère `'3'` qui correspond à la grandeur hexadécimale `0x33` :

```
#include <stdio.h>

int main(void) {
    char c = '3';

    printf("Le caractère %c vaut 0x%x en hexadécimal ou %d en décimal.\n",
    c, c, c);
    return 0;
}
```

6.7 Chaîne de caractères

Une chaîne de caractères est simplement la suite contigue de plusieurs caractère dans une zone mémoire donnée. Afin de savoir lorsque cette chaîne se termine, le standard impose que le dernier caractère d'une chaîne soit `NUL` ou `\0`.

La chaîne de caractère `Hello` sera en mémoire stockée en utilisant les codes ASCII suivants.

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

```
  H   E   L   L   O  \0
┌───┬───┬───┬───┬───┬───┐
│ 72│101│108│108│111│ 0 │
└───┴───┴───┴───┴───┴───┘

0x00 01001000
0x01 01100101
0x02 01101100
0x03 01101100
0x04 01101111
0x05 00000000
```

Exercice

Indiquez si les constantes littérales suivantes sont valides ou invalides.

1. `'a'`
2. `'A'`
3. `'ab'`
4. `'\x41'`

5. `'\041'`
6. `'\0x41'`
7. `'\n'`
8. `'\w'`
9. `'\t'`
10. `'\xp2'`
11. `"abcdef"`
12. `"\abc\ndef"`
13. `"\'\"\\"`
14. `"Hello \world !\n"`

Exercice

Pour les instructions ci-dessous, indiquer quel est l'affichage obtenu.

```
char a = 'a';
short sh1 = 5;
float f1 = 7.0f;
int i1 = 7, i2 = 'a';
```

1. `printf("Next char: %c.\n", a + 1);`
2. `printf("Char: %3c.\n", a);`
3. `printf("Char: %-3c.\n", a);`
4. `printf("Chars: \n-%c.\n-%c.\n", a, 'z' - 1);`
5. `printf("Sum: %i\n", i1 + i2 - a);`
6. `printf("Taux d'erreur\t%i %%\n", i1);`
7. `printf("Quel charabia horrible:\\\a\a\a%g\b\a%%\a\\\n", f1);`
8. `printf("Inventaire: %i4 pieces\n", i1);`
9. `printf("Inventory: %i %s\n", i1, "pieces");`
10. `printf("Inventaire: %4i pieces\n", i1);`
11. `printf("Inventaire: %-4i pieces\n", i1);`
12. `printf("Mixed sum: %f\n", sh1 + i1 + f1);`
13. `printf("Tension: %5.2f mV\n", f1);`
14. `printf("Tension: %5.2e mV\n", f1);`
15. `printf("Code: %X\n", 12);`
16. `printf("Code: %x\n", 12);`
17. `printf("Code: %o\n", 12);`
18. `printf("Value: %i\n", -1);`
19. `printf("Value: %hi\n", 65535u);`
20. `printf("Value: %hu\n", -1);`

6.8 Les booléens

Un booléen est un type de donnée à deux états consensuellement nommés *vrai* (`true`) et *faux* (`false`) et destiné à représenter les états en logique booléenne (Nom venant de George Boole fondateur de l'algèbre éponyme).

La convention est d'utiliser `1` pour mémoriser un état vrai, et `0` pour un état faux, c'est d'ailleurs de cette manière que les booléens sont encodés en C.

Les booléens ont étés introduits formellement en C avec **C99** et nécessitent l'inclusion du fichier d'en-tête `stdbool.h`. Avant cela le type boolean était `_Bool` et définir les états vrais et faux étaient à la charge du développeur.

```c
#include <stdbool.h>

bool is_enabled = false;
bool has_tail = true;
```

Afin de faciliter la lecture du code, il est courant de préfixer les variables booléenes avec les prefixes `is_` ou `has_`.

A titre d'exemple, si l'on souhaite stocker le genre d'un individu (male, ou femelle), on pourrait utiliser la variable `is_male`.

6.9 Énumérations

Ce style d'écriture permet de définir un type de données contenant un nombre fini de valeurs. Ces valeurs sont nommées textuellement et définies numériquement dans le type énuméré.

```c
enum ColorCode {
    COLOR_BLACK, // Vaut zéro par défaut
    COLOR_BROWN,
    COLOR_RED,
    COLOR_ORANGE,
    COLOR_YELLOW,
    COLOR_GREEN,
    COLOR_BLUE,
    COLOR_PURPLE,
    COLOR_GRAY,
    COLOR_WHITE
};
```

Le type d'une énumération est apparenté à un entier `int`. Sans autre précisions, la première valeur vaut 0, la suivante 1, etc.

Il est possible de forcer les valeurs de la manière suivante :

```
typedef enum country_codes {
    CODE_SWITZERLAND=41,
    CODE_FRANCE=33,
    CODE_US=1
} CountryCodes;
```

ou encore :

```
typedef enum country_codes {
    CODE_SWITZERLAND=41,
    CODE_BELGIUM=32
    CODE_FRANCE, // Sera 33...
    CODE_SPAIN, // Sera 34...
    CODE_US=1
} CountryCodes;
```

Pour ne pas confondre un type énuméré avec une variable, on utilise souvent la convention d'une notation en capitales. Pour éviter des éventuelles collisions avec d'autres types, un préfixe est souvent ajouté.

L'utilisation d'un type énuméré peut être la suivante :

```
void call(enum country_codes code) {
    switch(code) {
    case CODE_SWITZERLAND :
        printf("Calling Switzerland, please wait...\n");
        break;
    case CODE_BELGIUM :
        printf("Calling Belgium, please wait...\n");
        break;
    case CODE_FRANCE :
        printf("Calling France, please wait...\n");
        break;
    default :
        printf("No calls to this country are allowed yet!\n");
    }
}
```

6.10 Type incomplet

Un type incomplet est un qualificatif de type de donnée décrivant un objet dont sa taille en mémoire n'est pas connue.

6.11 Type vide (*void*)

Le type `void` est particulier. Il s'agit d'un type dit **incomplet** car la taille de l'objet qu'il représente en mémoire n'est pas connue. Il est utilisé comme type de retour pour les fonctions qui ne retournent rien :

```
void shout() {
    printf("Hey!\n");
}
```

Il peut être également utilisé comme type générique comme la fonction de copie mémoire `memcpy`

```
void *memcpy(void * restrict dest, const void * restrict src, size_
↪t n);
```

Le mot clé `void` ne peut être utilisé que dans les contextes suivants :

- Comme paramètre unique d'une fonction, indiquant que cette fonction n'a pas de paramètres `int main(void)`
- Comme type de retour pour une fonction indiquant que cette fonction ne retourne rien `void display(char c)`
- Comme pointeur dont le type de destination n'est pas spécifié `void* ptr`

6.12 Promotion implicite

Généralement le type `int` est de la même largeur que le bus mémoire de donnée d'un ordinateur. C'est à dire que c'est souvent, le type le plus optimisé pour véhiculer de l'information au sein du processeur. Les *registres* du processeur, autrement dit ses casiers mémoire, sont au moins assez grand pour contenir un `int`.

Aussi, la plupart des types de taille inférieure à `int` sont automatiquement et implicitement promu en `int`. Le résultat de `a + b` lorsque `a` et `b` sont des `char` sera automatiquement un `int`.

char	⇒	int
short	⇒	int
int	⇒	long
long	⇒	float
float	⇒	double

Notez qu'il n'y a pas de promotion numérique vers le type *short*. On passe directement à un type *int*. Exercice

Soit les instructions suivantes :

```
int n = 10;
int p = 7;
float x = 2.5;
```

Donnez le type et la valeur des expressions suivantes :

1. `x + n % p`
2. `x + p / n`
3. `(x + p) / n`
4. `.5 * n`
5. `.5 * (float)n`
6. `(int).5 * n`
7. `(n + 1) / n`
8. `(n + 1.0) / n`

Exercice

Représentez les promotions numériques qui surviennent lors de l'évaluation des expressions ci-dessous :

```
char c;
short sh;
int i;
float f;
double d;
```

1. `c * sh - f / i + d;`
2. `c * (sh – f) / i + d;`
3. `c * sh - f - i + d;`
4. `c + sh * f / i + d;`

6.12.1 Effets du transtypage

Le changement de type forcé (transtypage) entre des variables de différents types engendre des effets de bord qu'il faut connaître. Lors d'un changement de type vers un type dont le pouvoir de représentation est plus important, il n'y a pas de problème. A l'inverse, on peut rencontrer des erreurs sur la précision ou une modification radicale de la valeur représentée !

Transtypage d'un entier en réel

La conversion d'un entier (signé ou non) en réel (*double* ou *float*) n'a pas d'effet particulier. Le type

```
long l=3;
double d=(double)l; // valeur : 3 => OK
```

A l'exécution, la valeur de d sera la même que l.

Transtypage d'un réel en entier

La conversion d'un nombre réel (*double* ou *float*) en entier (signé) doit être étudié pour éviter tout problème. Le type entier doit être capable de recevoir la valeur (attention aux valeurs maxi).

```
double d=3.9;
long l=(long)d; // valeur : 3 => perte de précision
```

A l'exécution, la valeur de l sera la partie entière de d. Il n'y a pas d'arrondi.

```
double d=0x12345678;
short sh=(short)d; // valeur : 0x5678 => changement de valeur
```

La variable sh (*short* sur 16 bit) ne peut contenir la valeur réelle. Lors du transtypage, il y a modification de la valeur ce qui conduit à des erreurs de calculs par la suite.

```
double d=-123;
unsigned short sh=(unsigned short)d; // valeur : 65413 =>␣
↪changement de valeur
```

L'utilisation d'un type non signé pour convertir un nombre réel conduit également à une modification de la valeur numérique.

Transtypage d'un double en float

La conversion d'un nombre réel de type *double* en réel de type *float* pose un problème de précision de calcul.

```
double d=0.1111111111111111;
float f=(float)d; // valeur : 0.1111111119389533 => perte de␣
↪précision
```

A l'exécution, il y a une perte de précision lors de la conversion ce qui peut, lors d'un calcul itératif induire des erreurs de calcul. Exercice

On considère les déclarations suivantes :

```
float x;
short i;
unsigned short j;
long k;
unsigned long l;
```

Identifiez les expressions ci-dessous dont le résultat n'est pas mathématiquement correct.

```
x = 1e6;
i = x;
j = -20;
k = x;
l = k;
k = -20;
l = k;
```

```
x = 1e6;
i = x;    // Incorrect, i peut-être limité à -32767..+32767 (C99 §5.
↪2.4.2.1)
j = -20;  // Incorrect, valeur signée dans un conteneur non signé
k = x;
l = k;
k = -20;
l = k;    // Incorrect, valeur signée dans un conteneur non signé
```

Exercice

Que valent les valeurs de `p`, `x` et `n` :

```
float x;
int n, p;

p = 2;
x = (float)15 / p;
n = x + 1.1;
```

p 2 x = 7.5 n = 8 Exercice

Soit les déclarations suivantes :

```
float x, y;
bool condition;
```

Réécrire l'expression ci-dessous en mettant des parenthèses montrant l'ordre des opérations :

```
condition = x >= 0 && x <= 20 && y > x || y == 50 && x == 2 || y ==␣
↪60;
```

Donner la valeur de `condition` évaluée avec les valeurs suivantes de `x` et `y` :

1. `x = -1.0; y = 60.;`

2. `x = 0; y = 1.;`
3. `x = 19.0; y = 1.0;`
4. `x = 0.0; y = 50.0;`
5. `x = 2.0; y = 50.0;`
6. `x = -10.0; y = 60.0;`

```
condition = (
    (x >= 0) && (x <= 20) && (y > x))
    ||
    ((y == 50) && (x == 2))
    ||
    (y == 60)
);
```

1. `true`
2. `true`
3. `false`
4. `true`
5. `true`
6. `true`

Exercice

Vous participez à une revue de code et tomber sur quelques perles laissées par quelques collègues. Comment proposeriez-vous de corriger ces écritures ? Le code est écrit pour un modèle de donnée **LLP64**.

Pour chaque exemple, donner la valeur des variables après exécution du code.

1. ```
   unsigned short i = 32767;
   i++;
   ```

2. ```
   short i = 32767;
   i++;
   ```

3. ```
   short i = 0;
   i = i--;
   i = --i;
   i = i--;
   ```

Exercice

Considérons les déclarations suivantes :
   ```

```
char c = 3;
short s = 7;
int i = 3;
long l = 4;
float f = 3.3;
double d = 7.7;
```

Que vaut le type et la valeur des expressions suivantes ?

1. `c / 2`
2. `sh + c / 10`
3. `lg + i / 2.0`
4. `d + f`
5. `(int)d + f`
6. `(int)d + lg`
7. `c << 2`
8. `sh & 0xF0`
9. `sh && 0xF0`
10. `sh == i + lg`
11. `d + f == sh + lg`

Exercice

Que vaut `x` ?

```
float x = 10000000. + 0.1;
```

Le format float est stocké sur 32-bits avec 23-bits de mantisse et 8-bits d'exposants. Sa précision est donc limitée à environ 6 décimales. Pour représenter 10'000'000.1 il faut plus que 6 décimales et l'addition est donc caduc :

```
#include <stdio.h>

int main(void) {
    float x = 10000000. + 0.1;
    printf("%f\n", x);
}
```

```
$ ./a.out
10000000.000000
```

Exercice

Pour chaque entrée suivante, indiquez le nom et le type des variabels que vous utiliseriez pour représenter les données dans ce programme :

1. Gestion d'un parking : nombre de voitures présentes
2. **Station météo**
    1. Température moyenne de la journée

   2. Nombre de valeurs utilisées pour la moyenne
3. Montant disponible sur un compte en banque
4. Programme de calcul de d'énergie produite dans une centrale nucléaire
5. Programme de conversion décimal, hexadécimal, binaire
6. Produit scalaire de deux vecteurs plans
7. Nombre d'impulsions reçues par un capteur de position incrémental

Exercice

On considère un disque, divisé en 12 secteurs angulaires égaux, numérotés de 0 à 11. On mesure l'angle de rotation du disque en degrés, sous la forme d'un nombre entier non signé. Une flèche fixe désigne un secteur. Entre 0 et 29 °, le secteur désigné est le n° 0, entre 30 ° et 59 °, c'est le secteur 1, ...

Donnez une expression arithmétique permettant, en fonction d'un angle donné, d'indiquer que est le secteur du disque se trouvent devant la flèche. Note : l'angle de rotation peut être supérieur à 360 °. Vérifiez cette expression avec les angles de 0, 15, 29, 30, 59, 60, 360, 389, 390 degrés.

Ecrivez un programme demandant l'angle et affichant le numéro de secteur correspondant.
Exercice

Il est prouvé mathématiquement que la somme des entiers strictement positifs pris dans l'ordre croissant peut être exprimé comme :

$$\sum_{k=1}^{n} k = \frac{n(n+1)}{2}$$

Un grand mathématicien Srinivasa Ramanujan (En tamoul :) à démontré que ce la somme à l'infini donne :

$$\sum_{k=1}^{\inf} k = -\frac{1}{12}$$

Vous ne le croyez pas et décider d'utiliser le super-ordinateur Pensées Profondes pour faire ce calcul. Comme vous n'avez pas accès à cet ordinateur pour l'instant (et probablement vos enfants n'auront pas accès à cet ordinateur non plus), écrivez un programme simple pour tester votre algorithme et prenant en paramètre la valeur `n` à laquelle s'arrêter.

Tester ensuite votre programme avec des valeurs de plus en plus grandes et analyser les performances avec le programme `time` :

```
$ time ./a.out 1000000000
500000000500000000

real    0m0.180s
user    0m0.172s
sys     0m0.016s
```

A partir de quelle valeur, le temps de calcul devient significativement palpable ?

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

int main(int argc, char *argv[]) {
    long long n = atoi(argv[1]);
    long long sum = 0;
    for(size_t i = 0; i < n; i++, sum += i);
    printf("%lld\n", sum);
}
```

Exercice

La société japonaise Nakainoeil développe des systèmes de vision industriels pour l'inspection de pièces dans une ligne d'assemblage. Le programme du système de vision comporte les variables internes suivantes :

```c
uint32_t inspected_parts, bad_parts;
float percentage_good_parts;
```

A un moment du programme, on peut lire :

```c
percentage_good_parts = (inspected_parts - bad_parts) / inspected_
↪parts;
```

Sachant que `inspected_parts = 2000` et `bad_parts = 200` :

1. Quel résultat le développeur s'attend-il à obtenir ?
2. Qu'obtient-il en pratique ?
3. Pourquoi ?
4. Corrigez les éventuelles erreurs

1. Le développeur s'attend à obtenir le pourcentage de bonne pièces avec plusieurs décimales après la virgule.
2. En pratique, il obtient un entier, c'est à dire toujours 0.
3. **La promotion implicite des entiers peut être découpée comme suit :**

   ```c
   (uint32_t)numerator = (uint32_t)inspected_parts - (uint32_
   ↪t)bad_parts;
   (uint32_t)percentage = (uint32_t)numerator / (uint32_
   ↪t)inspected_parts;
   (float)percentage_good_parts = (uint32_t)percentage;
   ```

   La division est donc appliquée à des entiers et non des flottnts.
4. **Une possible correction consiste à forcer le type d'un des membres de la division :**

Exercice

Durant la guerre du golfe le 25 février 1991, une batterie de missile américaine à Dharan en arabie saoudite à échoué à intercepter un missile iraquien Scud. Cet échec tua 28 soldats

américains et en blessa 100 autres. L'erreur sera imputée à un problème de type de donnée sera longuement discutée dans le rapport **GAO/OMTEC-92-26** du commandement général.

Un registre 24-bit est utilisé pour le stockage du temps écoulé depuis le démarrage du logiciel de contrôle indiquant le temps en dixieème de secondes. Dès lors il a fallait multiplier ce temps par 1/10 pour obtenir le temps en seconde. La valeur 1/10 était tronquée à la 24 :sup :*ième* décimale après la virguleDes erreurs d'arrondi sont apparue menant à un décalage de près de 1 seconde après 100 heures de fonction. Or, cette erreur d'une seconde s'est traduit par 600 mètres d'erreur lors de la tentative d'interception.

Le stockage de la valeur 0.1 est donné par :

$$0.1_{10} \approx \lfloor 0.1_{10} \cdot 2^{23} \rfloor = 11001100110011001100_2 \approx 0.09999990463256836$$

Un registre contient donc le nombre d'heures écoulées exprimée en dixième de seconde soit pour 100 heures :

$$100 \cdot 60 \cdot 60 \cdot 10 = 3'600'000$$

En termes de virgule fixe, la première valeur est exprimée en Q1.23 tandis que la seconde en Q0.24. Multiplier les deux valeurs entre elles donne `Q1.23 x Q0.24 = Q1.47` le résultat est donc exprimé sur 48 bits. Il faut donc diviser le résultat du calcul par 2^{47} pour obtenir le nombre de secondes écoulées depuis le début la mise sous tension du système.

Quel est l'erreur en seconde cumulée sur les 100 heures de fonctionnement ?

Chapitre 7

Structures de contrôle

Les structures de contrôle appartiennent aux langages de programmation dits structurés. Elles permettent de modifier l'ordre des opérations lors de l'exécution du code. Il y a trois catégories de structures de contrôle en C :

1. Les embranchements (`branching`)
2. Les boucles (`loops`)
3. Les sauts (`jumps`)

Ces structures de contrôles sont toujours composées de :

— Séquences
— Sélections
— Répétitions
— Appels de fonctions

7.1 Séquences

En C, chaque instruction est séparée de la suivante par un point virgule `;` (U+003B) :

```
k = 8; k *= 2;
```

Une séquence est une suite d'instructions regroupées en un bloc matérialisé par des accolades `{}` :

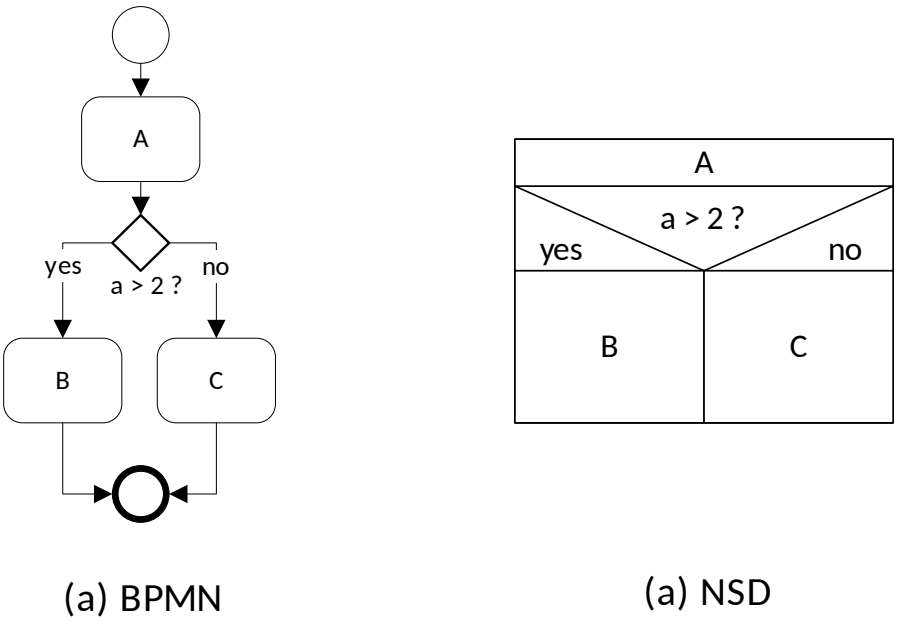
```
{
    double pi = 3.14;
    area = pi * radius * radius;
}
```

Note : N'allez pas confondre le point virgule `;` (U+003B) avec le `;` (U+037E), le point d'interrogation grec ( ). Certains farceurs aiment à le remplacer dans le code de camarades ce qui génère naturellement des erreurs de compilation.

7.2 Les embranchements

Les embranchements sont des instructions de prise de décision. Une prise de décision peut être binaire, lorsqu'il y a un choix *vrai* et un choix *faux*, ou multiple lorsque la condition est scalaire. En C il y en a trois type d'embranchements :

1. `if`, `if else`
2. `switch`
3. L'instruction ternaire



(a) BPMN (a) NSD

Fig. 7.1 – Exemples d'embranchements dans les diagrammes de flux BPMN (Business Process Modeleing Notation) et NSD (Nassi-Shneiderman)

Les embranchements s'appuient sur les séquences :

```c
if (value % 2)
{
    printf("odd\n");
}
else
{
    printf("even\n");
}
```

7.2.1 if..else

Le mot clé `if` est toujours suivi d'une condition entre parenthèses qui est évaluée. Si la condition est vraie, le premier bloc est exécuté, sinon, le second bloc situé après le `else` est exécuté.

Les enchaînements possibles sont :
- `if`
- `if` + `else`
- `if` + `else if`
- `if` + `else if` + `else if` + ...
- `if` + `else if` + `else`

Une condition n'est pas nécessairement unique mais peut-être la concaténation logique de plusieurs conditions séparées :

```
if((0 < x && x < 10) || (100 < x && x < 110) || (200 < x && x <␣
↪210))
{
    printf("La valeur %d est valide", x);
    is_valid = true;
}
else
{
    printf("La valeur %d n'est pas valide", x);
    is_valid = false;
}
```

Remarquons qu'au passage cet exemple peut être simplifié :

```
is_valid = (0 < x && x < 10) || (100 < x && x < 110) || (200 < x &&␣
↪x < 210);

if (is_valid)
{
    printf("La valeur %d est valide", x);
}
else
{
    printf("La valeur %d n'est pas valide", x);
}
```

Notons quelques erreurs courantes :
- Il est courant de placer un point virgule derrière un `if`. Le point virgule correspondant à une instruction vide, c'est cette instruction qui sera exécutée si la condition du test est vraie.

  ```
  if (z == 0);
  printf("z est nul"); // ALWAYS executed
  ```

- Le test de la valeur d'une variable s'écrit avec l'opérateur d'égalité `==` et non l'opérateur d'affectation `=`. Ici, l'évaluation de la condition vaut la valeur affectée

à la variable.

```
if (z = 0)                // set z to zero !!
    printf("z est nul"); // NEVER executed
```

— L'oubli des accolades pour déclarer un bloc d'instructions

```
if (z == 0)
    printf("z est nul");
    is_valid = false;
else
    printf("OK");
```

L'instruction `if` permet également l'embranchement multiple, lorsque les conditions ne peuvent pas être regroupées :

```
if (value % 2)
{
    printf("La valeur est impaire.");
}
else if (value > 500)
{
    printf("La valeur est paire et supérieure à 500.");
}
else if (!(value % 5))
{
    printf("La valeur est paire, inférieur à 500 et divisible par 5.");
}
else
{
    printf("La valeur ne satisfait aucune condition établies.");
}
```

Exercice

Comment se comporte l'exemple suivant :

```
if (!(i < 8) && !(i > 8))
    printf("i is %d\n", i);
```

Exercice

Compte tenu de la déclaration `int i = 8;`, indiquer pour chaque expressions si elles impriment ou non `i vaut 8` :

1.
```
if (!(i < 8) && !(i > 8)) then
    printf("i vaut 8\n");
```

2.
```c
if (!(i < 8) && !(i > 8))
    printf("i vaut 8");
    printf("\n");
```

3.
```c
if !(i < 8) && !(i > 8)
    printf("i vaut 8\n");
```

4.
```c
if (!(i < 8) && !(i > 8))
    printf("i vaut 8\n");
```

5.
```c
if (i = 8) printf("i vaut 8\n");
```

6.
```c
if (i & (1 << 3)) printf("i vaut 8\n");
```

7.
```c
if (i ^ 8) printf("i vaut 8\n");
```

8.
```c
if (i - 8) printf("i vaut 8\n");
```

9.
```c
if (i == 1 << 3) printf ("i vaut 8\n");
```

10.
```c
if (!((i < 8) || (i > 8)))
    printf("i vaut 8\n");
```

7.2.2 `switch`

L'embranchement multiple, lorsque la condition n'est pas binaire mais scalaire, l'instruction `switch` peut-être utilisée :

```c
switch (defcon)
{
    case 1 :
        printf("Guerre nucléaire imminente");
        break;
    case 2 :
        printf("Prochaine étape, guerre nucléaire");
        break;
    case 3 :
        printf("Accroissement de la préparation des forces");
        break;
    case 4 :

↪printf("Mesures de sécurité renforcées et renseignements accrus");
        break;
    case 5 :
```

(suite sur la page suivante)

(suite de la page précédente)

```
        printf("Rien à signaler, temps de paix");
        break;
    default :
        printf("ERREUR: Niveau d'alerte DEFCON invalide");
}
```

La valeur par défaut `default` est optionnelle mais recommandée pour traiter les cas d'erreurs possibles.

La structure d'un `switch` est composée d'une condition `switch (condition)` suivie d'une séquence `{}`. Les instructions de cas `case 42:` sont appelés *labels*. L'instruction `break` termine l'exécution de la séquence `switch`.

Les labels peuvent être chaînés sans instructions intermédiaires ni `break` :

```
switch (coffee)
{
    case IRISH_COFFEE :
        add_whisky();

    case CAPPUCCINO :
    case MACCHIATO :
        add_milk();

    case ESPRESSO :
    case AMERICANO :
        add_coffee();
        break;

    default :
        printf("ERREUR 418: Type de café inconnu");
}
```

Notons quelques observations :

— La structure `switch` bien qu'elle puisse toujours être remplacée par une structure `if..else if` est généralement plus élégante et plus lisible. Elle évite par ailleurs de répéter la condition plusieurs fois (c.f. Section 24.2.1).

— Le compilateur est mieux à même d'optimiser un choix multiple lorsque les valeurs scalaires de la condition triées se suivent directement e.g. `{12, 13, 14, 15}`.

— L'ordre des cas d'un `switch` n'a pas d'importance, le compilateur peut même choisir de réordonner les cas pour optimiser l'exécution.

7.3 Les boucles

Une boucle est une structure itérative permettant de répéter l'exécution d'une séquence. En C il existe trois types de boucles :

- `for`
- `while`
- `do .. while`

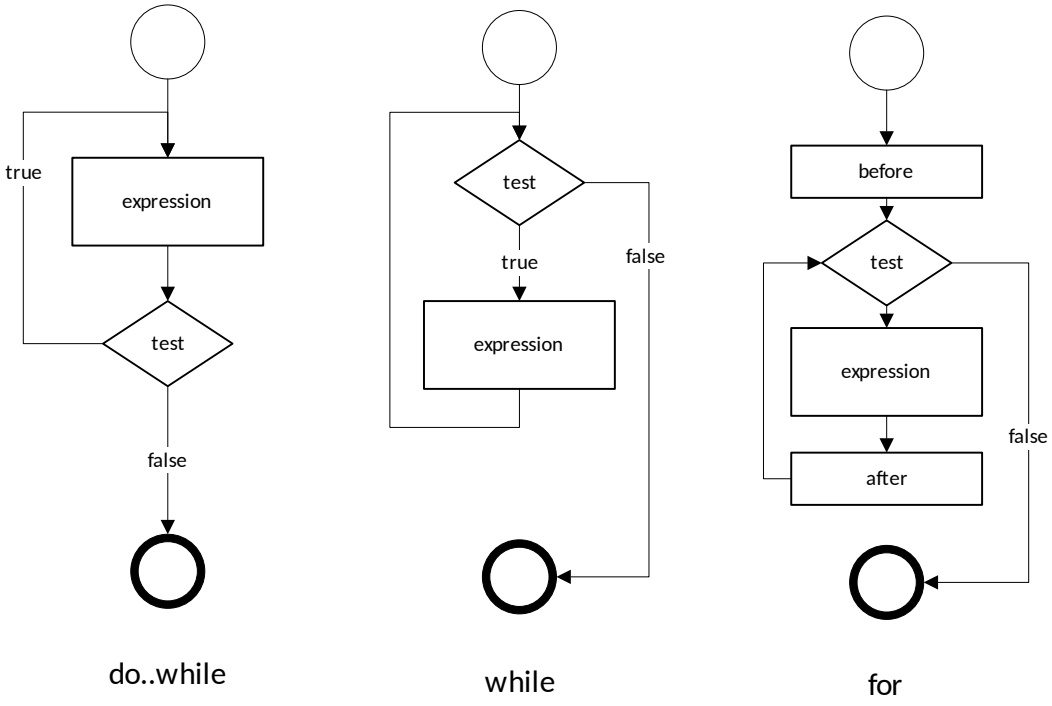


Fig. 7.2 – Aperçu des trois structure de boucles

7.3.1 while

La structure `while` répète une séquence **tant que** la condition est vraie.

Dans l'exemple suivant tant que le poids d'un objet déposé sur une balance est inférieur à une valeur constante, une masse est ajoutée et le système patiente avant stabilisation.

```
while (get_weight() < 420 /* newtons */)
{
    add_one_kg();
    wait(5 /* seconds */);
}
```

Séquentiellement une boucle `while` teste la condition, puis exécute la séquence associée.
Exercice

Comment se comportent ces programmes :

1. `size_t i=0;while(i<11){i+=2;printf("%i\n",i);}`
2. `i=11;while(i--){printf("%i\n",i--);}`
3. `i=12;while(i--){printf("%i\n",--i);}`
4. `i = 1;while ( i <= 5 ){ printf ( "%i\n", 2 * i++ );}`
5. `i = 1; while ( i != 9 ) { printf ( "%i\n", i = i + 2 ); }`
6. `i = 1; while ( i < 9 ) { printf ( "%i\n", i += 2 ); break; }`
7. `i = 0; while ( i < 10 ) { continue; printf ( "%i\n", i += 2 ); }`

7.3.2 do..while

De temps en temps il est nécessaire de tester la condition à la sortie de la séquence et non à l'entrée. La boucle `do`...`while` permet justement ceci :

```
size_t i = 10;

do {
    printf("Veuillez attendre encore %d seconde(s)\r\n", i);
    i -= 1;
} while (i);
```

Contrairement à la boucle `while`, la séquence est ici exécutée **au moins une fois**.

7.3.3 for

La boucle `for` est un `while` amélioré qui permet en une ligne de résumer les conditions de la boucle :

```
for (/* expression 1 */; /* expression 2 */; /* expression 3 */)
{
    /* séquence */
}
```

Expression 1 Exécutée une seule fois à l'entrée dans la boucle, c'est l'expression d'initialisation permettant par exemple de déclarer une variable et de l'initialiser à une valeur particulière.

Expression 2 Condition de validité (ou de maintient de la boucle). Tant que la condition est vraie, la boucle est exécutée.

Expression 3 Action de fin de tour. A la fin de l'exécution de la séquence, cette action est exécutée avant le tour suivant. Cette action permet par exemple d'incrémenter une variable.

Voici comment répéter 10x un block de code :

```c
for (size_t i = 0; i < 10; i++)
{
    something();
}
```

Notons que les portions de `for` sont optionnels et que la structure suivante est strictement identique à la boucle `while` :

```c
for (; get_weight() < 420 ;)
{
    /* ... */
}
```

Exercice

Comment est-ce que ces expressions se comportent-elles ?

```c
int i, k;
```

1. `for (i = 'a'; i < 'd'; printf ("%i\n", ++i));`
2. `for (i = 'a'; i < 'd'; printf ("%c\n", ++i));`
3. `for (i = 'a'; i++ < 'd'; printf ("%c\n", i ));`
4. `for (i = 'a'; i <= 'a' + 25; printf ("%c\n", i++ ));`
5. `for (i = 1 / 3; i ; printf("%i\n", i++ ));`
6. `for (i = 0; i != 1 ; printf("%i\n", i += 1 / 3 ));`
7. `for (i = 12, k = 1; k++ < 5 ; printf("%i\n", i-- ));`
8. `for (i = 12, k = 1; k++ < 5 ; k++, printf("%i\n", i-- ));`

Exercice

Identifier les deux erreurs dans ce code suivant :

```c
for (size_t = 100; i >= 0; --i)
    printf("%d\n", i);
```

Exercice

Écrivez un programme affichant les entiers de 1 à 100 en employant :

1. Une boucle `for`
2. Une boucle `while`
3. Une boucle `do..while`

Quel est la structure de contrôle la plus adaptée à cette situation ? Exercice

Expliquez quelle est la fonctionnalité globale du programme ci-dessous :

```
int main(void) {
    for(size_t i = 0, j = 0; i * i < 1000; i++, j++, j %= 26,␣
↪printf("\n"))
        printf("%c", 'a' + (char)j);
}
```

Proposer une meilleure implémentation de ce programme.

7.3.4 Boucles infinies

Une boucle infinie n'est jamais terminée. On rencontre souvent ce type de boucle dans ce que l'on appelle à tort *La boucle principale* aussi nommée run loop. Lorsqu'un programme est exécuté *bare-metal*, c'est à dire directement à même le microcontrôleur et sans système d'exploitation, il est fréquent d'y trouver une fonction `main` telle que :

```
void main_loop()
{
    // Boucle principale
}

int main(void)
{
    for (;;)
    {
        main_loop();
    }
}
```

Il y a différentes variantes de boucles infinies :

```
for (;;) { }

while (true) { }

do { } while (true);
```

Notions que l'expression `while (1)` que l'on rencontre fréquemment dans des exemples est fausse syntaxiquement. Une condition de validité devrait être un booléen, soit vrai, soit faux. Or, la valeur scalaire `1` devrait préalablement être transformée en une valeur booléenne. Il est donc plus juste d'écrire `while (1 == 1)` ou simplement `while (true)`.

On préférera néanmoins l'écriture `for (;;)` qui ne fait pas intervenir de conditions extérieure car en effet, avant **C99** définir la valeur `true` était à la charge du développeur et on pourrait s'imaginer cette plaisanterie de mauvais goût :

```
_Bool true = 0;

while (true) { /* ... */ }
```

Lorsque l'on a besoin d'une boucle infinie il est généralement préférable de permettre au programme de se terminer correctement lorsqu'il est interrompu par le signal **SIGINT** (c.f. Section 8.1.4). On rajoute alors une condition de sortie à la boucle principale :

```c
#include <stdlib.h>
#include <signal.h>
#include <stdbool.h>

static volatile bool is_running = true;

void sigint_handler(int dummy)
{
    is_running = false;
}

int main(void)
{
    signal(SIGINT, sigint_handler);

    while (is_running)
    {
       /* ... */
    }

    return EXIT_SUCCESS;
}
```

7.4 Les sauts

Il existe 4 instructions en C permettant de contrôler le déroulement de l'exécution d'un programme. Elles déclenchent un saut inconditionnel vers un autre endroit du programme.

- **`break` interrompt la structure de contrôle en cours. Elle est valide pour :**
  - `while`
  - `do`...“while“
  - `switch`
- `continue` : saute un tour d'exécution dans une boucle
- `goto` : interrompt l'exécution et saute à un label situé ailleurs dans la fonction
- `return`

7.4.1 `goto`

Il s'agit de l'instruction la plus controversée en C. Cherchez sur internet et les détracteurs sont nombreux, et ils ont partiellement raison car dans la très vaste majorité des cas ou vous pensez avoir besoin de `goto`, une autre solution plus élégante existe.

Néanmoins, il est importante de comprendre que `goto` était dans certain langage de prorgamation comme BASIC, la seule structure de contrôle disponible permettant de faire des sauts. Elle est par ailleurs le reflet du langage machine car la plupart des processeurs ne connaissent que cette instruction souvent appellée `JUMP`. Il est par conséquent possible d'immiter le comportement de n'importe quelle structure de contrôle si l'on dispose de `if` et de `goto`.

`goto` effectue un saut inconditionnel à un *label* défini en C par un *identificateur* suivi d'un `:`.

L'un des seuls cas de figure autorisé est celui d'un traitement d'erreur centralisé lorsque de multiples points de retours existent dans une fonction ceci évitant de répéter du code :

```
#include <time.h>

int parse_message(int message)
{
    struct tm *t = localtime(time(NULL));
    if (t->tm_hour < 7) {
        goto error;
    }

    if (message > 1000) {
        goto error;
    }

    /* ... */

    return 0;

    error:
        printf("ERROR: Une erreur a été commise\n");
        return -1;
}
```

7.4.2 `continue`

Le mot clé `continue` ne peut exister qu'à l'intérieur d'une boucle. Il permet d'interrompre le cycle en cours et directement passer au cycle suivant.

```
uint8_t airplane_seat = 100;

while (--airplane_seat)
```

(suite sur la page suivante)

(suite de la page précédente)

```
{
    if (airplane_seat == 13) {
        continue;
    }

    printf("Dans cet avion il y a un siège numéro %d\n", airplane_
↪seat);
}
```

Cette structure est équivalente à l'utilisation d'un goto avec un label placé à la fin de la séquence de boucle, mais promettez-moi que vous n'utiliserez jamais cet exemple :

```
while (true)
{
    if (condition) {
        goto contin;
    }

    /* ... */

    contin :
}
```

7.4.3 `break`

Le mot-clé `break` peut être utilisé dans une boucle ou dans un `switch`. Il permet d'interrompre l'execution de la boucle ou de la structure `switch` la plus proche. Nous avions déjà évoqué l'utilisation dans un `switch` (c.f. Section 7.2.2).

7.4.4 `return`

Le mot clé `return` suivi d'une valeur de retour ne peut apparaître que dans une fonction dont le type de retour n'est pas `void`. Ce mot-clé permet de stopper l'exécution d'une fonction et de retourner à son point d'appel.

```
void unlock(int password)
{
    static tries = 0;

    if (password == 4710 /* MacGuyver: A Retrospective 1986 */) {
        open_door();
        tries = 0;
        return;
    }

    if (tries++ == 3)
```

(suite sur la page suivante)

(suite de la page précédente)

```
    {
        alert_security_guards();
    }
}
```

Exercice

Considérons les déclarations suivantes :

```
long i = 0;
double x = 100.0;
```

Indiquer la nature de l'erreur dans les expressions suivants :

1. ```
   do
       x = x / 2.0;
       i++;
   while (x > 1.0);
   ```

2. ```
   if (x = 0)
       printf("0 est interdit !\n");
   ```

3. ```
   switch(x) {
       case 100 :
           printf("Bravo.\n");
           break;
       default :
           printf("Pas encore.\n");

   }
   ```

4. ```
   for (i = 0 ; i < 10 ; i++);
       printf("%d\n", i);
   ```

5. ```
   while i < 100 {
       printf("%d", ++i);
   }
   ```

Exercice

Parmis les cas suivants, quel structure de contrôle utiliser ?

1. Test qu'une variable soit dans un interval donné.
2. Actions suivant un choix multiple de l'utilsateur
3. Rechercher un caractère particulier dans une chaîne de caractère
4. Itérer toutes les valeurs paires sur un interval donné
5. Demander la ligne suivante du télégramme à l'utilisateur jusqu'à `STOP`
   ```

1. **Le cas est circonscrit à un interval de valeur donnée, le `if` est approprié :**

```c
if (i > min && i < max) { /* ... */ }
```

2. **Dans ce cas un *switch* semble le plus approprié**

```c
switch(choice) {
    case 0 :
        /* ... */
        break;
    case 1 :
        /* ... */
}
```

3. **À reformuler *tant que le caractère n'est pas trouvé ou que la fin de la chaîne n'es***

```c
size_t pos;
while (pos < strlen(str) && str[pos] != c) {
    pos++;
}
if (pos == strlen(str)) {
    // Not found
} else {
    // Found `c` in `str` at position `pos`
}
```

4. **La boucle `for` semble ici la plus adaptée**

```c
for (size_t i = 100; i < 200; i += 2) {
    /* ... */
}
```

5. **Il est nécessaire ici d'assurer au moins un tour de boucle :**

```c
const size_t max_line_length = 64;
char format[32];
snprintf(format, sizeof(format), "%%%zus", max_line_length -
↪ 1);
unsigned int line = 0;
char buffer[max_lines][max_line_length];
do {
    printf("%d. ", line);
} while (
    scanf(format, buffer[line]) == 1 &&
    strcmp(buffer[line], "STOP") &&
    ++line < max_lines
);
```

Exercice

Un texte est passé à un programme par `stdin`. Comptez le nombre de caractères transmis.

```
$ echo "Hello world" | count-this
11
```

Exercice

Quel est le problème avec cette ligne de code ?

```
if (x&mask==bits)
```

La priorité de l'opérateur unitaire `&` est plus élevée que `==` ce qui se traduit par :

```
if (x & (mask == bits))
```

Le développeur voulait probablement appliquer le masque à `x` puis le comparer au motif `bits`. La bonne réponse devrait alors être :

```
if ((x & mask) == bits)
```

Chapitre 8

Programmes et Processus

8.1 Qu'est-ce qu'un programme ?

Un programme informatique est un ensemble d'opérations destinées à être exécutées par un ordinateur.

Un programme peut se décliner sous plusieurs formes :
- Code source
- Listing assembleur
- Exécutable binaire

Un processus est l'état d'un programme en cours d'exécution. Lorsqu'un programme est exécuté, il devient processus pendant un temps donné. Les systèmes d'exploitation tels que Windows sont dit multitâches, il peuvent par conséquent faire tourner plusieurs processus en parallèle. Le temps processeur est ainsi partagé entre chaque processus.

8.1.1 Code source

Le **code source** est généralement écrit par un ingénieur/développeur/informaticien. Il s'agit le plus souvent d'un fichier texte lisible par un être humain et souvent pourvu de commentaires facilitant sa compréhension. Selon le langage de programmation utilisé, la programmation peut être graphique comme avec les diagrammes Ladder utilisés dans les automates programmables et respectant la norme IEC 61131-3, ou LabView un outil de développement graphique.

Le plus souvent le code source est organisé en une arborescence de fichiers. Des programmes complexes comme le noyau Linux contiennent plus de 100'000 fichiers et 10 millions de lignes de code, pour la plupart écrites en C.

Fig. 8.1 – Programmeuse en tenue décontractée à côté de 62'500 cartes perforées

8.1.2 Exécutable binaire

Une fois compilé en langage machine, il en résulte un fichier qui peut être exécuté soit par un système d'exploitation, soit sur une plateforme embarquée à microcontrôleur sans l'intermédiaire d'un système d'exploitation. On dit que ce type de programme est bare metal, qu'il s'exécute à même le métal.

Un exécutable binaire doit être compilé pour la bonne architecture matérielle. Un programme compilé pour un processeur INTEL ne pourra pas s'exécuter sur un processeur ARM, c'est pourquoi on utilise différents compilateurs en fonctions des architectures cibles. L'opération de compiler un programme pour une autre architecture, ou un autre système d'exploitation que celui sur lequel est installé le compilateur s'appelle la compilation croisée (cross-compilation).

Prenons l'exemple du programme suivant qui calcule la suite des nombres de Fibonacci :

Code source 8.1 – Fibonacci.c

```c
#include <stdio.h>

int main(void)
{
    int t1 = 0, t2 = 1;
    int n, next_term;
    printf("Enter the number of terms: ");
    scanf("%d", &n);
    printf("Fibonacci Series: ");
    for (size_t i = 1; i <= n; ++i)
    {
        printf("%d, ", t1);
        next_term = t1 + t2;
        t1 = t2;
        t2 = next_term;
    }
    printf("\n");
}
```

Une fois assemblé le code source est converti en langage assembleur, une version intermédiaire entre le C et le langage machine. L'exemple est compilé en utilisant gcc :

```
gcc Fibonacci.c -o fibonacci.exe
objdump -d fibonacci.exe
```

On obtiens un fichier similaire à ceci qui contient le code machine (`48 83 ec 20`), et l'équivalent en langage assembleur (`mov %fs:0x28,%rax`) :

```
0000000000000680 <main>:
680:   41 55                    push   %r13
682:   41 54                    push   %r12
684:   48 8d 35 59 02 00 00     lea    0x259(%rip),%rsi
```

(suite sur la page suivante)

(suite de la page précédente)

```
68b:   55                      push   %rbp
68c:   53                      push   %rbx
68d:   bf 01 00 00 00          mov    $0x1,%edi
692:   48 83 ec 18             sub    $0x18,%rsp
696:   64 48 8b 04 25 28 00    mov    %fs:0x28,%rax
69d:   00 00
69f:   48 89 44 24 08          mov    %rax,0x8(%rsp)
6a4:   31 c0                   xor    %eax,%eax
6a6:   e8 a5 ff ff ff          callq  650 <__printf_chk@plt>
6ab:   48 8d 74 24 04          lea    0x4(%rsp),%rsi
6b0:   48 8d 3d 49 02 00 00    lea    0x249(%rip),%rdi
6b7:   31 c0                   xor    %eax,%eax
6b9:   e8 a2 ff ff ff          callq  660 <__isoc99_scanf@plt>
6be:   48 8d 35 3e 02 00 00    lea    0x23e(%rip),%rsi
...
72e:   00 00
730:   75 0b                   jne    73d <main+0xbd>
732:   48 83 c4 18             add    $0x18,%rsp
736:   5b                      pop    %rbx
737:   5d                      pop    %rbp
738:   41 5c                   pop    %r12
73a:   41 5d                   pop    %r13
73c:   c3                      retq
73d:   e8 fe fe ff ff          callq  640 <__stack_chk_fail@plt>
742:   66 2e 0f 1f 84 00 00    nopw   %cs:0x0(%rax,%rax,1)
749:   00 00 00
74c:   0f 1f 40 00             nopl   0x0(%rax)
```

Avec un visualisateur hexadécimal, on peut extraire le langage machine du binaire exécutable. L'utilitaire `hexdump` est appelé avec deux options `-s` pour spécifier l'adresse de début, on choisi ici celle du début de la fonction `main` `0x680`, et `-n` pour n'extraire que les premiers 256 octets :

```
$ hexdump -s0x680 -n256 a.out
0000680 5541 5441 8d48 5935 0002 5500 bf53 0001
0000690 0000 8348 18ec 4864 048b 2825 0000 4800
00006a0 4489 0824 c031 a5e8 ffff 48ff 748d 0424
00006b0 8d48 493d 0002 3100 e8c0 ffa2 ffff 8d48
00006c0 3e35 0002 3100 bfc0 0001 0000 7fe8 ffff
00006d0 8bff 2444 8504 74c0 4c3d 2d8d 0236 0000
00006e0 bc41 0001 0000 01bd 0000 3100 0fdb 001f
00006f0 da89 c031 894c bfee 0001 0000 8349 01c4
0000700 4be8 ffff 8dff 2b04 eb89 c589 6348 2444
0000710 4c04 e039 da73 0abf 0000 e800 ff10 ffff
0000720 c031 8b48 244c 6408 3348 250c 0028 0000
0000730 0b75 8348 18c4 5d5b 5c41 5d41 e8c3 fefe
0000740 ffff 2e66 1f0f 0084 0000 0000 1f0f 0040
0000750 ed31 8949 5ed1 8948 48e2 e483 50f0 4c54
0000760 058d 016a 0000 8d48 f30d 0000 4800 3d8d
```

(suite sur la page suivante)

(suite de la page précédente)

```
0000770 ff0c ffff 15ff 0866 0020 0ff4 441f 0000
```

Il est facile de voir la correspondance entre l'assembleur et l'exécutable binaire. Les valeurs `41 55` puis `41 54` puis `48 8d 35 59` se retrouvent directement dans le *dump* : `5541 5441 8d48`. Si les valeurs sont interverties c'est parce qu'un PC est *little-endian* (c.f. Section 6.3), les octets de poids faible apparaissent par conséquent en premier dans la mémoire.

Sous Windows, l'extension des fichiers détermine leur type. Un fichier avec l'extension `.jpg` sera un fichier image du Join Photographic Experts Group et exécuter ce fichier correspond à l'ouvrir en utilisant l'application par défaut pour visualiser les images de ce type. Un fichier avec l'extension `.exe` est un exécutable binaire, et il sera exécuté en tant que programme par le système d'exploitation.

Sous POSIX (Linux, MacOS, UNIX), les *flags* d'un fichier qualifient son type. Le programme `ls` permet de visualiser les flags du programme `Fibonacci` que nous avons compilé :

```
$ ls -al a.out
-rwxr-xr-x 1 root ftp 8.3K Jul 17 09:53 Fibonacci
```

Les lettres `r-x` indiquent :

- **r** Lecture autorisée
- **w** Écriture autorisée
- **x** Exécution autorisée

Ce programme peut-être exécuté par tout le monde, mais modifié que par l'utilisateur `root`.

8.1.3 Entrées sorties

Tout programme doit pouvoir interagir avec son environnement. A l'époque des télé-scripteurs, un programme interagissait avec un clavier et une imprimante matricielle. Avec l'arrivée des systèmes d'exploitation, le champ d'action fut réduit à des entrées :

- L'entrée standard `STDIN` fourni au programme du contenu qui est généralement fourni par la sortie d'un autre programme.
- Les arguments du programme `ARGV`
- Les variables d'environnement `ENVP`

Ainsi qu'à des sorties :

- La sortie standard `STDOUT` est généralement affichée à l'écran
- La sortie d'erreur standard `STDERR` contient des détails sur les éventuelles erreurs d'exécution du programme.

La figure suivante résume les interactions qu'un programme peut avoir sur son environnement. Les appels système (syscall) sont des ordres transmis directement au système d'exploitation. Ils permettent par exemple de lire des fichiers, d'écrire à l'écran, de mettre le programme en pause ou de terminer le programme.

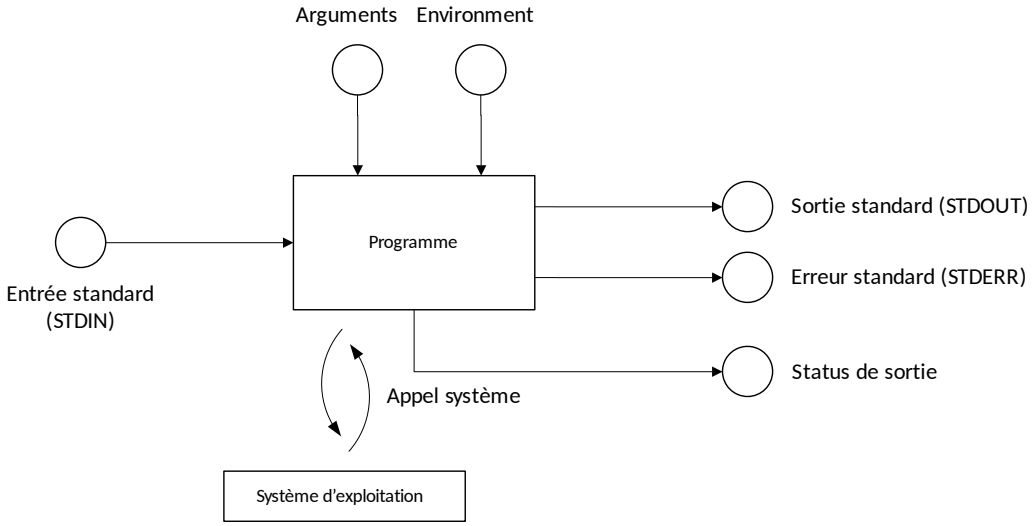


Fig. 8.2 – Résumé des interactions avec un programme

8.1.4 Signaux

Lorsqu'un programme est en cours d'exécution, il peut recevoir de la part du système d'exploitation des signaux. Il s'agit d'une notification asynchrone envoyée à un processus pour lui signaler l'apparition d'un évènement.

Si, en utilisant Windows, vous vous rendez dans le gestionnaire de tâches et que vous décider de *Terminer une tâche*, le système d'exploitation envoie un signal au programme lui demandant de se terminer.

Sous Linux, habituellement, le *shell* relie certains raccourcis clavier à des signaux particuliers :

- `C-c` envoie le signal `SIGINT` pour interrompre l'exécution d'un programme
- `C-z` envoie le signal `SIGTSTP` pour suspendre l'exécution d'un programme
- `C-t` envoie le signal `SIGINFO` permettant de visualiser certaines informations liées à l'exécution du processus.

Si le programme suivant est exécuté, il sera bloquant, c'est-à-dire qu'à moins d'envoyer un signal d'interruption, il ne sera pas possible d'interrompre le processus :

```
int main(void)
{
    for(;;);
}
```

8.2 Arguments et options

L'interpréteur de commande `cmd.exe` sous Windows ou `bash` sous Linux, fonctionne de façon assez similaire. L'**invite de commande** nommée *prompt* en anglais invite l'utilisateur à entrer une commande. Sous DOS puis sous Windows cet invite de commande ressemble à ceci :

```
C:\>
```

Sous Linux, le prompt est largement configurable et dépend de la distribution installée, mais le plus souvent il se termine par le caractère `$` ou `#`.

Une commande débute par le nom de cette dernière, qui peut être le nom du programme que l'on souhaite exécuter puis vient les arguments et les options.

- Une **option** est par convention un **argument** dont le préfix est `-` sous Linux ou `/` sous Windows même si le standard GNU gagne du terrain. Aussi, le consensus le plus large semble être le suivant :
- Une option peut être exprimée soit sous format court `-o`, `-v`, soit sous format long `--output=`, `--verbose` selon qu'elle commence par un ou deux tirets. Une option peut être un booléenne (présence ou non de l'option), ou scalaire, c'est-à-dire être associée à une valeur `--output=foo.o`. Les options modifient le comportement interne d'un programme.

Un argument est une chaîne de caractères utilisée comme entrée au programme. Un programme peut avoir plusieurs arguments.

En C, c'est au développeur de distinguer les options des arguments, car ils sont tous passés par le paramètre `argv` :

```c
#include <stdio.h>

int main(int argc, char *argv[]) {
    printf("Liste des arguments et options passés au programme:\n");

    for (size_t i = 0; i < argc; i++) {
        printf("  %u. %s\n", i, argv[i]);
    }
}
```

```
$ argverbose --help -h=12 3.14 'Baguette au beurre'␣
↪$'\t-Lait\n\t-Viande\n\t-Oeufs\f'
Liste des arguments et options passés au programme :
0. ./a.out
1. --help
2. -h=12
3. 3.14
4. Baguette au beurre
5.  -Lait
    -Viande
    -Oeufs
```

8.2.1 Norme POSIX

Le standard POSIX décrit une façon de distinguer des *options* passées à un programme. Par exemple, le programme cowsay peut être paramétré pour changer son comportement en utilisant des *options* standards comme `-d`. La fonction `getopt` disponible dans la biblothèque `<unistd.h>` permet de facilement interpréter ces options.

8.2.2 Extension GNU

Malheureusement, la norme POSIX ne spécifie que les options dites courtes (un tiret suivi d'un seul caractère). Une extension GNU et son en-tête `<getopt.h>` permet l'accès à la fonction `getopt_long` laquelle permet d'interpréter aussi les options longues `--version` qui sont devenues très répandues.

Ci-dessous une possible utilisation de cette fonction :

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

typedef struct Options {
    bool is_verbose;

    bool has_add;
    bool has_append;
    bool has_delete;
    bool has_create;

    char *create_name;
    char *delete_name;
    char *file_name;
} Options;

Options parse_options(int argc, char *argv[])
{
    Options options = {0};

    int c;
    static int verbose_flag;  // Set by --verbose/--brief

    for (;;) {
        static struct option long_options[] = {
            // These options set a flag.
            {"verbose", no_argument, &verbose_flag, true},
            {"brief", no_argument, &verbose_flag, false},

            // These options don't set a flag. We distinguish them␣
↪by their
```

(suite sur la page suivante)

(suite de la page précédente)

```
            // indices.
            {"add", no_argument, 0, 'a'},
            {"append", no_argument, 0, 'b'},
            {"delete", required_argument, 0, 'd'},
            {"create", required_argument, 0, 'c'},
            {"file", required_argument, 0, 'f'},

            // Sentinel marking the end of the structure.
            {0, 0, 0, 0}};

        // getopt_long stores the option index here.
        int option_index = 0;

        c = getopt_long(argc, argv, "abc:d:f:", long_options, &
↪option_index);

        // Detect the end of the options.
        if (c == -1) break;

        switch (c) {
            case 'a':
                options.has_add = true;
                break;

            case 'b':
                options.has_append = true;
                break;

            case 'c':
                options.create_name = optarg;
                break;

            case 'd':
                options.delete_name = optarg;
                break;

            case 'f':
                options.file_name = optarg;
                break;

            case '?':
                // getopt_long already printed an error message.
                break;

            default:
                abort();
        }
    }
    options.is_verbose = verbose_flag;
```

(suite sur la page suivante)

(suite de la page précédente)

```
    // Parses the remaining command line arguments if got any
    while (optind < argc) printf("%s\n", argv[optind++]);

    return options;
}

int main(int argc, char **argv)
{
    Options options = parse_options(argc, argv);

    // ...
}
```

8.2.3 Windows

Windows utilise à l'instare de `RDOS` ou OpenVMS, le caractère *slash* pour identifier ses options. Alors que sous POSIX l'affichage de la liste des fichiers s'écrira peut-être `ls -l -s D*`, sous Windows on utilisera `dir /q d* /o:s`.

8.3 Fonction main

Le standard définit une fonction nommée `main` comme étant la fonction principale appelée à l'exécution du programme. Or, sur un système d'exploitation, la fonction `main` a déjà été appelée il y a belle lurette lorsque l'ordinateur a été allumé et que le BIOS a chargé le système d'exploitation en mémoire. Dès lors la fonction `main` de notre programme Hello World n'est pas la première, mais est appelé.

8.3.1 Qui appelle main ?

Un exécutable binaire à un format particulier appelé **ELF** (Executable and Linkable Format) qui contient un **point d'entrée** qui sera l'adresse mémoire de début du programme. Sous un système POSIX ce point d'entrée est nommé `_init`. C'est lui qui est responsable de récolter les informations transmises par le système d'exploitation. Ce dernier transmet sur la **pile** du programme :

- Le nombre d'arguments `argc`
- La liste des arguments `argv`
- Les variables d'environnements `envp`
- Les pointeurs de fichiers sur `stdout`, `stdin`, `stderr`

C'est la fonction `__libc_start_main` de la bibliothèque standard qui a la responsabilité d'appeler la fonction `main`. Voici son prototype :

```
int __libc_start_main(int (*main) (int, char**, char**),
    int argc, char** ubp_av,
    void (*init)(void),
    void (*fini)(void),
    void (*rtld_fini)(void),
    void (*stack_end)
);
```

8.3.2 Valeur de retour

La fonction `main` renvoie toujours une valeur de retour qui agit comme le statut de sortie d'un programme (exit status). Sous POSIX et sous Windows, le programme parent s'attend à recevoir une valeur 32-bits à la fin de l'exécution d'un programme. L'interprétation est la suivante :

`0` Succès, le programme s'est terminé correctement.

`!0` Erreur, le programme ne s'est pas terminé correctement.

Par exemple le programme `printf` retourne dans le cas précis l'erreur 130 :

```
$ printf '%d' 42
42
$ echo $?
0

$ printf '%d' 'I am not a number'
printf: I am not a number: invalid number
$ echo $?
130
```

8.4 Entrées sorties standards

Le fichier d'en-tête `stdio.h` (man stdio) permet de simplifier l'interaction avec les fichiers. Sous Linux et MacOS principalement, mais d'une certaine manière également sous Windows, les canaux d'échanges entre un programme et son hôte (*shell*, gestionnaire de fenêtre, autre programme), se font par l'intermédiaire de fichiers particuliers nommés `stdin`, `stdout` et `stderr`.

La fonction de base est `putchar` qui écrit un caractère sur `stdout` :

```
#include <stdio.h>

int main(void) {
    putchar('H');
    putchar('e');
    putchar('l');
```

(suite sur la page suivante)

(suite de la page précédente)

```
    putchar('l');
    putchar('o');
    putchar('\n');
}
```

Bien vite, on préfèrera utiliser `printf` qui simplifie le formatage de chaînes de caractères et qui permet à l'aide de marqueurs (*tokens*) de formater des variables :

```
#include <stdio.h>

int main(void) {
    printf("Hello\v");
    printf("%d, %s, %f", 0x12, "World!", 3.1415);
}
```

Il peut être nécessaire, surtout lorsqu'il s'agit d'erreurs qui ne concernent pas la sortie standard du programme, d'utiliser le bon canal de communication, c'est-à-dire `stderr` au lieu de `stdout`. La fonction `fprintf` permet de spécifier le flux standard de sortie :

```
#include <stdio.h>

int main(void) {
    fprintf(stdout, "Sortie standard\n");
    fprintf(stderr, "Sortie d'erreur standard\n");
}
```

Pourquoi, me direz-vous, faut-il séparer la sortie standard du canal d'erreur ? Le plus souvent un programme n'est pas utilisé seul, mais en conjonction avec d'autres programmes :

```
$ echo "Bonjour" | tr 'A-Za-z' 'N-ZA-Mn-za-m' > data.txt
$ cat data.txt
Obawbhe
```

Dans cet exemple ci-dessus le programme `echo` prends en argument la chaîne de caractère `Bonjour` qu'il envoie sur la sortie standard. Ce flux de sortie est relié au flux d'entrée du programme `tr` qui effectue une opération de ROT13 et envoie le résultat sur la sortie standard. Ce flux est ensuite redirigé sur le fichier `data.txt`. La commande suivante `cat` lis le contenu du fichier dont le nom est passé en argument et écrit le contenu sur la sortie standard.

Dans le cas où un de ces programmes génère une alerte (*warning*), le texte ne sera pas transmis le long de la chaîne, mais simplement affiché sur la console. Il est donc une bonne pratique que d'utiliser le bon flux de sortie : `stdout` pour la sortie standard et `stderr` pour les messages de diagnostique et les erreurs.

8.5 Boucle d'attente

Comme évoqué, un programme est souvent destiné à tourner sur un système d'exploitation. Un programme simple comme celui-ci :

```
int main(void) {
    for(;;) {}
}
```

consommera 100% des ressources du processeur. En d'autres termes, le processeur dépensera toute son énergie à faire 150 millions de calculs par seconde, pour rien. Et les autres processus n'auront que très peu de ressources disponibles pour tourner.

Il est grandement préférable d'utiliser des appels système pour indiquer au noyau du système d'exploitation que le processus souhaite être mis en pause pour un temps donné. Le programme suivant utilise la fonction standard `sleep` pour demander au noyau d'être mis en attente pour une période de temps spécifiée en paramètre.

```
#include <unistd.h>

int main(void) {
    for(;;) {
        sleep(1 /* seconds */);

        ...
    }
```

Alternativement, lorsqu'un programme attend un retour de l'utilisateur par exemple en demandant la saisie au clavier d'informations, le système d'exploitation est également mis en attente et le processus ne consomme pas de ressources CPU. Le programme ci-dessous attend que l'utilisateur presse la touche enter.

```
#include <stdio.h>

int main(void) {
    for(;;) {
        getchar();
        printf("Vous avez pressé la touche enter (\\n)\n");
    }
}
```

Exercice

En rappelant l'historique des dernières commandes exécutées sur l'ordinateur du professeur pendant qu'il avait le dos tourné, vous tombez sur cette commande :

```
$ fortune | cowsay | lolcat
```

Quelle est sa structure et que fait-elle ?

Chapitre 9

Entrées Sorties

Comme nous l'avons vu (c.f. Section 8.1.3) un programme dispose de canaux d'entrées sorties `stdin`, `stdout` et `stderr`. Pour faciliter la vie du programmeur, les bibliothèques standard offrent toute une panoplie de fonctions pour formater les sorties et interpréter les entrées.

La fonction phare est bien entendu `printf` pour le formatage de chaîne de caractères et `scanf` pour la lecture de chaînes de caractères. Ces dernières fonctions se déclinent en plusieurs variantes :

- depuis/vers les canaux standards `printf`, `scanf`
- depuis/vers un fichier quelconque `fprintf`, `fscanf`
- depuis/vers une chaîne de caractères `sprintf`, `sscanf`

La liste citée est non exhaustive, mais largement documentée ici : <stdio.h>.

9.1 Sorties non formatées

Si l'on souhaite simplement écrire du texte sur la sortie standard, deux fonctions sont disponibles :

`putchar(char c)` Pour imprimer un caractère unique : `putchar('c')`

`puts(char[] str)` Pour imprimer une chaîne de caractères

Exercice

Écrire un programme qui retourne un mot parmi une liste de mot, de facon aléatoire.

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

char *words[] = {"Albédo", "Bigre", "Maringouin", "Pluripotent",␣
↪"Entrechat", "Caracoler" "Palinodie", "Sémillante", "Atavisme",␣
↪"Cyclothymie", "Idiosyncratique", "Entéléchie"};

#if 0
    srand(time(NULL));   // Initialization, should only be called␣
↪once.
```

(suite sur la page suivante)

(suite de la page précédente)

```
    size_t r = rand() % sizeof(words) / sizeof(char*); // Generate␣
↪random value
#endif
```

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

char *words[] = {"Albédo", "Bigre", "Maringouin", "Pluripotent",␣
↪"Entrechat", "Caracoler" "Palinodie", "Sémillante", "Atavisme",␣
↪"Cyclothymie", "Idiosyncratique", "Entéléchie"};

int main(void)
{
    srand(time(NULL));
    puts(words[rand() % (sizeof(words) / sizeof(char*))]);

    return 0;
}
```

9.2 Sorties formatées

Convertir un nombre en une chaîne de caractères n'est pas trivial. Prenons l'exemple de la valeur 123. Il faut pour cela diviser itérativement le nombre par 10 et calculer le reste :

```
Etape  Opération  Resultat  Reste
1      123 / 10   12        3
2      12 / 10    1         2
3      1 / 10     0         1
```

Comme on ne sait pas à priori combien de caractères on aura, et que ces caractères sont fournis depuis le chiffre le moins significatif, il faudra inverser la chaîne de caractères produite.

Voici un exemple possible d'implémentation :

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

void swap(char* a, char* b)
{
    *a ^= *b;
    *b ^= *a;
    *a ^= *b;
}

void reverse(char* str, size_t length)
```

(suite sur la page suivante)

(suite de la page précédente)

```
{
    for (size_t start = 0, end = length - 1; start < end; start++,␣
↪end--)
    {
        swap(str + start, str + end);
    }
}

void my_itoa(int num, char* str)
{
    const unsigned int base = 10;
    bool is_negative = false;
    size_t i = 0;

    if (num == 0) {
        str[i++] = '0';
        str[i] = '\0';
        return;
    }

    if (num < 0) {
        is_negative = true;
        num = -num;
    }

    while (num != 0) {
        int rem = num % 10;
        str[i++] = rem + '0';
        num /= base;
    }

    if (is_negative)
        str[i++] = '-';

    str[i] = '\0';

    reverse(str, i);
}
```

Cette implémentation pourrait être utilisée de la façon suivante :

```
#include <stdlib.h>

int main(void)
{
    int num = 123;
    char buffer[10];

    itoa(num, buffer);
```

(suite sur la page suivante)

(suite de la page précédente)

```
}
```

9.3 printf

Vous conviendrez que devoir manuellement convertir chaque valeur n'est pas des plus pratique, c'est pourquoi `printf` rend l'opération bien plus aisée en utilisant des marques substitutives (*placeholder*). Ces spécifieurs débutent par le caractère `%` suivi du formatage que l'on veut appliquer à une variable passée en paramètres. L'exemple suivant utilise `%d` pour formater un entier non signé.

```
#include <stdio.h>

int main()
{
    int32_t earth_perimeter = 40075;
    printf("La circonférence de la terre vaut vaut %d km", earth_
↪perimeter);
}
```

Le standard **C99** défini le prototype de `printf` comme étant :

```
int printf(const char *restrict format, ...);
```

Il définit que la fonction `printf` prend en paramètre un format suivi de `...`. La fonction `printf` comme toutes celles de la même catégorie sont dites variadiques, c'est-à-dire qu'elles peuvent prendre un nombre variable d'arguments. Il y aura autant d'arguments additionnels que de marqueurs utilisés dans le format. Ainsi le format `"Mes nombres préférés sont %d et %d, mais surtout %s"` demandera trois paramètres additionnels :

La fonction retourne le nombre de caractères formatés ou `-1` en cas d'erreur.

La construction d'un marqueur est loin d'être simple, mais heureusement on n'a pas besoin de tout connaitre et la page wikipedia printf format string est d'une grande aide. Le format de construction est le suivant :

```
%[parameter][flags][width][.precision][length]type
```

`parameter` (optionnel) Numéro de paramètre à utiliser

`flags` (optionnel) Modificateurs : préfixe, signe plus, alignement à gauche ...

`width` (optionnel) Nombre **minimum** de caractères à utiliser pour l'affichage de la sortie.

`.precision` (optionnel) Nombre **minimum** de caractères affichés à droite de la virgule. Essentiellement, valide pour les nombres à virgule flottante.

`length` (optionnel) Longueur en mémoire. Indique la longueur de la représentation binaire.

type Type de formatage souhaité

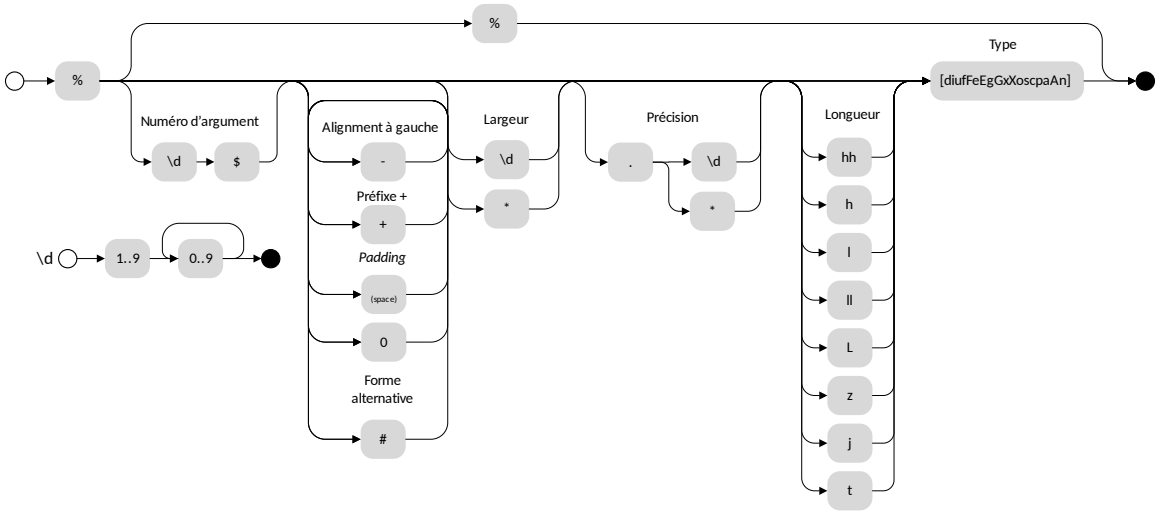


Fig. 9.1 – Formatage d'un marqueur

9.3.1 Exemples

Exemple	Sortie	Taille
`printf("%c", 'c')`	c	1
`printf("%d", 1242)`	1242	4
`printf("%10d", 42)`	“ 42“	10
`printf("%07d", 42)`	0000042	7
`printf("%+-5d", 23)`	“+23 “	6
`printf("%5.3f", 314.15)`	314.100	7
`printf("%*.*f", 4, 2, 102.1)`	102.10	7
`printf("%8x", 3141592)`	“ 2fefd8“	6
`printf("%s", "Hello")`	Hello	5

Exercice

Indiquez les erreurs dans les instructions suivantes :

```
printf("%d%d\n", 10, 20);
printf("%d, %d, %d\n", 10, 20);
printf("%d, %d, %d, %d\n", 10, 20, 30, 40.);
printf("%*d, %*d\n", 10, 20);
printf("%6.2f\n", 10);
printf("%10s\n", 0x9f);
```

9.4 Entrées formatées

À l'instar de la sortie formatée, il est possible de lire les saisies au clavier ou *parser* une chaîne de caractères, c'est-à-dire faire un analyse syntaxique de son contenu pour en extraire de l'information.

La fonction `scanf` est par exemple utilisée à cette fin :

```c
#include <stdio.h>

int main()
{
    int favorite;

    printf("Quelle est votre nombre favori ? ");
    scanf("%d", &favorite);

    printf("Saviez-vous que votre nombre favori, %d, est %s ?\n",
        favorite,
        favorite % 2 ? "impair" : "pair");
}
```

Cette fonction utilise l'entrée standard `stdin`. Il est donc possible soit d'exécuter ce programme en mode interactif :

```
$ ./a.out
Quelle est votre nombre favori ? 2
Saviez-vous que votre nombre favori, 2, est pair ?
```

soit d'exécuter ce programme en fournissant le nécessaire à stdin :

```
$ echo "23" | ./a.out
Quel est votre nombre favori ? Saviez-vous que votre nombre favori,␣
↪23, est impair ?
```

On observe ici un comportement différent, car le retour clavier lorsque la touche *enter* est pressée n'est pas transmis au programme, mais c'est le shell qui l'intercepte.

9.4.1 scanf

Le format de `scanf` se rapproche de `printf` mais en plus simple. Le man scanf ou même la page Wikipedia de scanf renseigne sur son format.

Cette fonction tient son origine une nouvelle fois de ALGOL 68 (`readf`), elle est donc très ancienne.

La compréhension de `scanf` n'est pas évidente et il est utile de se familiariser sur son fonctionnement à l'aide de quelques exemples.

Le programme suivant lit un entier et le place dans la variable `n`. `scanf` retourne le nombre d'assignements réussis. Ici, il n'y a qu'un *placeholder*, on s'attend naturellement à lire `1` si la fonction réussit. Le programme écrit ensuite les nombres dans l'ordre d'apparition.

```c
#include <stdio.h>

int main(void)
{
    int i = 0, n;

    while (scanf("%d", &n) == 1)
        printf("%i\t%d\n", ++i, n);
    return 0;
}
```

Si le code est exécuté avec une suite arbitraire de nombres :

```
456 123 789     456 12
456 1
    2378
```

il affichera chacun des nombres dans l'ordre d'apparition :

```
$ cat << EOF | ./a.out
456 123 789     456 12
456 1
    2378
EOF
1       456
2       123
3       789
4       456
5       12
6       456
7       1
8       2378
```

Voyons un exemple plus complexe (c.f. C99 §7.19.6.2-19).

```c
int count;
float quantity;
char units[21], item[21];

do {
    count = scanf("%f%20s de %20s", &quant, units, item);
    scanf("%*[^\n]");
} while (!feof(stdin) && !ferror(stdin));
```

Lorsqu'exécuté avec ce contenu :

```
2 litres de lait
-12.8degrés Celsius
beaucoup de chance
10.0KG de
poussière
100ergs d'énergie
```

Le programme se déroule comme suit :

```
quantity = 2; strcpy(units, "litres"); strcpy(item, "lait");
count = 3;

quantity = -12.8; strcpy(units, "degrees");
count = 2; // "C" échoue lors du test de "d" (de)

count = 0; // "b" de "beaucoup" échoue contre "%f" s'attendant à un␣
↪float

quantity = 10.0; strcpy(units, "KG"); strcpy(item, "poussière");
count = 3;

count = 0; // "100e" échoue contre "%f", car "100e3" serait un␣
↪nombre valable
count = EOF; // Fin de fichier
```

Dans cet exemple, la boucle `do... while` est utilisée, car il n'est pas simplement possible de traiter le cas `while(scanf(...) > 0` puisque l'exemple cherche à montrer les cas particuliers où justement, la capture échoue. Il est nécessaire alors de faire appel à des fonctions de plus bas niveau `feof` pour détecter si la fin du fichier est atteinte, et `ferror` pour détecter une éventuelle erreur sur le flux d'entrée.

La directive `scanf("%*[^\n]");` étant un peu particulier, il peut valoir la peine de s'y attarder un peu. Le *flag* `*`, différent de `printf` indique d'ignorer la capture en cours. L'exemple suivant montre comment ignorer un mot.

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

int main(void) {
    int a, b;
    char str[] = "24 kayaks 42";

    sscanf(str, "%d%*s%d", &a, &b);
    assert(a == 24);
    assert(b == 42);
}
```

Ensuite, `[^\n]`. Le marqueur `[`, terminé par `]` cherche à capturer une séquence de caractères parmi une liste de caractères acceptés. Cette syntaxe est inspirée des expressions régulières très utilisées en informatique. Le caractère `^` à une signification particulière,

il indique que l'on cherche à capturer une séquence de caractères parmi une liste de caractères **qui ne sont pas acceptés**. C'est une sorte de négation. Dans le cas présent, cette directive `scanf` cherche à consommer tous les caractères jusqu'à une fin de ligne, car, dans le cas ou la capture échoue à `C` de `Celsius`, le pointeur de fichier est bloqué au caractère `C` et au prochain tour de boucle, `scanf` échouera au même endroit. Cette instruction est donc utilisée pour repartir sur des bases saines en sautant à la prochaine ligne. Exercice

Considérant les déclarations :

```
int i, j, k;
float f;
```

Donnez les valeurs de chacune des variables après exécution. Chaque ligne est indépendante des autres.

```
i = sscanf("1 12.5", "%d %d, &j, &k);
sscanf("12.5", "%d %f", &j, %f);
i = sscanf("123 123", "%d %f", &j, &f);
i = sscanf("123a 123", "%d %f", &j, &f);
i = sscanf("%2d%2d%f", &j, &k, &f);
```

Exercice

Considérant les déclarations suivantes, donner la valeur des variables après l'exécution des instructions données avec les captures associées :

```
int i = 0, j = 0, n = 0;
float x = 0;
```

1. `n = scanf("%1d%1d", &i, &j);`, `12\n`
2. `n = scanf("%d%d", &i, &j);`, `1 , 2\n`
3. `n = scanf("%d%d", &i, &j);`, `-1 -2\n`
4. `n = scanf("%d%d", &i, &j);`, `- 1 - 2\n`
5. `n = scanf("%d,%d", &i, &j);`, `1 , 2\n`
6. `n = scanf("%d ,%d", &i, &j);`, `1 , 2\n`
7. `n = scanf("%4d %2d", &i, &j);`, `1 234\n`
8. `n = scanf("%4d %2d", &i, &j);`, `1234567\n`
9. `n = scanf("%d%*d%d", &i, &j);`, `123 456 789\n`
10. `n = scanf("i=%d , j=%d", &i, &j);`, `1 , 2\n`
11. `n = scanf("i=%d , j=%d", &i, &j);`, `i=1, j=2\n`
12. `n = scanf("%d%d", &i, &j);`, `1.23 4.56\n`
13. `n = scanf("%d.%d", &i, &j);`, `1.23 4.56\n`
14. `n = scanf("%x%x", &i, &j);`, `12 2a\n`
15. `n = scanf("%x%x", &i, &j);`, `0x12 0X2a\n`
16. `n = scanf("%o%o", &i, &j);`, `12 018\n`

17. `n = scanf("%f", &x);`, `123\n`
18. `n = scanf("%f", &x);`, `1.23\n`
19. `n = scanf("%f", &x);`, `123E4\n`
20. `n = scanf("%e", &x);`, `12\n`

1. `i` : 1, `j` : 2, `n` : 2
2. `i` : 1, `j` : 0, `n` : 1. `j` n'est pas lue car arrêt prématuré sur `,`
3. `i` : -1, `j` : -2, `n` : 2
4. `i` : 0, `j` : 0, `n` : 0. `i` n'est pas lue car arrêt prématuré sur `-`
5. `i` : 1, `j` : 0, `n` : 1.
6. `i` : 1, `j` : 2, `n` : 2
7. `i` : 1, `j` : 23, `n` : 2
8. `i` : 1234, `j` : 56, `n` : 2
9. `i` : 123, `j` : 789, `n` : 2
10. `i` : 0, `j` : 0, `n` : 0
11. `i` : 1, `j` : 2, `n` : 2
12. `i` : 1, `j` : 0, `n` : 1
13. `i` : 1, `j` : 23, `n` : 2
14. `i` : 18, `j` : 42, `n` : 2
15. `i` : 10, `j` : 1, `n` : 2. Le chiffre 8 interdit en octal provoque un arrêt
16. `x` : `123.`, `n` : 1
17. `x` : `1.23`, `n` : 1
18. `x` : `1.23E6`, `n` : 1
19. `x` : `12`, `n` : 1

Exercice

1. Saisir 3 caractères consécutifs dans des variables `i`, `j`, `k`.
2. Saisir 3 nombres de type float séparés par un point-virgule et un nombre quelconque d'espaces dans des variables `x`, `y` et `z`.
3. Saisir 3 nombres de type double en affichant avant chaque saisie le nom de la variable et un signe `=`, dans des variables `t`, `u` et `v`.

1. **Saisir 3 caractères consécutifs dans des variables `i`, `j`, `k`.**

```
scanf("%c%c%c", &i, &j, &k);
```

2. **Saisir 3 nombres de type float séparés par un point-virgule et un nombre quelconq**

```
scanf("%f ;%f ;%f", &x, &y, &z);
```

3. **Saisir 3 nombres de type double en affichant avant chaque saisie le nom de la varia**

```
printf("t="); scanf("%f", &t);
printf("u="); scanf("%f", &u);
printf("v="); scanf("%f", &v);
```

9.4.2 Saisie de chaîne de caractères

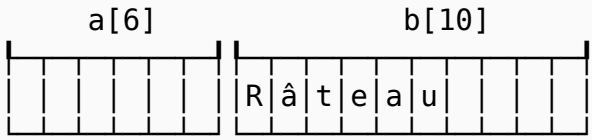
Lors d'une saisie de chaîne de caractères, il est nécessaire de **toujours** indiquer une taille maximum de chaîne comme `%20s` qui limite la capture à 20 caractères, soit une chaîne de 21 caractères avec son `\0`. Sinon, il y a risque de fuite mémoire :

```
int main(void) {
    char a[6];
    char b[10] = "Râteau";

    char str[] = "jardinage";
    sscanf(str, "%s", a);

    printf("a. %s\nb. %s\n", a, b);
}
```

```
$ ./a.out
a. jardinage
b. age
```

Ici la variable b contient `age` alors qu'elle devrait contenir `râteau`. La raison est que le mot capturé `jardinage` est trop long pour la variable `a` qui n'est disposée à stocker que 5 caractères imprimables. Il y a donc dépassement mémoire et comme vous le constatez, le compilateur ne génère aucune erreur. La bonne méthode est donc de protéger la saisie ici avec `%5s`.

En mémoire, ces deux variables sont adjacentes et naturellement *a[7]* est équivalent à dire *la septième case mémoire à partir du début de "a"*.



9.4.3 Saisie arbitraire

Comme brièvement évoqué plus haut, il est possible d'utiliser le marqueur `[` pour capturer une séquence de caractères. Imaginons que je souhaite capturer un nombre en tetrasexagesimal (base 64). Je peux écrire :

```
char input[] = "Q2hvY29sYXQ";
char output[128];
sscanf(input, "%127[0-9A-Za-z+/]", &output);
```

Dans cet exemple je capture les nombres de 0 à 9 `0-9` (10), les caractères majuscules et minuscules `A-Za-z` (52), ainsi que les caractères `+`, `/` (2), soit 64 caractères. Le buffer d'entrée étant fixé à 128 positions, la saisie est contrainte à 127 caractères imprimables.

Exercice

Parmi les instructions ci-dessous, indiquez celles qui sont coorectes et celle qui comportent des erreurs. Pour celles comportant des erreurs, détaillez la nature des anomalies.

```
short i;
long j;
unsigned short u;
float x;
double y;
printf(i);
scanf(&i);
printf("%d", &i);
scanf("%d", &i);
printf("%d%ld", i, j, u);
scanf("%d%ld", &i, j);
printf("%u", &u);
scanf("%d", &u);
printf("%f", x);
scanf("%f", &x);
printf("%f", y);
scanf("%f", &y);
```

```
// Incorrect ! Le premier paramètre de printf doit être la chaîne␣
↪de format.
printf(i);

// Incorrect ! Le premier paramètre de scanf doit être la chaine de␣
↪format.
scanf(&i);

// Correct, mais surprenant.
// Cette instruction affichera l'adresse de I, et non pas sa valeur␣
↪!
printf("%d", &i);

// Incorrect. Le paramètre i est de type short, alors que la chaîne␣
↪de
// format spécifie un type int. Fonctionnera sur les machines dont␣
↪le type
// short et int sont identiques
```

(suite sur la page suivante)

(suite de la page précédente)

```
scanf("%d", &i);

// Incorrect, la troisième variable passée en paramètre ne sera pas␣
 ↪affichée.
printf("%d%ld", i, j, u);

// Incorrect ! Le premier paramètre est de type short alors que int
// est spécifié dans la chaîne de format.
// Le deuxième paramètre n'est pas passé par adresse, ce qui va
// probablement causer une erreur fatale.
scanf("%d%ld", &i, j);

// Correct, mais étonnant. Affiche l'adresse de la variable u.
printf("%u", &u);

// Incorrect ! Le paramètre est de type unsigned short, alors que
// la chaîne de format spécifie int. Fonctionnera pour les valeurs
// positives sur les machines dont le type short et int sont␣
 ↪identiques.
// Pour les valeurs négatives, le résultat sera l'interprétation non
// signée de la valeur en complément à 2.
scanf("%d", &u);

// Correct, mais x est traité comme double.
printf("%f", x);

// Correct.
scanf("%f", &x);

// Correct ! %f est traité comme double par printf !
printf("%f", y);

// Incorrect ! La chaîne de format spécifie float,
// le paramètre passé est l'adresse d'une variable de type double.
scanf("%f", &y);
```

Exercice

Écrivez un programme déclarant des variables réelles `x`, `y` et `z`, permettant de saisir leur valeur en une seule instruction, et vérifiant que les 3 valeurs ont bien été assignées. Dans le cas contraire, afficher un message du type "données invalides".

```
int n;
float x, y, z;
printf("Donnez les valeurs de x, y et z :");
n = scanf("%f%f%f", &x, &y, &z);
if (n != 3)
printf("Erreur de saisie.\n");
```

Exercice

Écrire un programme effectuant les opérations suivantes :
- Saisir les coordonnées réelles `x1` et `y1` d'un vecteur `v1`.
- Saisir les coordonnées réelles `x2` et `y2` d'un vecteur `v2`.
- Calculer le produit scalaire. Afficher un message indiquant si les vecteurs sont orthogonaux ou non.

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

int main(void)
{
    float x1, y1
    printf("Coordonnees du vecteur v1 separees par un \";\" :\n");
    scanf("%f ;%f", &x1, &y1);

    float x2, y2;
    printf("Coordonnees du vecteur v2 separees par un \";\" :\n");
    scanf("%f ;%f", &x2, &y2);

    float dot_product = x1 * x2 + y1 * y2;
    printf("Produit scalaire : %f\n", dot_product);
    if (dot_product == 0.0)
        printf("Les vecteurs sont orthogonaux.\n");
}
```

Ce programme risque de ne pas bien détecter l'orthogonalité de certains vecteurs, car le test d'égalité à 0 avec les virgules flottantes pourrait mal fonctionner. En effet, pour deux vecteurs orthogonaux, les erreurs de calcul en virgule flottante pourraient amener à un produit scalaire calculé très proche mais cependant différent de zéro. On peut corriger ce problème en modifiant le test pour vérifier si le produit scalaire est très petit, par exemple compris entre `-0.000001` et `+0.000001` :

```c
if (dot_product >= -1E-6 && dot_product <= 1E-6)
```

Ce qui peut encore s'écrire en utilisant la fonction valeur absolue :

```c
if (fabs(dot_product) <= 1E-6)
```

Exercice

Votre collègue n'a pas cessé de se plaindre de crampes... aux doigts... Il a écrit le programme suivant avant de prendre congé pour se rendre chez son médecin.

Grâce à votre esprit affuté et votre oeil perçant, vous identifiez 13 erreurs. Lesquelles sont-elles ?

```c
#include <std_io.h>
#jnclude <stdlib.h>
INT Main()
{
```

(suite sur la page suivante)

(suite de la page précédente)

```
int a, sum;
printf("Addition de 2 entiers a et b.\n");

printf("a: ")
scanf("%d", a);

printf("b: ");
scanf("%d", &b);

/* Affichage du résultat
somme = a - b;
Printf("%d + %d = %d\n", a, b, sum);

retturn EXIT_FAILURE;
}
}
```

Une fois la correction effectuée, vous utilisez l'outil de `diff` pour montrer les différences :

```
1,3c1,3
<         #include <stdio.h>
<         #include <stdlib.h>
<         int main()
---
>         #include <std_io.h>
>         #jnclude <stdlib.h>
>         INT Main()
5c5
<         int a, b, sum;
---
>         int a, sum;
9c9
<         scanf("%d", &a);
---
>         scanf("%d", a);
14,16c14,16
<         /* Affichage du résultat */
<         sum = a + b;
<         printf("%d + %d = %d\n", a, b, sum);
---
>         /* Affichage du résultat
>         somme = a - b;
>         Printf("%d + %d = %d\n", a, b, sum);
18c18,19
<         return EXIT_SUCCESS;
---
>         retturn EXIT_FAILURE;
>         }
```

Exercice

Considérez le programme suivant :

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

int main(void)
{
    float a;
    printf("a = ");
    scanf("%f", &a);

    float b;
    printf("b = ");
    scanf("%f", &b);

    float x;
    printf("x = ");
    scanf("%f", &x);

    float y = a * x + b;

    printf("y = %f\n", y);

    return 0;
}
```

1. À quelle ligne commence l'exécution de ce programme ?
2. Dans quel ordre s'exécutent les instructions ?
3. Décrivez ce que fait ce programme étape par étape
4. Que verra l'utilisateur à l'écran ?
5. Quelle est l'utilité de ce programme ?

1. Ligne 6
2. C est un langage impératif, l'ordre est séquentiel du haut vers le bas
3. **Les étapes sont les suivantes :**
    1. Demande de la valeur de `a` à l'utilisateur
    2. Demande de la valeur de `b` à l'utilisateur
    3. Demande de la valeur de `x` à l'utilisateur
    4. Calcul de l'image affine de `x` (équation de droite)
    5. Affichage du résultat
4. **Que verra l'utilisateur à l'écran ?**
    1. Il verra `y = 12` pour `a = 2; x = 5; b = 2`
5. **Quelle est l'utilité de ce programme ?**
    1. Le calcul d'un point d'une droite

Exercice

L'exercice précédent souffre de nombreux défauts. Sauriez-vous les identifier et perfectionner l'implémentation de ce programme ?

Citons les défauts de ce programme :

- Le programme ne peut pas être utilisé avec les arguments, uniquement en mode interactif
- Les invités de dialogue `a = ``, ``b = `` ne sont pas clair, ``a` et `b` sont associés à quoi ?
- La valeur de retour n'est pas exploitable directement.
- Le nom des variables utilisé n'est pas clair.
- Aucune valeur par défaut.

Une solution possible serait :

Code source 9.1 – linear.c

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

int main(int argc, char* argv[])
{
    float x;
    float offset;
    float slope;

    if (argc > 2) {
        offset = atof(argv[1]);
        slope = atof(argv[2]);
    } else {
        float offset_default = 0.;
        printf("Offset? [%f]: ", offset_default);
        if (!scanf("%f", &offset)) {
            offset = offset_default;
        }

        float slope_default = 1.;
        printf("Pente? [%f]: ", slope_default);
        if (!scanf("%f", &slope)) {
            slope = slope_default;
        }
    }

    if (argc == 2 || argc > 3) {
        slope = atof(argv[argc == 2 ? 2: 3]);
    } else {
        float x_default = 0;
        printf("x (abscisse) [%f]:", x_default);
        if (!scanf("%f", &x)) {
            x = x_default;
        }
```

(suite sur la page suivante)

(suite de la page précédente)

```
            }

            printf("%f\n", slope * x + offset);

            return 0;
        }
```

Exercice

Écrivez un programme demandant deux réels `tension` et `résistance`, et affichez ensuite le `courant`. Prévoir un test pour le cas où la résistance serait nulle. Exercice

Considérons le programme suivant :

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

int main()
{
    printf("Quel angle mesurez-vous en visant le sommet du bâtiment (en degrés): "
    float angle_degre;
    scanf("%f", &angle_degrees);
    float angle_radian = angle_degrees * M_PI / 45.;


    printf("À quelle distance vous trouvez vous du bâtiment (en mètres): ");
    float distance;
    scanf("%f", &distance);

    float height = distance / tan(angle_radian);
    printf("La hauteur du bâtiment est : %g mètres.\n", height);

    return 0;
}
```

1. Que fait le programme étape par étape ?
2. Que verra l'utilisateur à l'écran ?
3. À quoi sert ce programme ?
4. Euh, mais ? Ce programme comporte des erreurs, lesquelles ?
5. Implémentez-le et testez-le.

Exercice

Hyperloop (aussi orthographié **Hyperl∞p**) est un projet ambitieux d'Elon Musk visant à construire un moyen de transport ultra rapide utilisant des capsules voyageant dans

un tube sous vide. Ce projet est analogue à celui étudié en suisse et nommé Swissmetro, mais abandonné en 2009.

Néanmoins, les ingénieurs suisses avaient à l'époque écrit un programme pour calculer, compte tenu d'une vitesse donnée, le temps de parcours entre deux villes de Suisse.

Écrire un programme pour calculer la distance entre deux villes de suisse parmi lesquelles proposées sont :

- Genève
- Zürich
- Bâle
- Bern
- St-Galle

Considérez une accélération de 0.5 g pour le calcul de mouvement, et une vitesse maximale de 1220 km/h.

Chapitre 10

Fonctions

A l'époque d'Apollo 11, les fonctions n'existaient pas, le code n'était qu'une suite monolithique d'instruction ésotérique dont les sources du Apollo Guidance Computer ont été publiées sur GitHub. Le langage est l'assembler yaYUL dispose de sous-routines, ou procédures qui sont des fonctions sans paramètres. Ce type de langage est procédural.

Néanmoins, dans ce langage assembleur étrange, le code reste **monolithique** et toutes les variables sont globales.

Un programme convenablement **structuré** est découpé en éléments fonctionnels qui disposent pour chacun d'entrées et de sorties. De la même manière qu'un téléencéphale hautement développé et son pouce préhenseur aime organiser sa maison en pièces dédiées à des occupations particulières et que chaque pièce dispose de rangements assignés les uns à des assiettes, les autres à des couverts, le développeur organisera son code en blocs fonctionnels et cherchera à minimiser les effets de bords.

Une fonction est donc un ensemble de code exécutable délimité du programme principal et disposant :

- D'un identifiant unique
- D'une valeur de retour
- De paramètres d'appel

L'utilisation des fonctions permet :

- De décomposer un programme complexe en tâches plus simples
- De réduire la redondance de code
- De maximiser la réutilisation du code
- De s'abstraire des détails d'implémentation
- D'augmenter la lisibilité du code
- D'accroître la traçabilité à l'exécution

En revanche, une fonction apporte quelques désavantages qui à l'échelle des ordinateurs moderne sont parfaitement négligeables. L'appel à une fonction ou sous-routine requiert du housekeeping, qui se compose d'un prélude et d'un aboutissant et dans lequel le contexte doit être sauvegardé.

Fig. 10.1 – Margaret Hamilton la directrice du projet Apollo Guidance Computer (AGC) à côté du code du projet.

10.1 Conventions d'appel

Dans le Voyage de Chihiro ( ) de Hayao Miyazaki, le vieux Kamaji ( ) travaille dans la chaudière des bains pour l'alimenter en charbon et préparer les décoctions d'herbes pour parfumer les bains des clients.

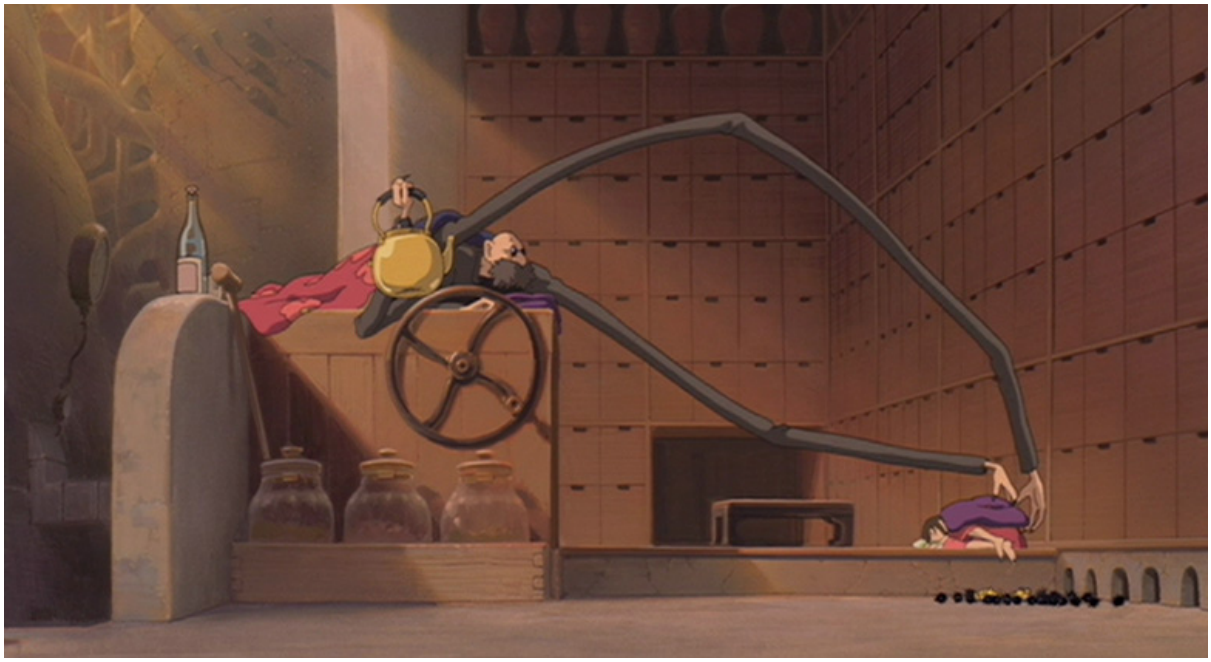


Fig. 10.2 – Le vieux Kamaji et ses bras extensibles.

Je vous propose bâtir une métaphore du changement de contexte en s'inspirant de cette illustration. Les murs de la chaudière sont emplis de casiers contenant différentes herbes, ces casiers peuvent être apparentés à la mémoire de l'ordinateur, et les différentes herbes, des types de données différents. De son pupitre Kamaji dispose de plusieurs mortiers dans lequel il mélange les herbes ; ils sont à l'instar de l'ALU d'un ordinateur le siège d'opérations transformant, à l'aide du pilon, plusieurs entrées en une seule sortie : le mélange d'herbes servant à la décoction. Bien qu'il ait six bras et afin de s'éviter des manipulations inutiles, il garde de petites réserves d'herbes à côté de son pupitre dans de petits casiers, similaires aux registres du processeur.

Il profite de son temps libre, pendant que les bains sont fermés pour préparer certains mélanges d'herbes les plus populaires et il place ce stock dans un casier du mur. Préparer un mélange est très similaire à un programme informatique dans lequel une suite d'opération représente une recette donnée. Le vieux Kamaji à une très grande mémoire, et il ne dispose pas de livre de recettes, mais vous, moi, n'importe qui, aurions besoin d'instructions claires du type :

```
AUTUMN_TONIC_TEA :

  MOVE  R1 @B4      # Déplace de la grande ortie du casier B4 au␣
↪registre R1
  MOVE  R2 @A8      # Déplace la menthe verte (Mentha spicata) du␣
↪casier A8 au registre R2
```

(suite sur la page suivante)

(suite de la page précédente)

```
  MOVE  R3 @C7      # Déplace le gingembre du casier C7 au registre␣
↪R3
  ...
  CHOP  R4 R3, FINE # Coupe très finement le gingembre et le place␣
↪dans R4
  ...
  LEAV  R2 R5       # Détache les feuilles des tiges de la menthe␣
↪verte, place les feuilles en R5
  ...
  ADD   R8 R1 R5    # Pilonne le contenu de R1 et R2 et place dans␣
↪R8
  ADD   R8 R8 R4
  ...
  STO   R8 @F6      # Place le mélange d'herbe automnale tonic dans␣
↪le casier F6
```

Souvent, le vieux Kamaji répète les mêmes suites d'opération et ce, peu importe les herbes qu'il manipule, une fois placées dans les petits casiers (registres), il pourrait travailler les yeux fermés.

On pourrait résumer ce travail par une fonction C, ici prenant un rhizome et deux herbes en entrée et générant un mélange en sortie.

```
blend slice_and_blend(rootstock a, herb b, herb c);
```

Pour des recettes complexes, il se pourrait que la fonction `slice_and_blend` soit appelée plusieurs fois à la suite, mais avec des ingrédients différents. De même que cette fonction fait appel à une autre fonction plus simple tel que `slice` (découper) ou `blend_together` (incorporer).

Et le contexte dans tout cela ? Il existe selon le langage de programmation et l'architecture processeur ce que l'on appelle les conventions d'appel. C'est à dire les règles qui régissent les interactions entre les appels de fonctions. Dans notre exemple, on adoptera peut-être la convention que n'importe quelle fonction trouvera ses ingrédients d'entrées dans les casiers R1, R2 et R3 et que le résultat de la fonction, ici le *blend*, sera placé dans le casier R8. Ainsi peu importe les herbes en entrée, le vieux Kamaji peut travailler les yeux fermés, piochant simplement dans R1, R2 et R3.

On observe néanmoins dans la recette évoquée plus haut qu'il utilise d'autres casiers, R4, et R5. Il faut donc faire très attention à ce qu'une autre fonction peut-être la fonction `slice`, n'utilise pas dans sa propre recette le casier R5, car sinon, c'est la catastrophe.

```
herb slice(herb a);
```

Kamaji entrepose temporairement les feuilles de menthe verte dans R5 et lorsqu'il en a besoin, plus tard, après avoir découpé les fleurs de molène que R5 contient des tiges d'une autre plante.

Dans les conventions d'appel, il faut donc également donner la responsabilité à quelqu'un de ne pas utiliser certains casiers, ou alors d'en sauvegarder ou de restaurer le contenu

au début et à la fin de la recette. Dans les conventions d'appel, il y en réalité plusieurs catégories de registres :

— ceux utilisés pour les paramètres de la fonction,
— ceux utilisés pour les valeurs de retour,
— ceux qui peuvent être utilisés librement par une fonction (la sauvegarde est à la charge du *caller*, la fonction qui appelle une autre fonction),
— ceux qui doivent être sauvegardés par le *callee* (la fonction qui est appelée).

En C, ce mécanisme est parfaitement automatique, le programmeur n'a pas à ce soucier du processeur, du nom des registres, de la correspondance entre le nom des herbes et le casier ou elles sont entreposées. Néanmoins, l'électronicien développeur, proche du matériel doit parfois bien comprendre ces mécanismes et ce qu'ils coûtent (en temps et en place mémoire) à l'exécution d'un programme.

10.1.1 Overhead

L'appel de fonction coûte à l'exécution, car avant chaque fonction, le compilateur ajoute automatiquement des instructions de sauvegarde et de restauration des registres utilisés :

Ce coût est faible, très faible, un ordinateur fonctionnant à 3 GHz et une fonction complexe utilisant tous les registres disponibles, mettons 10 registres, consommera entre l'appel de la fonction et son retour 0.000'000'003 seconde, ça va, c'est raisonnable. Sauf que, si la fonction ne comporte qu'une seule opération comme ci-dessous, l'overhead sera aussi plus faible.

```
int add(int a, int b) {
    return a + b;
}
```

10.1.2 Stack

En français la pile d'exécution, est un emplacement mémoire utilisé pour sauvegarder les registres du processeur entre les appels de fonctions, sauvegarder les adresses de retour des fonctions qui sont analogue à sauvegarder le numéro de page du livre de recettes : p 443. Recette du Bras de Vénus : commencer par réaliser une génoise de 300g (p. 225). Une fois la génoise terminée, il faut se rappeler de retourner à la page 443. Enfin le *stack* est utilisé pour mémoriser les paramètres des fonctions supplémentaires qui ne tiendraient pas dans les registres d'entrées. La convention d'appel de la plupart des architectures prévoie généralement 3 registres pour les paramètres d'entrées, ci bien qu'une fonction à 4 paramètres, pourrait bien aussi utiliser le *stack* :

```
double quaternion_norm(double a1, double b1, double c1, double d1);
```

La pile d'exécution est, comme son nom l'indique, une pile sur laquelle sont empilés et dépilés les éléments au besoin. A chaque appel d'une fonction, la valeur des registres à sauvegarder est empilée et au retour d'une fonction les registres sont dépilés si bien que la fonction d'appel retrouve le *stack* dans le même état qu'il était avant l'appel d'une fonction enfant.

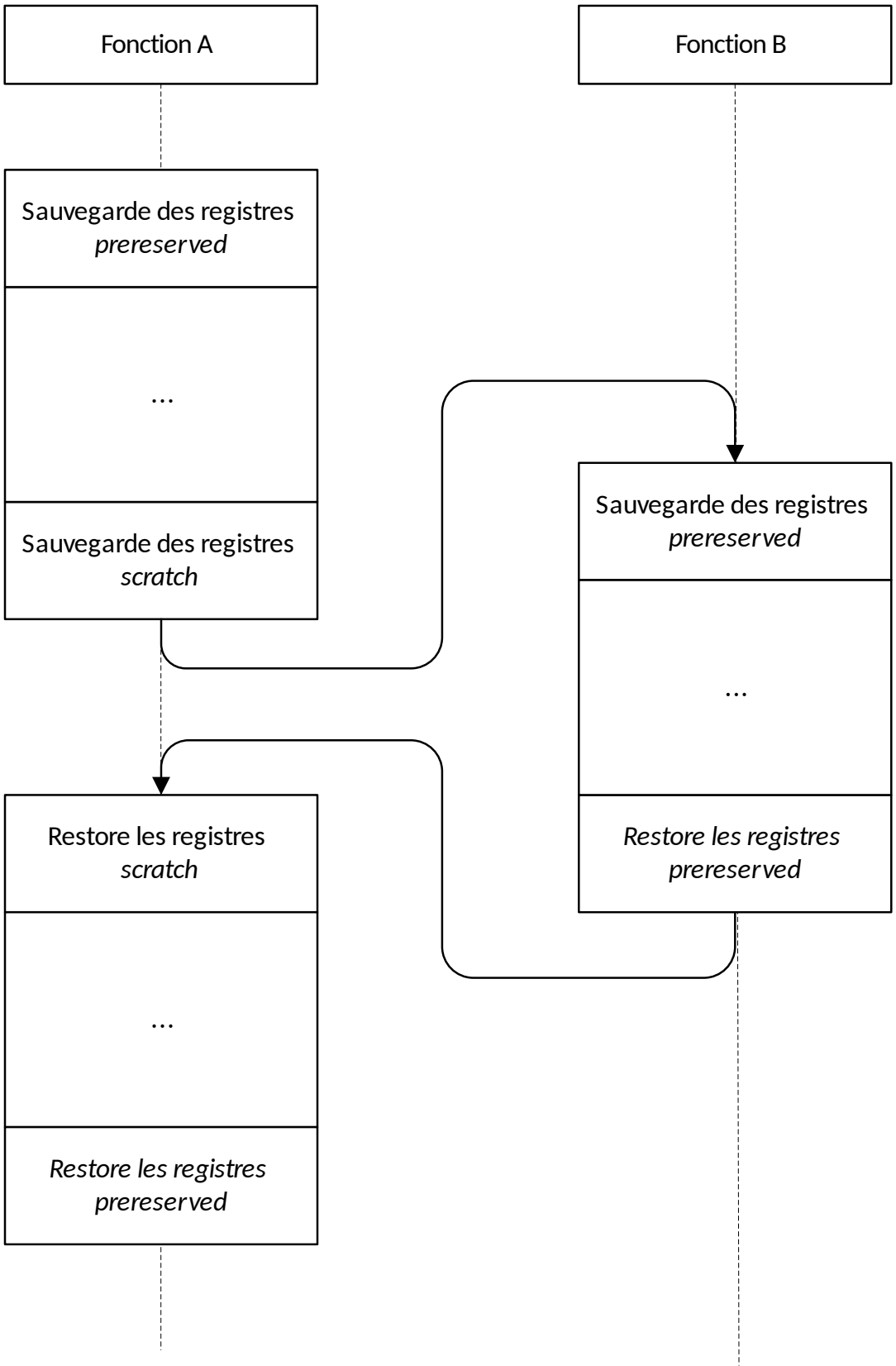


Fig. 10.3 – Sauvegarde des registres du processeur et convention d'appel de fonction.

10.2 Prototype

Le prototype d'une fonction est son interface avec le monde extérieur. Il déclare la fonction, son type de retour et ses paramètres d'appel. Le prototype est souvent utilisé dans un fichier d'en-tête pour construire des bibliothèques logicielles. La fonction `printf` que nous ne cessons pas d'utiliser voit son prototype résider dans le fichier `<stdio.h>` et il est déclaré sous la forme :

```
int printf(const char* format, ...);
```

Notons qu'il n'y a pas d'accolades ici.

Rappelons-le, C est un langage impératif et déclaratif, c'est-à-dire que les instructions sont séquentielles et que les déclarations du code sont interprétées dans l'ordre ou elles apparaissent. Si bien si je veux appeler la fonction `make_coffee`, il faut qu'elle ait été déclarée avant, c'est à dire plus haut.

Le code suivant fonctionne :

```
int make_coffee(void) {
    printf("Please wait...\n)";
}

int main(void) {
    make_coffee();
}
```

Mais celui-ci ne fonctionnera pas, car `make_coffee` n'est pas connu au moment de l'appel :

```
int main(void) {
    make_coffee();
}

int make_coffee(void) {
    printf("Please wait...\n)";
}
```

Si pour une raison connue seule du développeur on souhaite déclarer la fonction après `main`, on peut ajouter le prototype de la fonction avant cette dernière. C'est ce que l'on appelle la déclaration avancée ou forward declaration.

```
int make_coffee(void);

int main(void) {
    make_coffee();
}

int make_coffee(void) {
```

(suite sur la page suivante)

(suite de la page précédente)

```
    printf("Please wait...\n");
}
```

Un **prototype** de fonction diffère de son **implémentation** par fait qu'il ne dispose pas du code, mais simplement sa définition, permettant au compilateur d'établir les *conventions d'appel* de la fonction.

10.3 Syntaxe

La syntaxe d'écriture d'une fonction peut être assez compliquée et la source de vérité est issue de la grammaire du langage, qui n'est pas nécessairement accessible au profane. Or, depuis **C99**, une fonction prend la forme :

```
<storage-class> <return-type> <function-name> ( <parameter-type>
↪<parameter-name>, ... )
```

<storage-class> Classe de stockage, elle n'est pas utile à ce stade du cours, nous aborderons plus tard les mots clés `extern`, `static` et `inline`.

<return-type> Le type de retour de la fonction, s'agit-il d'un `int`, d'un `float` ? Le type de retour est anonyme, il n'a pas de nom et ce n'est pas nécessaire.

<function-name> Il s'agit d'un *identifiant* qui représente le nom de la fonction. Généralement on préfère choisir un verbe, quelquefois associé à un nom : `compute_norm`, `make_coffee`, ... Néanmoins lorsqu'il n'y a pas d'ambigüité, on peut choisir des termes plus simples tels que `main`, `display` ou `dot_product`.

<parameter-type> <parameter-name> La fonction peut prendre en paramètre zéro à plusieurs paramètres chaque paramètre est défini par son type et son nom tel que : `double real, double imag` pour une fonction qui prendrait en paramètre un nombre complexe.

Après la fermeture de la parenthèse de la liste des paramètres, deux possibilités :

Prototype On clos la déclaration avec un `;`

Implémentation On poursuit avec l'implémentation du code `{ ... }`

10.3.1 void

Le type `void` est à une signification particulière dans la syntaxe d'une fonction. Il peut être utilisé de trois manières différentes :

— **Pour indiquer l'absence de valeur de retour :**

```
void foo(int a, int b);
```

— **Pour indiquer l'absence de paramètres :**

```c
int bar(void);
```

— **Pour indiquer que la valeur de retour n'est pas utilisée par le parent :**

```c
(void) foo(23, 11);
```

La déclaration suivante est formellement fausse, car la fonction ne possède pas un prototype complet. En effet, le nombre de paramètres n'est pas contraint et le code suivant est valide au sens de **C99**.

```c
void dummy() {}

int main(void) {
    dummy(1, 2, 3);
    dummy(120, 144);
}
```

Aussi, il est impératif de toujours écrire des prototypes complets et d'explicitement utiliser `void` lorsque la fonction ne prend aucun paramètre en entrée. Si vous utilisez un compilateur C++, une déclaration incomplète génèrera une erreur.

10.4 Paramètres

Comme nous l'avons vu plus haut, pour de meilleures performances à l'exécution, il est préférable de s'en tenir à un maximum de trois paramètres, c'est également plus lisible pour le développeur, mais rien n'empêche d'en avoir plus.

En plus de cela, les paramètres peuvent être passés de deux manières :

— Par valeur
— Par référence

En C, fondamentalement, tous les paramètres sont passés par valeur, c'est-à-dire que la valeur d'une variable est copiée à l'appel de la fonction. Dans l'exemple suivant, la valeur affichée sera bel et bien `33` et non `42`

```c
void alter(int a) {
    a = a + 9;
}

void main(void) {
    int a = 33;
    alter(a);
    printf("%d\n", a);
}
```

Dans certains cas, on souhaite utiliser plus d'une valeur de retour et l'on peut utiliser un tableau. Dans l'exemple suivant, la valeur affichée sera cette fois-ci `42` et non `33`.

```c
void alter(int array[]) {
    array[0] += 9;
}

void main(void) {
    int array[] = {33, 34, 35};
    alter(array);
    printf("%d\n", array[0]);
}
```

Par abus de langage et en comparaison avec d'autres langages de programmation, on appellera ceci un passage par référence, car ce n'est pas une copie du tableau qui est passée à la fonction `alter`, mais seulement une référence sur ce tableau.

En des termes plus corrects, mais nous verrons cela au chapitre sur les pointeurs, c'est bien un passage par valeur dans lequel la valeur d'un pointeur sur un tableau est passée à la fonction `alter`.

Retenez simplement que lors d'un passage par référence, on cherche à rendre la valeur passée en paramètre modifiable par le *caller*.

10.5 Récursion

La récursion, caractère d'un processus, d'un mécanisme récursif, c'est à dire qui peut être répété un nombre indéfini de fois par l'application de la même règle, est une méthode d'écriture dans laquelle une fonction s'appelle elle même.

Au chapitre sur les fonctions, nous avions donné l'exemple du calcul de la somme de la suite de fibonacci jusqu'à `n` :

```c
int fib(int n)
{
    int sum = 0;
    int t1 = 0, t2 = 1;
    int next_term;
    for (int i = 1; i <= n; i++)
    {
        sum += t1;
        next_term = t1 + t2;
        t1 = t2;
        t2 = next_term;
    }
    return sum;
}
```

Il peut sembler plus logique de raisonner de façon récursive. Quelque soit l'itération à laquelle l'on soit, l'assertion suivante est valable :

fib(n) == fib(n - 1) + fib(n - 2)

Donc pourquoi ne pas réécrire cette fonction en employant ce caractère récursif ?

```c
int fib(int n)
{
    if (n < 2) return 1;
    return fib(n - 1) + fib(n - 2);
}
```

Le code est beaucoup plus simple a écrire, et même à lire. Néanmoins cet algorithme est notoirement connu pour être mauvais en terme de performance. Calculer `fib(5)` revient à la chaîne d'appel suivant.

Cette chaîne d'appel représente le nombre de fois que `fib` est appelé et à quel niveau elle est appelée. Par exemple `fib(4)` est appelé dans `fib(5)` :

```
fib(5)
    fib(4)
        fib(3)
            fib(2)
            fib(1)
        fib(2)
            fib(1)
            fib(0)
    fib(3)
        fib(2)
            fib(1)
            fib(0)
        fib(1)
```

Si l'on somme le nombre de fois que chacune de ces fonctions est appelée :

```
fib(5)   1x
fib(4)   1x
fib(3)   2x
fib(2)   3x
fib(1)   4x
fib(0)   2x
-----------
fib(x)  13x
```

Pour calculer la somme de fibonacci, il faut appeler 13 fois la fonction. On le verra plus tard mais la complexité algoritmique de cette fonction est dite $O(2^n)$. C'est à dire que le nombre d'appels suit une relation exponentielle. La réelle complexité est donnée par la relation :

$$T(n) = O\left(\frac{1+\sqrt{(5)}}{2}^{n}\right) = O\left(1.6180^n\right)$$

Ce terme 1.6180 est appelé le nombre d'or.

Ainsi pour calculer fib(100) il faudra sept cent quatre-vingt-douze trillions soixante-dix mille huit cent trente-neuf billions huit cent quarante-huit milliards trois cent soixante-

quatorze millions neuf cent douze mille deux cent quatre-vingt-douze appels à la fonction *fib* (792'070'839'848'374'912'292). Pour un processeur Core i7 (2020) capable de calculer environ 100 GFLOPS (milliards d'opérations par seconde), il lui faudra tout de même 251 ans.

En revanche, dans l'approche itérative, on constate qu'une seule boucle `for`. C'est à dire qu'il faudra seulement 100 itérations pour calculer la somme.

Généralement les algorithmes récursifs (s'appelant eux même) sont moins performants que les algorithmes itératifs (utilisant des boucles). Néanmoins il est parfois plus facile d'écrire un algorithme récursif.

Notons que tout algorithme récursif peut être écrit en un algorithme itératif, mais ce n'est pas toujours facile.

10.6 Mémoïsation

En informatique la mémoïsation est une technique d'optimisation du code souvent utilisée conjointement avec des algorithmes récursifs. Cette technique est largement utilisée en programmation dynamique.

Nous l'avons vu précédemment, l'algorithme récursif du calcul de la somme de la suite de Fibonacci n'est pas efficace du fait que les mêmes appels sont répétés un nombre inutiles de fois. La parade est de mémoriser pour chaque appel de `fib`, la sortie correspondante à l'entrée.

Dans cet exemple nous utiliserons un mécanisme composé de trois fonctions :

- `int memoize(Cache *cache, int input, int output)`
- `bool memoize_has(Cache *cache, int input)`
- `int memoize_get(Cache *cache, int input)`

La première fonction mémorise la valeur de sortie `output` liée à la valeur d'entrée `input`. Pour des raisons de simplicité d'utilsation, la fonction retourne la valeur de sortie `output`.

La seconde fonction `memoize_has` vérifie si une valeur de correspondance existe pour l'entrée `input`. Elle retourne `true` en cas de correspondance et `false` sinon.

La troisième fonction `memoize_get` retourne la valeur de sortie correspondante à la valeur d'entrée `input`.

Notre fonction récursive sera ainsi modifiée comme suit :

```
int fib(int n)
{
    if (memoize_has(n)) return memoize_get(n);
    if (n < 2) return 1;
    return memoize(n, fib(n - 1) + fib(n - 2));
}
```

Quant aux trois fonctions utilitaires, voici une proposition d'implémentation. Notons que cette implémentation est très élémentaire et n'est valable que pour des entrées inférieures à 1000. Il sera possible ultérieurement de perfectionner ces fonctions, mais nous aurons

pour cela besoin de concepts qui n'ont pas encore été abordés, tels que les structures de données complexes.

```c
#define SIZE 1000

bool cache_input[SIZE] = { false };
int cache_output[SIZE];

int memoize(int input, int output) {
    cache_input[input % SIZE] = true;
    cache_output[input % SIZE] = output;
    return output;
}

bool memoize_has(int input) {
    return cache_input[input % SIZE];
}

int memoize_get(int input) {
    return cache_output[input % SIZE];
}
```

Exercice

Écrire une fonction `mean` qui reçoit 3 paramètres réels et qui retourne la moyenne.

```c
double mean(double a, double b, double c) {
    return (a + b + c) / 3.;
}
```

Exercice

Écrire une fonction `min` qui reçoit 3 paramètres réels et qui retourne la plus petite valeur.

```c
double min(double a, double b, double c) {
    double min_value = a;
    if (b < min_value)
        min_value = b;
    if (c < min_value)
        min_value = c;
    return min_value;
}
```

Une manière plus compacte, mais moins lisible serait :

```c
double min(double a, double b, double c) {
    return (a = (a < b ? a : b)) < c ? a : c;
}
```

Exercice

On considère le cas d'une caisse automatique de parking. Cette caisse délivre des tickets au prix unique de CHF 0.50 et dispose d'un certain nombre de pièces de 10 et 20 centimes pour le rendu de monnaie.

Dans le code du programme, les trois variables suivantes seront utilisées :

```
// Available coins in the parking ticket machine
unsigned int ncoin_10, ncoin_20;

// How much money the user inserted into the machine (in cents)
unsigned int amount_payed;
```

Écrivez l'algorithme de rendu de la monnaie tenant compte du nombre de pièces de 10 et 20 centimes restants dans l'appareil. Voici un exemple du fonctionnement du programme :

```
$ echo "10 10 20 20 20" | ./ptm 30 1
ticket
20
10
```

Le programme reçois sur `stdin` les pièces introduites dans la machine. Les deux arguments passés au programme `ptm` sont 1. le nombre de pièces de 10 centimes disponibles et 2. le nombre de pièces de 20 centimes disponibles. `stdout` contient les valeurs rendue à l'utilisateur. La valeur `ticket` correspond au ticket distribué.

Le cas échéant, s'il n'est possible de rendre la monnaie, aucun ticket n'est distribué et l'argent donné est rendu.

Voici une solution partielle :

```
#define TICKET_PRICE 50

void give_coin(unsigned int value) { printf("%d\n", value); }
void give_ticket(void) { printf("ticket\n"); }

bool no_ticket = amount_payed < TICKET_PRICE;

int amount_to_return = amount_payed - TICKET_PRICE;
do {
    while (amount_to_return > 0) {
        if (amount_to_return >= 20 && ncoin_20 > 0) {
            give_coin(20);
            amount_to_return -= 20;
            ncoin_20--;
        } else if (amount_to_return >= 10 && ncoin_10 > 0) {
            give_coin(10);
            amount_to_return -= 10;
            ncoin_10--;
        } else {
            no_ticket = true;
            break;
```

(suite sur la page suivante)

(suite de la page précédente)

```
        }
    }
} while (amount_to_return > 0);

if (!no_ticket) {
    give_ticket();
}
```

Exercice

Considérons le programme suivant :

```
int f(float x) {
    int i;
    if (x > 0.0)
        i = (int)(x + 0.5);
    else
        i = (int)(x - 0.5);
    return i;
}
```

Quel sont les types et les valeurs retournées par les expressions ci-dessous ?

```
f(1.2)
f(-1.2)
f(1.6)
f(-1.6)
```

Quel est votre conclusion sur cette fonction ? Exercice

Le programme suivant compile sans erreurs graves mais ne fonctionne pas correctement.

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

long get_integer()
{
    bool ok;
    long result;
    do
    {
        printf("Enter a integer value: ");
        fflush(stdin); // Empty input buffer
        ok = (bool)scanf("%ld", &result);
        if (!ok)
            printf("Incorrect value.\n");
    }
    while (!ok);
    return result;
```

(suite sur la page suivante)

(suite de la page précédente)

```
}

int main(void)
{
    long a = get_integer;
    long b = get_integer;

    printf("%d\n", a + b);
}
```

Quel est le problème ? A titre d'information voici ce que le programme donne, notez que l'invité de saisie n'est jamais apparue :

```
$ ./sum
8527952
```

Chapitre 11

Types composites

11.1 Tableaux

Les tableaux (*arrays*) représentent une séquence finie d'éléments d'un type donné que l'on peut accéder par leur position (indice) dans la séquence. Un tableau est par conséquent une liste indexée de variable du même type.

L'opérateur crochet `[]` est utilisé à la fois pour le déréférencement (accès à un indice du tableau) et pour l'assignation d'une taille à un tableau :

La déclaration d'un tableau d'entiers de dix éléments s'écrit de la façon suivante :

```
int array[10];
```

Par la suite il est possible d'accéder aux différents éléments ici l'élément 1 et 3 (deuxième et quatrième position du tableau) :

```
array[1];
array[5 - 2];
```

L'opérateur `sizeof` permet d'obtenir la taille d'un tableau en mémoire, mais attention, c'est la taille du tableau et non le nombre d'éléments qui est retourné. Dans l'exemple suivant `sizeof(array)` retourne $10 \cdot 4 = 40$ tandis que `sizeof(array[0])` retourne la taille d'un seul élément 4 ; et donc, `sizeof(array) / sizeof(array[0])` est le nombre d'éléments de ce tableau, soit 10.

```
uint32_t array(5);
size_t length = sizeof(array) / sizeof(array[0]);
```

Indication : L'index d'un tableau commence toujours à **0** et par conséquent l'index maximum d'un tableau de 5 éléments sera 4. Il est donc fréquent dans une boucle d'utiliser **<** et non **<=** :

```c
for(size_t i = 0; i < sizeof(array) / sizeof(array[0]); i++) {
   /* ... */
}
```

Une variable représentant un tableau est en réalité un pointeur sur ce tableau, c'est-à-dire la position mémoire à laquelle se trouvent les éléments du tableau. Nous verrons ceci plus en détail à la section Section 14. Ce qu'il est important de retenir c'est que lorsqu'un tableau est passé à une fonction comme dans l'exemple suivant, l'entier du tableau n'est pas passé par copie, mais seul une **référence** sur ce tableau est passée.

La preuve étant que le contenu du tableau peut être modifié à distance :

```c
void function(int i[5]) {
   i[2] = 12
}

int main(void) {
   int array[5] = {0};
   function(array);
   assert(array[2] == 12);
}
```

Un fait remarquable est que l'opérateur `[]` est commutatif. En effet, l'opérateur *crochet* est un sucre syntaxique :

Et cela fonctionne même avec les tableaux à plusieurs dimensions : Exercice

Écrire un programme qui lit la taille d'un tableau de cinquante entiers de 8 bytes et assigne à chaque élément la valeur de son indice.

```c
int8_t a[50];
for (size_t i = 0; i < sizeof(a) / sizeof(a[0]); i++) {
    a[i] = i;
}
```

Exercice

Soit un tableau d'entiers, écrire une fonction retournant la position de la première occurence d'une valeur dans le tableau.

Traitez les cas particuliers.

```c
int index_of(int *array, size_t size, int search);
```

```c
int index_of(int *array, size_t size, int search) {
    int i = 0;
    while (i < size && array[i++] != search);
    return i == size ? -1 : i;
}
```

Exercice

Considérant les déclarations suivantes :

```
#define LIMIT 10
const int twelve = 12;
int i = 3;
```

Indiquez si les déclarations suivantes (qui n'ont aucun lien entre elles), sont correcte ou non.

```
int t(3);
int k, t[3], l;
int i[3], l = 2;
int t[LIMITE];
int t[i];
int t[douze];
int t[LIMITE + 3];
float t[3, /* five */ 5];
float t[3]        [5];
```

Exercice

Soit deux tableaux *char u[]* et *char v[]*, écrire une fonction comparant leur contenu et retournant :

0 La somme des deux tableaux est égale.

-1 La somme du tableau de gauche est plus petite que le tableau de droite

1 La somme du tableau de droite est plus grande que le tableau de gauche

Le prototype de la fonction à écrire est :

```
int comp(char a[], char b[], size_t length);
```

```
int comp(char a[], char b[], size_t length) {
    int sum_a = 0, sum_b = 0;

    for (size_t i = 0; i < length; i++) {
        sum_a += a[i];
        sum_b += b[i];
    }

    return sum_b - sum_a;
}
```

Exercice

Dans le canton de Genève, il existe une tradition ancestrale : l'Escalade. En comémoration de la victoire de la république protestante sur les troupes du duc de Savoie suite à l'attaque lancée contre Genève dans la nuit du 11 au 12 décembre 1602 (selon le calendrier julien), une traditionnelle marmite en chocolat est brisée par l'ainé et le cadet après la récitation de la phrase rituelle "Ainsi périrent les ennemis de la République !".

Pour gagner du temps et puisque l'assemblée est grande, il vous est demandé d'écrire un programme pour identifier le doyen et le benjamin de l'assistance.

Un fichier contenant les années de naissance de chacun vous est donné, il ressemble à ceci :

```
1931
1986
1996
1981
1979
1999
2004
1978
1964
```

Votre programme sera exécuté comme suit :

```
$ cat years.txt | marmite
2004
1931
```

Exercice

Un indice magique d'un tableau `A[0..n-1]` est défini tel que la valeur `A[i] == i`. Compte tenu que le tableau est trié avec des entiers distincts (sans répétition), écrire une méthode pour trouver un indice magique s'il existe.

Exemple :

```
  0   1   2   3   4   5   6   7   8   9   10
┌───┬───┬───┬───┬───┬───┬───┬───┬───┬───┬───┐
│-90│-33│ -5│ 1 │ 2 │ 4 │ 5 │ 7 │ 10│ 12│ 14│
└───┴───┴───┴───┴───┴───┴───┴───┴───┴───┴───┘
                          ^
```

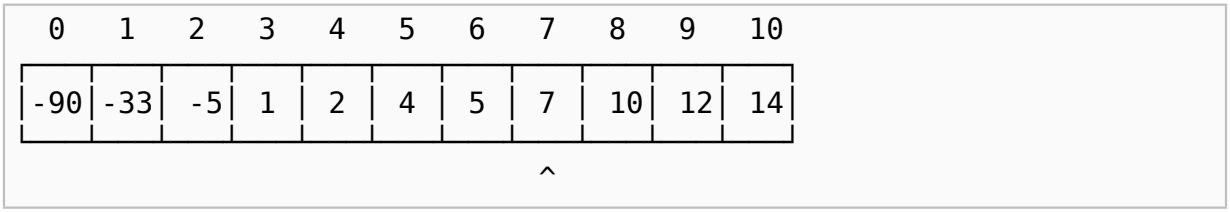
c

Une solution triviale consite à itérer tous les éléments jusqu'à trouver l'indice magique :

```c
int magic_index(int[] array) {
    const size_t size = sizeof(array) / sizeof(array[0]);

    size_t i = 0;

    while (i < size && array[i] != i) i++;

    return i == size ? -1 : i;
}
```

La complexité de cet algorithme est $O(n)$ or, la donnée du problème indique que le tableau est trié. Cela veut dire que probablement, cette information n'est pas donnée par hasard.

Pour mieux se représenter le problème prenons l'exemple d'un tableau :



La première valeur magique est `7`. Est-ce qu'une approche dichotomique est possible ?

Prenons le milieu du tableau `A[5] = 4`. Est-ce qu'une valeur magique peut se trouver à gauche du tableau ? Dans le cas le plus favorable qui serait :

On voit qu'il est impossible que la valeur se trouve à gauche car les valeurs dans le tableau sont distinctes et il n'y a pas de répétitions. La règle que l'on peut poser est `A[mid] < mid` où `mid` est la valeur mediane.

Il est possible de répéter cette approche de façon dichotomique :

```
int magic_index(int[] array) {
    return _magic_index(array, 0, sizeof(array) / sizeof(array[0]) -
↪ 1);
}

int _magic_index(int[] array, size_t start, size_t end) {
    if (end < start) return -1;
    int mid = (start + end) / 2;
    if (array[mid] == mid) {
        return mid;
    } else if (array[mid] > mid) {
        return _magic_index(array, start, mid - 1);
    } else {
        return _magic_index(array, mid + 1, end);
    }
}
```

11.1.1 Initialisation

Lors de la déclaration d'un tableau, le compilateur réserve un espace mémoire de la taille suffisante pour contenir tous les éléments du tableaux. La déclaration suivante :

```
int32_t even[6];
```

contient 6 entiers, chacuns d'une taille de 32-bits (4 bytes). L'espace mémoire réservé est donc de 24 bytes.

Compte tenu de cette déclaration, il n'est pas possible de connaître la valeur qu'il y a, par exemple, à l'indice 4 (`even[4]`), car ce tableau n'a pas été initialisé et le contenu mémoire est non prédictible puisqu'il peut contenir les vestiges d'un ancien programme ayant résidé dans cette région mémoire auparavant. Pour s'assurer d'un contenu il faut initialiser le tableau, soit affecter des valeurs pour chaque indice :

```
int32_t sequence[6];
sequence[0] = 4;
sequence[1] = 8;
sequence[2] = 15;
sequence[3] = 16;
sequence[4] = 23;
sequence[5] = 42;
```

Cette écriture n'est certainement pas la plus optimisée car l'initialisation du tableau n'est pas réalisée à la compilation, mais à l'exécution du programme ; et ce seront pas moins de six instructions qui seront nécessaires à initialiser ce tableau. L'initialisation d'un tableau utilise les accolades :

```
int32_t sequence[6] = {4, 8, 15, 16, 23, 42};
```

Dans cette dernière écriture, il existe une redondance d'information. La partie d'initialisation `{4, 8, 15, 16, 23, 42}` comporte six éléments et le tableau est déclaré avec six éléments `[6]`. Pour éviter une double source de vérité, il est ici possible d'omettre la taille du tableau :

```
int32_t sequence[] = {4, 8, 15, 16, 23, 42};
```

Notons que dans premier de ces deux cas, si un nombre inférieur à 6 éléments est initialisé, les autrs éléments seront initializés à **zéro**

```
int32_t sequence[6] = {4, 8, 15, 16 /* le reste vaudra zéro */ };
```

Il est également possible d'initialiser un tableau de façon explicite en utilisant une notation plus spécifique :

```
int32_t sequence[6] = {[0]=4, [1]=8, [2]=15, [3]=16, [4]=23, [5]=42}
↪;
```

Et naturellement il est possible d'omettre certaines valeurs, lesquelles seront initialisées à zéro par défaut. Dans l'exemple suivant les valeurs aux indices 1 à 4 vaudront zéro.

```
int32_t sequence[6] = {[0]=4, [5]=42};
```

Notons que lorsque que la notation `[]=` est utilisée, les valeurs qui suivent seront positionnées aux indices suivants :

```
int32_t sequence[6] = {[0]=4, 8, [3]=16, 23, 42};
```

Dans l'exemple ci-dessus `sequence[2]` vaudra zéro.

Notons qu'un type composé tel qu'un tableau ne peut pas être initialisé après sa déclaration. L'exemple suivant ne fonctionne pas :

```
int array[10];

// Erreur: l'initialisation tardive n'est pas autorisée.
array = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9};
```

11.1.2 Initialisation à zéro

Enfin, un sucre syntaxique `{0}` permet d'initialiser tout un tableau à zéro. En effet, la valeur 0 est inscrite à l'indice zéro, les autres valeurs sont par défaut initialisées à zéro si non mentionnées :

```
int32_t sequence[6] = {0};
```

Cette écriture est nécessaire pour les variables locales, car, nous verrons plus loin (c.f. Section 13) les variables globales sont placées dans le segment mémoire `.bss` et sont initialisées à zéro au démarrage du programme. Toute variable globale est donc initialisée à zéro par défaut.

11.1.3 Initialization à une valeur particulière

Cette écriture n'est pas normalisée **C99**, mais est généralement compatible avec la majorité des compilateurs.

```
int array[1024] = { [ 0 ... 1023 ] = -1 };
```

En **C99**, il n'est pas possible d'initialiser un type composé à une valeur unique. La manière traditionnelle reste la boucle itérative :

```
for (size_t i = 0; i < sizeof(array)/sizeof(array[0]); i++)
    array[i] = -1;
```

11.1.4 Tableaux non modifiables

À présent que l'on sait initialiser un tableau, il peut être utile de définir un tableau avec un contenu qui n'est pas modifiable. Le mot clé `const` est utilisé a cette fin.

```
const int32_t sequence[6] = {4, 8, 15, 16, 23, 42};
sequence[2] = 12; // Interdit !
```

Dans l'exemple ci-dessus, la seconde ligne génèrera l'erreur suivante :

```
error: assignment of read-only location ‘sequence[2]’
```

Notons que lors de l'utilisation de pointeurs, il serait possible, de façon détournée, de modifier ce tableau malgré tout :

```
int *p = sequence;
p[2] = 12;
```

Dans ce cas, ce n'est pas une erreur mais une alerte du compilateur qui survient :

```
warning: initialization discards ‘const’ qualifier from pointer
target type [-Wdiscarded-qualifiers]
```

11.1.5 Tableaux multi-dimensionnels

Il est possible de déclarer un tableau à plusieurs dimensions. Si par exemple on souhaite définir une grille de jeu du *tic-tac-toe* ou morpion, il faudra une grille de 3x3.

Pour ce faire, il est possible de définir un tableau de 6 éléments comme vu auparavant, et utiliser un artifice pour adresser les lignes et les colonnes :

```
char game[6] = {0};
int row = 1;
int col = 2;
game[row * 3 + col] = 'x';
```

Néanmoins, cette écriture n'est pas pratique et le langage C dispose du nécessaire pour alléger l'écriture. La grille de jeu sera simplement initialisée comme suit :

```
char game[3][3] = {0};
```

Jouer `x` au centre équivaut à écrire :

```
game[1][1] = 'x';
```

De la même façon il est possible de définir structure tri-dimensionnelles :

```
int volume[10][4][8];
```

L'initialisation des tableaux multi-dimensionnel est très similaire au tableaux standards mais il est possible d'utiliser plusieurs niveau d'accolades.

Ainsi le jeu de morpion suivant :

```
 o | x | x
---+---+---
 x | o | o
---+---+---
 x | o | x
```

Peut s'initialiser comme suit :

```
char game[][3] = {{'o', 'x', 'x'}, {'x', 'o', 'o'}, {'x', 'o', 'x'}}
 ↪;
```

Notons que l'écriture suivante est similaire, car un tableau multidimensionnel est toujours représenté en mémoire de façon linéaire, comme un tableau à une dimension :

```
char game[][3] = {'o', 'x', 'x', 'x', 'o', 'o', 'x', 'o', 'x'};
```

Exercice

Voici les dépenses de service annuelles d'un célèbre bureau de détectives privés :

Janvier	414.38	222.72	99.17	153.81
Février	403.41	390.61	174.39	18.11
Mars	227.55	73.86	291.08	416.55
Avril	220.20	342.25	139.45	86.98
Mai	13.46	172.66	252.33	265.32
Juin	259.37	378.72	173.02	208.43
Juillet	327.06	16.53	391.05	266.84
Août	50.82	3.37	201.71	170.84
Septembre	450.78	9.33	111.63	337.07
Octobre	434.45	77.80	459.46	479.17
Novembre	420.13	474.69	343.64	273.28
Décembre	147.76	250.73	201.47	9.75

Afin de laisser plus de temps aux détectives à résoudres des affaires, vous êtes mandaté pour écrire une fonction qui reçois en paramètre le tableau de réels ci-dessus formaté comme suit :

```
double accounts[][] = {
    {414.38, 222.72,  99.17, 153.81, 0},
    {403.41, 390.61, 174.39, 18.11,  0},
    {227.55,  73.86, 291.08, 416.55, 0},
    {220.20, 342.25, 139.45, 86.98,  0},
    {13.46 , 172.66, 252.33, 265.32, 0},
    {259.37, 378.72, 173.02, 208.43, 0},
    {327.06,  16.53, 391.05, 266.84, 0},
    {50.82 ,   3.37, 201.71, 170.84, 0},
    {450.78,   9.33, 111.63, 337.07, 0},
    {434.45,  77.80, 459.46, 479.17, 0},
    {420.13, 474.69, 343.64, 273.28, 0},
    {147.76, 250.73, 201.47, 9.75,   0},
    {  0,      0,      0,    0,      0}
};
```

Et laquelle complète les valeurs manquantes. Exercice

A l'instar de l'outil *pot de peinture* des éditeurs d'image, il vous est demandé d'implé-

menter une fonctionnalité similaire.

L'image est représentée par un tableau bi-dimensionnel contenant des couleurs indexées :

```
typedef enum { BLACK, RED, PURPLE, BLUE, GREEN YELLOW, WHITE }␣
↪Color;

#if 0 // Image declaration example
Color image[100][100];
#endif

boolean paint(Color* image, size_t rows, size_t cols, Color fill_
↪color);
```

Indication : Deux approches intéressantes sont possibles : **DFS** (Depth-First-Search) ou **BFS** (Breadth-First-Search), toutes deux récursives.

11.2 Chaînes de caractères

Une chaîne de caractères est représentée en mémoire comme une succession de bytes, chacuns représentant un caractère ASCII spécifique. La chaîne de caractère `hello` contient donc 5 caractères et sera stockée en mémoire sur 5 bytes. Une chaîne de caractère est donc équivalente à un tableau de `char`.

En C, un artifice est utilisé pour faciliter les opérations sur les chaînes de caractères. Tous les caractères de 1 à 255 sont utilisables sauf le 0 qui est utilisé comme sentinelle. Lors de la déclaration d'une chaîne comme ceci :

```
char str[] = "hello, world!";
```

Le compilateur ajoutera automatiquement un caractère de terminaison `'\0'` à la fin de la chaîne. Pour comprendre l'utilité, imaginons une fonction qui permet de compter la longueur de la chaîne. Elle aurait comme prototype ceci :

```
size_t strlen(const char str[]);
```

On peut donc lui passer un tableau dont la taille n'est pas définie et par conséquent, il n'est pas possible de connaître la taille de la chaîne passée sans le bénéfice d'une sentinelle.

```
size_t strlen(const char str[]) {
    size_t len = 0,
    while (str[len++] != '\0') {}
    return len;
}
```

Une chaîne de caractère est donc strictement identique à un tableau de `char`.

Ainsi une chaîne de caractère est initialisée comme suit :

```
char str[] = "Pulp Fiction";
```

La taille de ce tableau sera donc de 12 caractères plus une sentinelle `'\0'` insérée automatiquement. Cette écriture est donc identique à :

```
char str[] = {
    'P', 'u', 'l', 'p', ' ', 'F', 'i', 'c', 't', 'i', 'o', 'n', '\0'
};
```

11.2.1 Tableaux de chaînes de caractères

Un tableau de chaîne de caractères est identique à un tableau multidimensionnel :

```
char conjunctions[][10] = {
    "mais", "ou", "est", "donc", "or", "ni", "car"
};
```

Il est ici nécessaire de définir la taille de la seconde dimension, comme pour les tableaux. C'est à dire que la variable `conjunctions` aura une taille de 7x10 caractères et le contenu mémoire de `conjunctions[1]` sera équivalent à :

```
{'o', 'u', 0, 0, 0, 0, 0, 0, 0, 0}
```

D'ailleurs, ce tableau aurait pu être initialisé d'une tout autre façon :

```
char conjunctions[][10] = {
    'm', 'a', 'i', 's', 0, 0, 0, 0, 0, 0, 'o', 'u', 0, 0, 0,
    0, 0, 0, 0, 0, 'e', 's', 't', 0, 0, 0, 0, 0, 0 , 0, 'd',
    'o', 'n', 'c', 0, 0, 0, 0, 0 , 0, 'o', 'r', 0, 0, 0, 0,
    0, 0, 0, 0, 'n', 'i', 0, 0, 0, 0, 0, 0, 0, 0, 'c', 'a',
    'r', 0, 0, 0, 0, 0, 0, 0,
};
```

Notons que la valeur `0` est strictement identique au caractère 0 de la table ASCII `'\0'`. La chaîne de caractère `"mais"` aura une taille de 5 caractères, ponctuée de la sentinelle `\0`.

11.3 Structures

Les structures sont des déclarations spécifiques permettant de regrouper une liste de variables dans un même bloc mémoire et permettant de s'y référer à partir d'une référence commune. Historiquement le type `struct` a été dérivé de `ALGOL 68`. Il est également utilisé en C++ et est similaire à une classe.

Il faut voir une structure comme un container à variables qu'il est possible de véhiculer comme un tout.

La structure suivante décrit un agrégat de trois grandeurs scalaires formant un point tridimensionnel :

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

Il ne faut pas confondre l'écriture ci-dessus avec ceci, dans lequel il y a un bloc de code avec trois variables locales déclarées :

```
{
    double x;
    double y;
    double z;
};
```

En utilisant le mot-clé `struct` devant un bloc, les variables déclarées au sein de ce bloc ne seront pas réservées en mémoire. Autrement dit, il ne sera pas possible d'accéder à `x` puisqu'il n'existe pas de variable `x`. En revanche, un nouveau container contenant trois variable est défini, mais pas encore déclaré.

La structure ainsi déclarée n'est pas très utile telle quelle, en revanche elle peut-être utilisée pour déclarer une variable de type `struct` :

```
struct {
    double x;
    double y;
    double z;
} point;
```

A présent on a déclaré une variable `point` de type `struct` contenant trois éléments de type `double`. L'affectaction d'une valeur à cette variable utilise l'opérateur `.` :

```
point.x = 3060426.957;
point.y = 3192003.220;
point.z = 4581359.381;
```

Comme `point` n'est pas une primitive standard mais un container à primitive, il n'est pas correct d'écrire `point = 12`. Il est essentiel d'indiquer quel élément de ce container on souhaite accéder.

Ces coordonnées sont un clin d'oeil aux Pierres du Niton qui sont deux blocs de roche erratiques déposés par le glacier du Rhône lors de son retrait après la dernière glaciation. Les coordonnées sont exprimées selon un repère géocentré ; l'origine étant le centre de la terre. Ces pierres sont donc situées à 4.5 km du centre de la terre, et donc un sacré défi pour Axel Lidenbrock et son fulmicoton.

11.3.1 Structures nommées

L'écriture que l'on a vu initialement `struct { ... };` est appelée structure annonyme, c'est à dire qu'elle n'a pas de nom. Telle quelle elle ne peut pas être utilisée et elle ne sert donc pas à grand chose. En revanche, il est possible de déclarer une variable de ce type en ajoutant un identificateur à la fin de la déclaration `struct { ... } nom;`. Néanmoins la structure est toujours annonyme.

Le langage C prévoit la possibilté de nommer une structure pour une utilisation ultérieure en rajoutant un nom après le mot clé `struct` :

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

Pour ne pas confondre un nom de structure avec un nom de variable, on préférera un identificateur en capitales ou en écriture *camel-case*. Maintenant qu'elle est nommée, il est possible de déclarer plusieurs variables de ce type ailleurs dans le code :

```
struct Point foo;
struct Point bar;
```

Dans cet exemple, on déclare deux variables `foo` et `bar` de type `struct Point`. Il est donc possible d'accéder à `foo.x` ou `bar.z`.

Rien n'empêche de déclarer une structure nommée et d'également déclarer une variable par la même occasion :

```
struct Point {
    double x;
    double y;
    double z;
} foo;
struct Point bar;
```

Notons que les noms de structures sont stockés dans un espace de noms différent de celui des variables. C'est à dire qu'il n'y a pas de collision possible et qu'un identifiant de fonction ou de variable ne pourra jamais être comparé à un identifiant de structure. Aussi, l'écriture suivante, bien que perturbante, est tout à fait possible :

```
struct point { double x; double y; double z; };
struct point point;
point.x = 42;
```

11.3.2 Initialisation

Une structure se comporte à peu de chose près comme un tableau sauf que les éléments de la structure ne s'accèdent pas avec l'opérateur crochet `[]` mais avec l'opérateur `.`. Néanmoins une structure est représentée en mémoire comme un contenu linéaire. Notre structure `struct Point` serait identique à un tableau de trois `double` et par conséquent l'initialisation suivante est possible :

```
struct Point point = { 3060426.957, 3192003.220, 4581359.381 };
```

Néanmoins on préfèrera la notation suivante, équivalente :

```
struct Point point = { .x=3060426.957, .y=3192003.220, .z=4581359.
↪381 };
```

Comme pour un tableau, les valeurs omises sont initialisées à zéro. Et de la même manière qu'un tableau, il est possible d'initialiser une structure à zéro avec `= {0};`.

Il faut savoir que **C99** restreint l'ordre dans lequel les éléments peuvent être initialisés. Ce dernier doit être l'ordre dans lequel les variables sont déclarées dans la structure.

Notons que des stuctures comportant des types différents peuvent aussi être initialisée de la même manière :

```
struct Product {
    int weight; // Grams
    double price; // Swiss francs
    int category;
    char name[64];
}

struct Product apple = {321, 0.75, 24, "Pomme Golden"};
```

11.3.3 Tableaux de structures

Une structure est un type comme un autre. Tout ce qui peut être fait avec `char` ou `double` peut donc être fait avec `struct`. Et donc, il est aussi possibel de déclarer un tableau de structures. Ici donnons l'exemple d'un tableaux de points initialisés :

```
struct Point points[3] = {
    {.x=1, .y=2, .z=3},
    {.z=1, .x=2, .y=3},
    {.y=1}
};
```

Assigner une nouvelle valeur à un point est facile :

```
point[2].x = 12;
```

11.3.4 Structures en paramètres

L'intérêt d'une structure est de pouvoir passer ou retourner un ensemble de données à une fonction. On a vu qu'une fonction ne permet de retourner qu'une seule primitive. Une structure est ici considérée comme un seul container et l'écriture suivante est possible :

```
struct Point generate_point(void) {
    struct Point p = {
        .x = rand(),
        .y = rand(),
        .z = rand()
    };

    return p;
}
```

Il est également possible de passer une structure en paramètre d'une fonction :

```
double norm(struct point p) {
    return sqrt(p.x * p.x + p.y * p.y + p.z * p.z);
}

int main(void) {
    struct Point p = { .x = 12.54, .y = -8.12, .z = 0.68 };

    double n = norm(p);
}
```

Contrairement aux tableaux, les structures sont toujours passées par valeur, c'est à dire que l'entier du contenu de la structure sera copié sur la pile (*stack*) en cas d'appel à une fonction. En revanche, en cas de passage par pointeur, seul l'adresse de la structure est passée à la fonction appelée qui peut dès lors modifier le contenu :

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

void foo(struct Point m, struct Point *n) {
    m.x++;
    n->x++;
}

int main(void) {
    struct Point p = {0}, q = {0};
    foo(p, &q);
    printf("%g, %g\n", p.x, q.x);
}
```

Le résultat affiché sera `0.0, 1.0`. Seul la seconde valeur est modifiée.

Indication : Lorsqu'un membre d'une structure est accédé, via son pointeur, on utilise la notation `->` au lieu de `.` car il est nécessaire de déréférencer le pointeur. Il s'agit d'un sucre syntaxique permettant d'écrire `p->x` au lieu de `(*p).x`

11.3.5 Structures flexibles

Introduit avec C99, les membres de structures flexibles ou *flexible array members* (§6.7.2.1) sont un membre de type tableau d'une structure défini sans dimension. Ces membres ne peuvent apparaître qu'à la fin d'une structure.

```
struct Vector {
    char name[16]; // name of the vector
    size_t len; // length of the vector
    double array[]; // flexible array member
};
```

Cette écriture permet par exemple de réserver un espace mémoire plus grand que la structure de base, et d'utiliser le reste de l'espace domme tableau flexible.

```
struct Vector *vector = malloc(1024);
strcpy(vector->name, "Mon vecteur");
vector->len = 1024 - 16 - 4;
for (int i = 0; i < vector->len; i++)
    vector->array[i] = ...
```

Ce type d'écriture est souvent utilisé pour des contenus ayant un en-tête fixe comme des images BMP ou des fichiers sons WAVE.

11.3.6 Structure de structures

On comprends aisément que l'avantage des structures et le regroupement de variables. Une structure peut être la composition d'autres types composites.

Nous déclarons ici une structure `struct Line` composée de `struct Point` :

```
struct Line {
    struct Point a;
    struct Point b;
};
```

L'accès à ces différentes valeurs s'effectue de la façon suivante :

```
struct Line line = {.a.x = 23, .a.y = 12, .b.z = 33};
printf("%g, %g", line.a.x, line.b.x);
```

11.3.7 Alignement mémoire

Une structure est agencée en mémoire dans l'ordre de sa déclaration. C'est donc un agencement linéaire en mémoire :

```
struct Line lines[2]; // Chaque point est un double, codé sur 8␣
↪bytes.
```

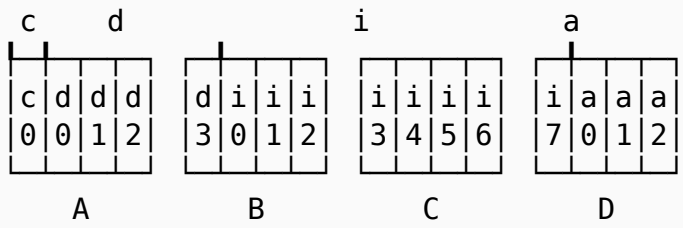
Ci-dessous est représenté l'offset mémoire (en bytes) à laquel est stocké chaque membre de la structure, ainsi que l'élément correspondant.

```
0x0000 line[0].a.x
0x0008 line[0].a.y
0x0010 line[0].a.z
0x0018 line[0].b.x
0x0020 line[0].b.y
0x0028 line[0].b.z
0x0030 line[1].a.x
0x0038 line[1].a.y
0x0040 line[1].a.z
0x0048 line[1].b.x
0x0050 line[1].b.y
0x0048 line[1].b.z
```

Néanmoins dans certains cas, le compilateur se réserve le droit d'optimiser l' alignement mémoire. Une architecture 32-bits aura plus de facilité à accéder à des grandeurs de 32 bits or, une structure composée de plusieurs entiers 8-bits demanderait au processeur un coût additionnel pour optimiser le stockage d'information. Considérons par exemple la structure suivante :

```
struct NoAlign
{
    int8_t c;
    int32_t d;
    int64_t i;
    int8_t a[3];
};
```

Imaginons pour comprendre qu'un casier mémoire sur une architecture 32-bits est assez grand pour y stocker 4 bytes. Tentons de représenter en mémoire cette structure en *little-endian*, en considérant des casiers de 32-bits :



On constate que la valeur `d` est à cheval entre deux casiers. De même que la valeur `i`

est répartie sur trois casiers au lieu de deux. Le processeur communique avec la mémoire en utilisant des *bus mémoire*, ils sont l'analogie d'une autoroute qui ne peux acceuillir que des voitures, chacune ne pouvant transporter que 4 passagers. Un passager ne peut pas arpenter l'autoroute sans voiture. Le processeur est la gare de triage et s'occupe de réassembler les passagers, et l'opération consistant à demander à un passager de sortir de la voiture `B` pour s'installer dans une autre, ou même se déplacer de la place conducteur à la place du passager arrière prend du temps.

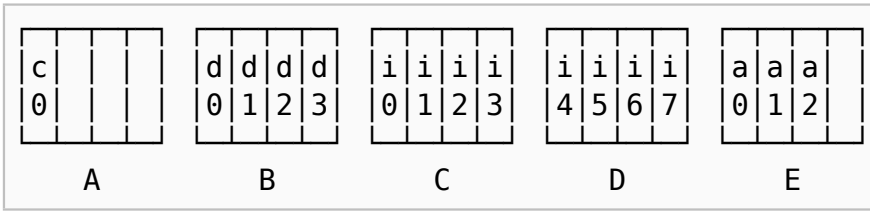
Le compilateur sera donc obligé de faire du zèle pour accéder à d. Formellement l'accès à `d` pourrait s'écrire ainsi :

```
int32_t d = (data[0] << 8) | (data[1] & 0x0F);
```

Pour éviter ces manoeuvres, le compilateur, selon l'architecture donnée, va insérer des éléments de rembourrage (*padding*) pour forcer l'alignement mémoire et ainsi optimiser les lectures. La même structure que ci-dessus sera fort probablement implémentée de la façon suivante :

```
struct Align
{
    int8_t c;
    int8_t __pad1[3]; // Inséré par le compilateur
    int32_t d;
    int64_t i;
    int8_t a[3];
    int8_t __pad2; // Inséré par le compilateur
};
```

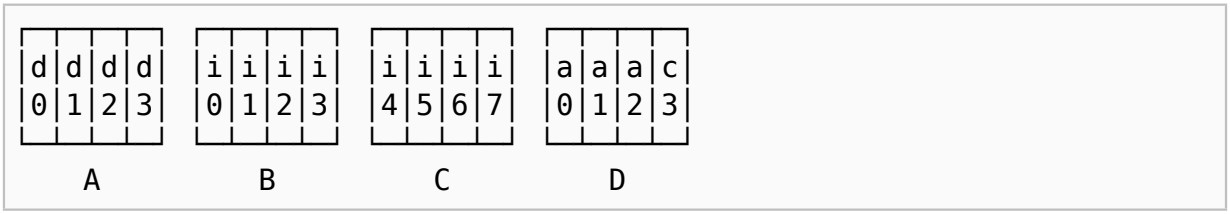
En reprenant notre analogie de voitures, le stockage est maintenant fait comme ceci :



Le compromis est qu'une voiture supplémentaire est nécessaire, mais le processeur n'a plus besoin de réagencer les passagers. L'accès à `d` est ainsi facilité au détriment d'une perte substentielle de l'espace de stockage.

Ceci étant, en changeant l'ordre des éléments dans la structure pour que chaque membre soit aligné sur 32-bits, il est possible d'obtenir un meilleur compromis :

```
struct Align
{
    int32_t d;
    int64_t i;
    int8_t a[3];
    int8_t c;
};
```

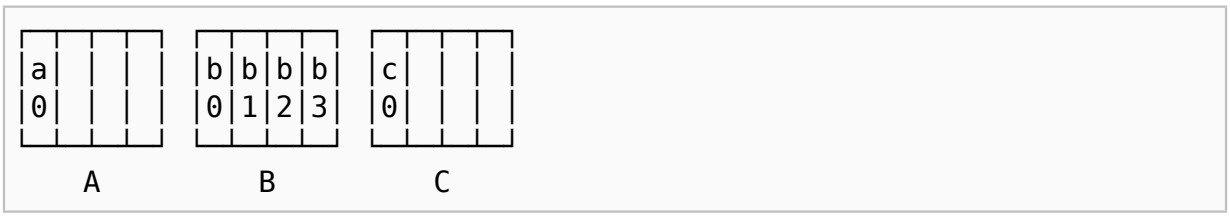


L'option `-Wpadded` de GCC permet lever une alerte lorsqu'une structure est alignée par le compilateur. Si l'on utilise par exemple une structure pour écrire un fichier binaire respectant un format précis par exemple l'en-tête d'un fichier BMP. Et que cette structure `BitmapFileHeader` est enregistrée avec `fwrite(header, sizeof(BitmapFileHeader), ...)`. Si le compilateur rajoute des éléments de rembourrage, le fichier BMP serait alors compromis. Il faudrait donc considérer l'alerte `Wpadded` comme une erreur critique.

Pour palier à ce problème, lorsqu'une structure mémoire doit être respectée dans un ordre précis. Une option de compilation non standard existe. La directive `#pragma pack` permet de forcer un type d'alignement pour une certaine structure. Considérons par exemple la structure suivante :

```
struct Test
{
    char a;
    int b;
    char c;
};
```

Elle serait très probablement représentée en mémoire de la facon suivante :



En revance si elle est décrite en utilisant un *packing* sur 8-bits, avec `#pragma pack(1)` on aura l'alignement mémoire suivant :

11.4 Champs de bits

Les champs de bits sont des structures dont une information supplémentaire est ajoutée : le nombre de bits utilisés.

Prenons l'exemple du module I2C du microcontrôleur TMS320F28335. Le registre `I2CMDR` décrit à la page 23 est un registre 16-bits qu'il conviendrait de décrire avec un champ de bits :

```
struct I2CMDR {
    int  bc  :3;
    bool fdf :1;
    bool stb :1;
    bool irs :1;
    bool dlb :1;
    bool rm  :1;
    bool xa  :1;
    bool trx :1;
    bool mst :1;
    bool stp :1;
    bool _reserved :1;
    bool stt  :1;
    bool free :1;
    bool nackmod :1;
};
```

Activer le bit `stp` (bit numéro 12) devient une opération triviale :

```
struct I2CMDR i2cmdr;

i2cmdr.stp = true;
```

Alors qu'elle demanderait une manipulation de bit sinon :

```
int32_t i2cmdr;

i2cmdr |= 1 << 12;
```

Notons que les champs de bits, ainsi que les structures seront déclarées différemment selon que l'architecture cible est *little-endian* ou *big-endian*.

11.5 Unions

Une union est une variable qui peut avoir plusieurs représentations d'un même contenu mémoire. Rappelez-vous, au Section 6.2 nous nous demandions quelle était l'interprétation d'un contenu mémoire donné. Il est possible en C d'avoir toutes les interprétations à la fois :

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

union Mixed
{
    int32_t signed32;
    uint32_t unsigned32;
    int8_t signed8[4];
    int16_t signed16[2];
    float float32;
};

int main(void) {
    union Mixed m = {
        .signed8 = {0b11011011, 0b0100100, 0b01001001, 0b01000000}
    };

    printf(
        "int32_t\t%d\n"
        "uint32_t\t%u\n"
        "char\t%c, %c, %c, %c\n"
        "short\t%hu, %hu\n"
        "float\t%f\n",
        m.signed32,
        m.unsigned32,
        m.signed8[0], m.signed8[1], m.signed8[2], m.signed8[3],
        m.signed16[0], m.signed16[1],
        m.float32
    );
}
```

Les unions sont très utilisées en combinaison avec des champs de bits. Pour reprendre l'exemple du champ de bit évoqué plus haut, on peut souhaiter accéder au registre soit sous la forme d'un entier 16-bits soit via chacun de ses bits indépendamment.

```c
union i2cmdr {
    struct {
        int  bc  :3;
        bool fdf :1;
        bool stb :1;
        bool irs :1;
        bool dlb :1;
```

(suite sur la page suivante)

(suite de la page précédente)

```
        bool rm  :1;
        bool xa  :1;
        bool trx :1;
        bool mst :1;
        bool stp :1;
        bool _reserved :1;
        bool stt  :1;
        bool free :1;
        bool nackmod :1;
    } bits;
    uint16_t all;
};
```

11.6 Création de type

Le mot clé `typedef` permet de déclarer un nouveau type. Il est particulièrement utilisé conjointement avec les structures et les unions afin de s'affranchir de la lourdeur d'écriture (préfixe `struct`), et dans le but de cacher la complexité d'un type à l'utilisateur qui le manipule.

L'exemple suivant déclare un type `Point` et un prototype de fonction permettant l'addition de deux points.

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

Point add(Point a, Point b);
```

11.7 Compound Literals

Naïvement traduit en *litéraux composés*, un *compound literal* est une méthode de création d'un type composé "à la volée" utilisé de la même façon que les transtypages.

Reprenons notre structure Point `struct Point` vue plus haut. Si l'on souhaite changer la valeur du point `p` il faudrait on pourrait écrire ceci :

```
struct Point p; // Déclaré plus haut

// ...

{
    struct Point q = {.x=1, .y=2, .z=3};
```

(suite sur la page suivante)

(suite de la page précédente)

```
    p = q;
}
```

Notons que passer par une variable intermédiaire `q` n'est pas très utile. Il serait préférable d'écrire ceci :

```
p = {.x=1, .y=2, .z=3};
```

Néanmoins cette écriture mènera à une erreur de compilation car le compilateur cherchera à déterminer le type de l'expression `{.x=1, .y=2, .z=3}`. Il est alors essentiel d'utiliser la notation suivante :

```
p = (struct Point){.x=1, .y=2, .z=3};
```

Cette notation de litéraux composés peut également s'appliquer aux tableaux. L'exemple suivant montre l'initialisation d'un tableau à la volée passé à la fonction `foo` :

```
void foo(int array[3]) {
    for (int i = 0; i < 3; i++) printf("%d ", array[i]);
}

void main() {
    foo((int []){1,2,3});
}
```

Exercice

Chaque élément du taleau périodique des éléments comporte les propriétés suivantes :

- Un nom jusqu'à 20 lettres
- Un symbole jusqu'à 2 lettres
- Un numéro atomique de 1 à 118 (2019)
- **Le type de l'élément**
  - **Métaux**
    - Alcalin
    - Alcalino-terreux
    - Lanthanides
    - Actinides
    - Métaux de transition
    - Métaux pauvres
  - Métalloïdes
  - **Non métaux**
    - Autres
    - Halogène
    - Gaz noble
- La période : un entier de 1 à 7
- Le groupe : un entier de 1 à 18

Déclarer une structure de données permettant de stocker tous les éléments chimiques de tel facon qu'ils puissent être accédés comme :

```
assert(strcmp(table.element[6].name, "Helium") == 0);
assert(strcmp(table.element[54].type, "Gaz noble") == 0);
assert(table.element[11].period == 3);

Element *el = table.element[92];
assert(el->atomic_weight == 92);
```

Chapitre 12

Les fichiers

12.1 Système de fichiers

Dans un environnement POSIX tout est fichier. `stdin` est un fichier, une souris USB est un fichier, un clavier est un fichier, un terminal est un fichier, un programme est un fichier.

Les fichiers sont organisés dans une arborescence gérée par un système de fichiers. Sous Windows l'arborescence classique est :

```
C :
├── Program Files          Programmes installés
├── Users                  Comptes utilisateur
│   └── John
│       ├── Desktop
│       ├── Documents
│       └── Music
└── Windows
    ├── Fonts              Polices de caractères
    ├── System32           Système d'exploitation 64-bits (oui, oui)
    └── Temp               Fichiers temporaires
```

Il y a une arborescence par disque physique `C:`, `D:`, une arborescence par chemin réseau `\\eistore2`, etc. Sous POSIX, la stratégie est différente, car il n'existe qu'UN SEUL système de fichier dont la racine est `/`.

```
/
├── bin                    Programmes exécutables cruciaux
├── dev                    Périphériques (clavier, souris, ...)
├── usr
│   └── bin                Programmes installés
├── mnt                    Points de montage (disques réseaux, CD,␣
↪clé USB)
│   └── eistore2
```

(suite sur la page suivante)

(suite de la page précédente)

```
├── tmp                     Fichiers temporaires
├── home                    Comptes utilisateurs
│   └── john
│       └── documents
└── var                     Fichiers variables comme les logs ou les␣
↪database
```

Chaque élément qui contient d'autres éléments est appelé un **répertoire** ou **dossier**, en anglais *directory*. Chaque répertoire contient toujours au minimum deux fichiers spéciaux :

- `.` Un fichier qui symbolise le répertoire courant, celui dans lequel je me trouve
- `..` Un fichier qui symbolise le répertoire parent, c'est à dire `home` lorsque je suis dans `john`.

La localisation d'un fichier au sein d'un système de fichier peut être soit **absolue** soit **relative**. Cette localisation s'appelle un **chemin** ou *path*. La convention est d'utiliser le symbole :

— Slash `/` sous POSIX
— Antislash `\` sous Windows

Le chemin `/usr/bin/.././bin/../../home/john/documents` est correct mais il n'est pas canonique. La forme canonique est `/home/john/documents`. Un chemin peut être relatif s'il ne commence pas par un `/` : `../bin`. Sous Windows c'est pareil mais la racine différemment selon le type de média `C:\`, `\\network`, ...

Lorsqu'un programme s'exécute, son contexte d'exécution est toujours par rapport à son emplacement dans le système de fichier donc le chemin peut être soit relatif, soit absolu.

12.1.1 Navigation

Sous Windows (PowerShell) ou un système **POSIX** (Bash/Sh/Zsh), la navigation dans une arborescence peut être effectuée en ligne de commande à l'aide des commandes (programmes) suivants :

— `ls` est un raccourcis du nom *list*, ce programme permet d'afficher sur la sortie standard le contenu d'un répertoire.
— `cd` pour *change directory* permet de naviger dans l'arboresence. Le programme prend en argument un chemin absolu ou relatif. En cas d'absence d'arguments, le programme redirige vers le répertoire de l'utilisateur courant.

12.2 Format d'un fichier

Un fichier peut avoir un contenu arbitraire ; une suite de zéros et de uns binaire. Selon l'interprétation, un fichier pourrait contenir une image, un texte ou un programme. Le cas particulier ou le contenu est lisible par un éditeur de texte, on appelle ce fichier un fichier texte. C'est-à-dire que chaque caractère est encodé sur 8-bit et que la table ASCII est utilisée pour traduire le contenu en un texte intelligible. Lorsque le contenu n'est pas du texte, on l'appelle un fichier binaire.

La frontière est parfois assez mince, car parfois le fichier binaire peut contenir du texte intelligible, la preuve avec ce programme :

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

int main(char* argc, char** argv)
{
    static const char password[] = "un mot de passe secret";
    return strcmp(argv[1], password);
}
```

Si nous le compilons et cherchons dans son code binaire :

```
$ gcc example.c
| $ hexdump -C a.out                                          | grep␣
↪-C3 sec     |
| 000006f0  f3 c3 00 00 48 83 ec 08  48 83 c4 08 c3 00 00 00 | ....
↪H...H....... |
| 00000700  01 00 02 00 00 00 00 00  00 00 00 00 00 00 00 00 | .....
↪.......... |
| 00000710  75 6e 20 6d 6f 74 20 64  65 20 70 61 73 73 65 20 | un␣
↪mot de passe  |
| 00000720  73 65 63 72 65 74 00 00  01 1b 03 3b 3c 00 00 00 |␣
↪secret.....;<... |
| 00000730  06 00 00 00 e8 fd ff ff  88 00 00 00 08 fe ff ff | .....
↪.......... |
| 00000740  b0 00 00 00 18 fe ff ff  58 00 00 00 22 ff ff ff | .....
↪...X..."... |
| 00000750  c8 00 00 00 58 ff ff ff  e8 00 00 00 c8 ff ff ff | ....
↪X.......... |
```

Sous un système POSIX, il n'existe aucune distinction formelle entre un fichier binaire et un fichier texte. En revanche sous Windows il existe une subtile différence concernant surtout le caractère de fin de ligne. La commande `copy a.txt + b.txt c.txt` considère des fichiers textes et ajoutera automatiquement une fin de ligne entre chaque partie concaténée, mais celle-ci `copy /b a.bin + b.bin c.bin` ne le fera pas.

12.3 Ouverture d'un fichier

Sous POSIX, un programme doit demander au système d'exploitation l'accès à un fichier soit en lecture, soit en écriture soit les deux. Le système d'exploitation retourne un descripteur de fichier qui est simplement un entier unique pour le programme.

```c
#include <fcntl.h>
#include <stdio.h>
#include <sys/stat.h>
```

(suite sur la page suivante)

(suite de la page précédente)

```c
int main(void)
{
    int fd = open("toto", O_RDONLY);
    printf("%d\n", fd);
    getchar();
}
```

Lorsque le programme ci-dessus est exécuté, il va demander l'ouverture du fichier `toto` en lecture et recevoir un descripteur de fichier `fd` (*file descriptor*) positif en cas de succès ou négatif en cas d'erreur.

Dans l'exemple suivant, on compile, puis exécute en arrière-plan le programme qui ne se terminera pas puisqu'il attend un caractère d'entrée. L'appel au programme `ps` permet de lister la liste des processus en cours et la recherche de `test` permet de noter le numéro du processus, ici `6690`. Dans l'arborescence de fichiers, il est possible d'aller consulter les descripteurs de fichiers ouverts pour le processus concerné.

```
$ gcc test.c -o test && ./test &
$ ps -u | grep test
ycr       6690  0.0  0.0  10540   556 pts/4    T    11:19   0:00␣
 ↪test
$ ls /proc/6690/fd
0  1  2  3
```

On observe que trois descripteurs de fichiers sont ouverts.

- `0` pour `STDIN`
- `1` pour `STDOUT`
- `2` pour `STDERR`
- `3` pour le fichier `toto` ouvert en lecture seule

La fonction `open` est en réalité un appel système qui n'est standardisé que sous POSIX, c'est-à-dire que son utilisation n'est pas portable. L'exemple cité est principalement évoqué pour mieux comprendre le mécanisme de fond pour l'accès aux fichiers.

En réalité la bibliothèque standard, respectueuse de C99, dispose d'une fonction `fopen` pour *file open* qui offre plus de fonctionnalités. Ouvrir un fichier se résume donc à

```c
#include <stdio.h>

int main(void)
{
    FILE *fp = fopen("toto", "r");

    if (fp == NULL) {
        return -1; // Error the file cannot be accessed
    }

    // ...
}
```

Le mode d'ouverture du fichier peut être :

r Ouverture en lecture seule depuis le début du fichier.

r+ Ouverture pour lecture et écriture depuis le début du fichier.

w Ouverture en écriture. Le fichier est créé s'il n'existe pas déjà, sinon le contenu est effacé. Le pointeur de fichier est positionné au début de ce dernier.

w+ Ouverture en écriture et lecture. Le fichier est créé s'il n'existe pas déjà. Le pointeur de fichier est positionné au début de ce dernier.

a Ouverture du fichier pour insertion. Le fichier est créé s'il n'existe pas déjà. Le pointeur est positionné à la fin du fichier.

a+ Ouverture du fichier pour lecture et écriture. Le fichier est créé s'il n'existe pas déjà et le pointeur du fichier est positionné à la fin.

Sous Windows et pour soucis de compatibilité, selon la norme C99, le flag `b` pour *binary* existe. Pour ouvrir un fichier en mode binaire on peut alors écrire `rb+`.

L'ouverture d'un fichier cause, selon le mode, un accès exclusif au fichier. C'est-à-dire que d'autres programmes ne pourront pas accéder à ce fichier. Il est donc essentiel de toujours refermer l'accès à un fichier dès lors que l'opération de lecture ou d'écriture est terminée :

```
flose(fp);
```

On peut noter que sous POSIX, écrire sur `stdout` ou `stderr` est exactement la même chose qu'écrire sur un fichier, il n'y a aucune distinction. Exercice

Écrire un programme qui saisit le nom d'un fichier texte, ainsi qu'un texte à rechercher. Le programme affiche ensuite le numéro de toutes les lignes du fichier contenant le texte recherché.

```
$ ./search
Fichier: foo.txt
Recherche: bulbe

4
5
19
132
981
```

Question subsidiaire : que fait le programme suivant :

```
$ grep foo.txt bulbe
```

12.4 Navigation dans un fichier

Lorsqu'un fichier est ouvert, un curseur virtuel est positionné soit au début soit à la fin du fichier. Lorsque des données sont lues ou écrites, c'est à la position de ce curseur, lequel peut être déplacé en utilisant plusieurs fonctions utilitaires.

La navigation dans un fichier n'est possible que si le fichier est *seekable*. Généralement les pointeurs de fichiers `stdin`, `stdout` et `stderr` ne sont pas *seekable*, et il n'est pas possible de se déplacer dans le fichier mais seulement écrire dedans.

12.4.1 fseek

```
int fseek(FILE *stream, long int offset, int whence)
```

Le manuel man fseek indique les trois constantes possibles pour `whence` :

SEEK_SET Positionne le curseur au début du fichier.

SEEK_CUR Position courante du curseur. Permets d'ajouter un offset relatif à la position courante.

SEEK_END Positionne le curseur à la fin du fichier.

12.4.2 ftell

Il est parfois utile de savoir où se trouve le curseur. `ftell()` retourne la position actuelle du curseur dans un fichier ouvert.

```
char filename[] = "foo";

FILE *fp = fopen(filename, 'r');
fseek(fp, 0, SEEK_END);
long int size = ftell();

printf("The file %s has a size of %ld Bytes\n", filename, size);
```

12.4.3 rewind

L'appel `rewind()` est équivalent à `(void) fseek(stream, 0L, SEEK_SET)` et permet de se positionner au début du fichier.

12.5 Lecture / Écriture

La lecture, écriture dans un fichier s'effectue de manière analogue aux fonctions que nous avons déjà vues `printf` et `scanf` pour les flux standards (*stdout*, *stderr*), mais en utilisant les pendants fichiers :

`int fscanf(FILE *stream, const char *format, ...)` Équivalent à `scanf` mais pour les fichiers

`int fprintf(FILE *stream, const char *format, ...)` Équivalent à `printf` mais pour les fichiers

`int fgetc(FILE *stream)` Équivalent à `getchar` (ISO/IEC 9899 §7.19.7.6-2)

`int fputc(FILE *stream, char char)` Équivalent à `putchar` (ISO/IEC 9899 §7.19.7.9-2)

`char *fgets(char * restrict s, int n, FILE * restrict stream)` Équivalent à `gets`

`int fputs(const char * restrict s, FILE * restrict stream)` Équivalent à `puts`

Bref... Vous avez compris.

Les nouvelles fonctions à connaître sont les suivantes :

`size_t fread(void *ptr, size_t size, size_t nmemb, FILE *stream)` Lecture arbitraire de `nmemb * size` bytes depuis le flux `stream` dans le buffer `ptr` :

```c
int32_t buffer[12] = {0};
fread(buffer, 2, sizeof(int32_t), stdin);

printf("%x\n%x\n", buffer[0], buffer[1]);
```

```
$ echo -e "0123abcdefgh" | ./a.out
33323130
64636261
```

On notera au passage la nature *little-endian* du système.

`size_t fwrite(const void *ptr, size_t size, size_t nmemb, FILE *stream` La fonction est similaire à `fread` mais pour écrire sur un flux.

12.6 Buffer de fichier

Pour améliorer les performances, C99 prévoit (§7.19.3-3), un espace tampon pour les descripteurs de fichiers qui peuvent être :

`unbuffered` (**`_IONBF`**) Pas de buffer, les caractères lus ou écrits sont acheminés le plus vite possible de la source à la destination.

`fully buffered` (`_IOFBF`)

`line buffered` (`_IO_LBF`)

Il faut comprendre qu'à chaque instant un programme souhaite écrire dans un fichier, il doit générer un appel système et donc interrompre le noyau. Un programme qui écrirait caractère par caractère sur la sortie standard agirait de la même manière qu'un employé des postes qui irait distribuer son courrier en ne prenant qu'une enveloppe à la fois, de la centrale de distribution au destinataire.

Par défaut, un pointeur de fichier est *fully buffered*. C'est-à-dire que dans le cas du programme suivant devrait exécuter 10x l'appel système `write`, une fois par caractère.

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

int main(int argc, char* argv[])
{
    if (argc > 1 && strcmp("--no-buffering", argv[1]) == 0)
        setvbuf(stdout, NULL, _IONBF, 0);

    for (int i = 0; i < 10; i++)
        putchar('c');
}
```

Cependant le comportement réel est différent. Seulement si le buffer est désactivé que le programme interrompt le noyau pour chaque caractère :

```
$ gcc buftest.c -o buftest

$ strace ./buftest 2>&1 | grep write
write(1, "cccccccccc", 10cccccccccc)              = 10

$ strace ./buftest --no-buffering 2>&1 | grep write
write(1, "c", 1c)                                 = 1
write(1, "c", 1c)                                 = 1
write(1, "c", 1c)                                 = 1
write(1, "c", 1c)                                 = 1
write(1, "c", 1c)                                 = 1
write(1, "c", 1c)                                 = 1
write(1, "c", 1c)                                 = 1
write(1, "c", 1c)                                 = 1
write(1, "c", 1c)                                 = 1
write(1, "c", 1c)                                 = 1
```

Le changement de mode peut être effectué avec la fonction `setbuf` ou `setvbuf` :

```c
#include <stdio.h>

int main(void) {
    char buf[1024];
```

(suite sur la page suivante)

(suite de la page précédente)

```
    setbuf(stdout, buf);

    fputs("Allo ?");

    fflush(stdout);
}
```

La fonction `fflush` force l'écriture malgré l'utilisation d'un buffer.

12.7 Fichiers et Flux

Historiquement les descripteurs de fichiers sont appelés `FILE` alors qu'ils sont préférablement appelés `streams` en C++. Un fichier au même titre que `stdin`, `stdout` et `stderr` sont des flux de données. La norme POSIX, décrit que par défaut les flux :

- `0`. `STDIN`,
- `1`. `STDOUT`,
- `2`. `STDERR`,

sont ouverts au début du programme. Le premier fichier ouvert, par exemple avec `fopen` sera très probablement assigné à l'identifiant `3`.

Pour se convaincre de cela, on peut exécuter l'exemple suivant avec le programme `strace` :

```c
#include <stdio.h>

int main(void) {
    char c = fgetc(stdin);

    FILE *fd = fopen("file", "w");
    fputc(c, fd);
    fputc(c + 1, stdout);
    fputc(c + 2, stderr);
}
```

Pour mémoire `strace` permet de capturer les appels systèmes du programme passé en argument et de les afficher. Deux particularités de la commande exécutée sont `2>&1` qui redirige `stderr` vers `stdout` afin de pouvoir rediriger le flux vers `grep`. Ensuite `grep` permet de filtrer la sortie pour n'afficher que les lignes contenant `open`, `read`, `write` ou `close` :

```
$ echo k | strace ./a.out 2>&1 | grep -P 'open|read|write|close'
read(0, "k\n", 4096)                    = 2
openat(AT_FDCWD, "file", O_WRONLY|O_CREAT|O_TRUNC, 0666) = 3
write(2, "m", 1m)                        = 1
write(3, "k", 1)                        = 1
write(1, "l", 1l)                        = 1
```

On peut voir que l'on lit `k\n` sur le flux `0`, soit `stdin`, puis que le fichier `file` est ouvert, il porte l'identifiant `3`, enfin on écrit sur `1`, `2` et `3`.

12.8 Formats de sérialisation

Souvent les fichiers sont utilisés pour stocker de l'information organisée en grille, par exemple, la liste des températures maximales par ville et par mois :

Pays	Ville	01	02	03	04	05	06	07	08	09	10	11	12
Suisse	Zürich	0.3	1.3	5.3	8.8	13.3	16.4	18.6	18.0	14.1	9.9	4.4	1.4
Italie	Rome	7.5	8.2	10.2	12.6	17.2	21.1	24.1	24.5	20.8	16.4	11.4	8.4
Allemagne	Berlin	0.6	2.3	5.1	10.2	14.8	17.9	20.3	19.7	15.3	10.5	6.0	1.33
Yémen	Aden	25.7	26.0	27.2	28.9	31.0	32.7	32.7	31.5	31.6	28.9	27.1	26.01
Russie	Yakutsk	-38.6	-33.8	-20.1	-4.8	7.5	16.4	19.5	15.2	6.1	-7.8	-27.0	-37.6

Il existe plusieurs manière d'écrire ces informations dans un fichier :

- Écriture tabulée
- Écriture avec remplissage
- Utiliser un langage de sérialisation de haut niveau comme JSON, YAML ou XML

12.8.1 Format tabulé

Un fichier dit tabulé, utilise une sentinelle, souvent le caractère de tabulation `\t` pour séparer les données. Chaque ligne du tableau est physiquement séparée de la suivante avec un `\n` :

```
Pays\tVille\t01\t02\t03\t04\t05\t06\t07\t08\t09\t10\t11\t12\n
Suisse\tZürich\t0.3\t1.3\t5.3\t8.8\t13.3\t16.4\t18.6\t18.0\t14.1\t9.
↪9\t4.4\t1.4\n
Italie\tRome\t7.5\t8.2\t10.2\t12.6\t17.2\t21.1\t24.1\t24.5\t20.8\
↪t16.4\t11.4\t8.4\n
Allemagne\tBerlin\t0.6\t2.3\t5.1\t10.2\t14.8\t17.9\t20.3\t19.7\t15.
↪3\t10.5\t6.0\t1.33\n
Yémen\tAden\t25.7\t26.0\t27.2\t28.9\t31.0\t32.7\t32.7\t31.5\t31.6\
↪t28.9\t27.1\t26.01\n
Russie\tYakutsk\t-38.6\t-33.8\t-20.1\t-4.8\t7.5\t16.4\t19.5\t15.2\
↪t6.1\t-7.8\t-27.0\t-37.6\n
```

Ce fichier peut être observé avec un lecteur hexadécimal :

```
$ hexdump -C data.dat
00000000  50 61 79 73 09 56 69 6c  6c 65 09 30 31 09 30 32  |Pays.
↪Ville.01.02|
```

(suite sur la page suivante)

(suite de la page précédente)

```
00000010  09 30 33 09 30 34 09 30  35 09 30 36 09 30 37 09  |.03.04.
 ↪05.06.07.|
00000020  30 38 09 30 39 09 31 30  09 31 31 09 31 32 0a 53  |08.09.
 ↪10.11.12.S|
00000030  75 69 73 73 65 09 5a c3  bc 72 69 63 68 09 30 2e  |uisse.
 ↪Z..rich.0.|
00000040  33 09 31 2e 33 09 35 2e  33 09 38 2e 38 09 31 33  |3.1.3.
 ↪5.3.8.8.13|
00000050  2e 33 09 31 36 2e 34 09  31 38 2e 36 09 31 38 2e  |.3.16.
 ↪4.18.6.18.|
00000060  30 09 31 34 2e 31 09 39  2e 39 09 34 2e 34 09 31  |0.14.1.
 ↪9.9.4.4.1|
00000070  2e 34 0a 49 74 61 6c 69  65 09 52 6f 6d 65 09 37  |.4.
 ↪Italie.Rome.7|
00000080  2e 35 09 38 2e 32 09 31  30 2e 32 09 31 32 2e 36  |.5.8.2.
 ↪10.2.12.6|
00000090  09 31 37 2e 32 09 32 31  2e 31 09 32 34 2e 31 09  |.17.2.
 ↪21.1.24.1.|
000000a0  32 34 2e 35 09 32 30 2e  38 09 31 36 2e 34 09 31  |24.5.
 ↪20.8.16.4.1|
000000b0  31 2e 34 09 38 2e 34 0a  41 6c 6c 65 6d 61 67 6e  |1.4.8.
 ↪4.Allemagn|
000000c0  65 09 42 65 72 6c 69 6e  09 30 2e 36 09 32 2e 33  |e.
 ↪Berlin.0.6.2.3|
000000d0  09 35 2e 31 09 31 30 2e  32 09 31 34 2e 38 09 31  |.5.1.
 ↪10.2.14.8.1|
000000e0  37 2e 39 09 32 30 2e 33  09 31 39 2e 37 09 31 35  |7.9.20.
 ↪3.19.7.15|
000000f0  2e 33 09 31 30 2e 35 09  36 2e 30 09 31 2e 33 33  |.3.10.
 ↪5.6.0.1.33|
00000100  0a 59 c3 a9 6d 65 6e 09  41 64 65 6e 09 32 35 2e  |.Y..
 ↪men.Aden.25.|
00000110  37 09 32 36 2e 30 09 32  37 2e 32 09 32 38 2e 39  |7.26.0.
 ↪27.2.28.9|
00000120  09 33 31 2e 30 09 33 32  2e 37 09 33 32 2e 37 09  |.31.0.
 ↪32.7.32.7.|
00000130  33 31 2e 35 09 33 31 2e  36 09 32 38 2e 39 09 32  |31.5.
 ↪31.6.28.9.2|
00000140  37 2e 31 09 32 36 2e 30  31 0a 52 75 73 73 69 65  |7.1.26.
 ↪01.Russie|
00000150  09 59 61 6b 75 74 73 6b  09 2d 33 38 2e 36 09 2d  |.
 ↪Yakutsk.-38.6.-|
00000160  33 33 2e 38 09 2d 32 30  2e 31 09 2d 34 2e 38 09  |33.8.-
 ↪20.1.-4.8.|
00000170  37 2e 35 09 31 36 2e 34  09 31 39 2e 35 09 31 35  |7.5.16.
 ↪4.19.5.15|
00000180  2e 32 09 36 2e 31 09 2d  37 2e 38 09 2d 32 37 2e  |.2.6.1.
 ↪-7.8.-27.|
00000190  30 09 2d 33 37 2e 36 0a                           |0.-37.
 ↪6.|
```

(suite sur la page suivante)

(suite de la page précédente)

```
00000198
```

L'inconvénient de ce format est que pour obtenir directement la température du mois de mars à Berlin, sachant que Berlin est la quatrième ligne du fichier, il est nécessaire de parcourir le fichier depuis le début car la longueur des lignes n'est à priori pas connue. On dit que la lecture séquentielle est facilitée, mais la lecture aléatoire est plus lente.

12.8.2 Format avec remplissage

Pour palier au défaut du format tabulé, il est possible d'écrire le fichier en utilisant un caractère de remplissage. Dans le fichier suivant, les mois de mai sont toujours aligné avec la 48 ième colonne :

```
Pays       Ville    01     02     03     04    05    06    07    08␣
↪   09    10    11     12
Suisse     Zürich   0.3    1.3    5.3    8.8   13.3  16.4  18.6  18.
↪0  14.1  9.9   4.4    1.4
Italie     Rome     7.5    8.2    10.2   12.6  17.2  21.1  24.1  24.
↪5  20.8  16.4  11.4   8.4
Allemagne  Berlin   0.6    2.3    5.1    10.2  14.8  17.9  20.3  19.
↪7  15.3  10.5  6.0    1.33
Yémen      Aden     25.7   26.0   27.2   28.9  31.0  32.7  32.7  31.
↪5  31.6  28.9  27.1   26.01
Russie     Yakutsk  -38.6  -33.8  -20.1  -4.8  7.5   16.4  19.5  15.
↪2  6.1   -7.8  -27.0  -37.6
```

Idéalement on utilise comme caractère de remplissage le caractère nulle `\0` mais le caractère espace peut aussi convenir à condition que les données ne contiennent pas d'espace.

La lecture aléatoire de ce type de fichier est facilitée car la position de chaque entrée est connue à l'avance, on sait par exemple que le pays est stocké sur 11 caracètres, la ville sur 9 caractères et chaque température sur 7 caractères.

L'utilisation de `fseek` est par conséquent utile :

```
int line = 2;
int month = 3;
double temperature;

fseek(fd, line * (11 + 9 + 12 * 7 + 1), SEEK_SET);
fseek(fd, 11 + 9 + month * 7 SEEK_CUR);
fscanf(fd, "%lf", &temperature);
```

L'incovénient de ce format de fichier est la place qu'il prend en mémoire. L'autre problème est que si le nom d'une ville dépasse les 9 caractères alloués, il faut réécrire tout le fichier. Généralement ce problème est contourné en allouant des champs d'une taille suffisante, par exemple 256 carctères pour le nom des villes.

12.8.3 Format sérialisé

Des langages de sérialisation permettent de stucturer de l'information en utilisant un format spécifique. Ici JSON :

```
[
    {
        "pays": "Suisse",
        "ville": "Zürich",
        "mois": {
            "janvier": 0.3,
            "février": 1.3,
            "mars": 5.3,
            "avril": 8.8,
            "mai": 13.3,
            "juin": 16.4,
            "juillet": 18.6,
            "août": 18.0,
            "septembre": 14.1,
            "octobre": 9.9,
            "novembre": 4.4,
            "décembre": 1.4
        }
    },
    {
        "pays": "Italie",
        "ville": "Rome",
        "mois": {
            "janvier": 7.5,
            "février": 8.2,
            "mars": 10.2,
            "avril": 12.6,
            "mai": 17.2,
            "juin": 21.1,
            "juillet": 24.1,
            "août": 24.5,
            "septembre": 20.8,
            "octobre": 16.4,
            "novembre": 11.4,
            "décembre": 8.4
        }
    }
]
```

L'avantage de ce type de format est qu'il est facilement modifiable avec un éditeur de texte et qu'il est très interopérable. C'est à dire qu'il est facilement lisible depuis différents langage de programmation.

En C, on pourra utilier la bibliothèque logicielle json-c.

Exercice

Considérez les deux programmes ci-dessous très similaires.

```c
#include <stdio.h>

int main(void)
{
    char texte[80];

    printf("Saisir un texte:");
    gets(texte);
    printf("Texte: %s\n", texte);
}
```

```c
#include <stdio.h>

int main(void)
{
    char texte[80];

    printf("Saisir un texte:");
    fgets(texte, 80, stdin);
    printf("Texte: %s\n", texte);
}
```

1. Quelle est la différence entre ces 2 programmes ?
2. Dans quel cas est-ce que ces programmes auront un comportement différent ?
3. Quelle serait la meilleure solution ?

Chapitre 13

Gestion de la mémoire

Vous l'aurez appris à vos dépens, l'erreur *Segmentation fault* (erreur de segmentation) arrive souvent lors du développement. Ce chapitre s'intéresse à la mémoire et vulgarise les concepts de segmentation et traite de l'allocation dynamique.

La mémoire d'un programme est découpée en segments de données. Les principaux segments sont :

Segment de code `.text` Les instructions du programme exécutable sont chargées dans ce segment.

Segment de constantes et chaînes de caractères `.rodata` Les constantes globales `const int = 13` et les chaînes de caractères sont enregistrées dans ce segment.

Segment de variables initialisées `.bss` Ce segment est garanti d'être initialisé à zéro lorsque le programme est chargé en mémoire. Les variables globales statiques tels que `static int foo = 0` seront stockées dans ce segment.

Segment de variables non initialisées `.data` Les variables globales non initialisées comme `static int bar;` seront placées dans ce segment.

Segment de tas `.heap` Les allocations dynamiques décrites plus bas dans ce chapitre sont déclarées ici.

Segment de pile `.stack` La chaîne d'appel de fonction ainsi que toutes les variables locales sont mémorisées dans ce segment.

13.1 Allocation statique

Jusqu'ici toutes les variables que nous avons déclarées ont été déclarées statiquement. C'est-à-dire que le compilateur est capable a priori de savoir combien de place prend telle ou telle variable et les agencer en mémoire dans les bons segments. On appelle cette méthode d'allocation de mémoire l'allocation statique.

La déclaration statique suivante déclare un tableau de 1024 entiers 64-bits initialisés à zéro et stockés dans le segment `.bss`, soit 64 kio :

```
static int64_t vector[1024] = {0};
```

13.2 Allocation dynamique

Il est des circonstances ou un programme ne sait pas combien de mémoire il a besoin. Par exemple un programme qui compterait le nombre d'occurrences de chaque mot dans un texte devra se construire un index de tous les mots qu'il découvre lors de la lecture du fichier d'entrée. A priori ce fichier d'entrée étant inconnu au moment de l'exécution du programme, l'espace mémoire nécessaire à construire ce dictionnaire de mots est également inconnu.

L'approche la plus naïve serait d'anticiper le cas le plus défavorable. Le dictionnaire Littré comporte environ 132'000 mots tandis que le Petit Larousse Illustré 80'000 mots environ. Pour se donner une bonne marge de manoeuvre et anticiper les anglicismes et les noms propres. Il suffirait de réserver un tableau de 1 million de mots de 10 caractères soit un peu plus de 100 MiB de mémoire quand bien même le fichier qui serait lu ne comporterait que 2 mots : `Hello World!`.

L'approche correcte est d'allouer la mémoire au moment ou on en a besoin, c'est ce que l'on appelle l'allocation dynamique.

Lorsqu'un programme à besoin de mémoire, il peut générer un appel système pour demander au système d'exploitation le besoin de disposer de plus de mémoire. En pratique on utilise deux fonctions de la bibliothèque standard *<stdlib.h>* :

void *malloc(size_t size) Alloue dynamiquement un espace mémoire de `size` bytes. Le terme *malloc* découle de *Memory ALLOCation*.

void *calloc(size_t nitems, size_t size) Fonctionne de façon similaire à `malloc` mais initialise l'espace alloué à zéro.

void free(void *ptr) Libère un espace préalablement alloué par `malloc` ou `calloc`

L'allocation se fait sur le *tas* (*heap*) qui est de taille variable. À chaque fois qu'un espace mémoire est demandé, `malloc` recherche dans le segment un espace vide de taille suffisante, s'il ne parvient pas, il exécute l'appel système sbrk qui permet de déplacer la frontière du segment mémoire et donc d'agrandir le segment.

13.3 Mémoire de programme

Les segments mémoires sont une construction de la bibliothèque standard, selon la bibliothèque utilisée et à fortiori le système d'exploitation utilisé, l'agencement mémoire peut varier.

Néanmoins une bonne représentation est la suivante :

On observe que le tas et la pile vont à leur rencontre, et que lorsqu'ils se percutent c'est le crash avec l'erreur bien connue stack overflow.

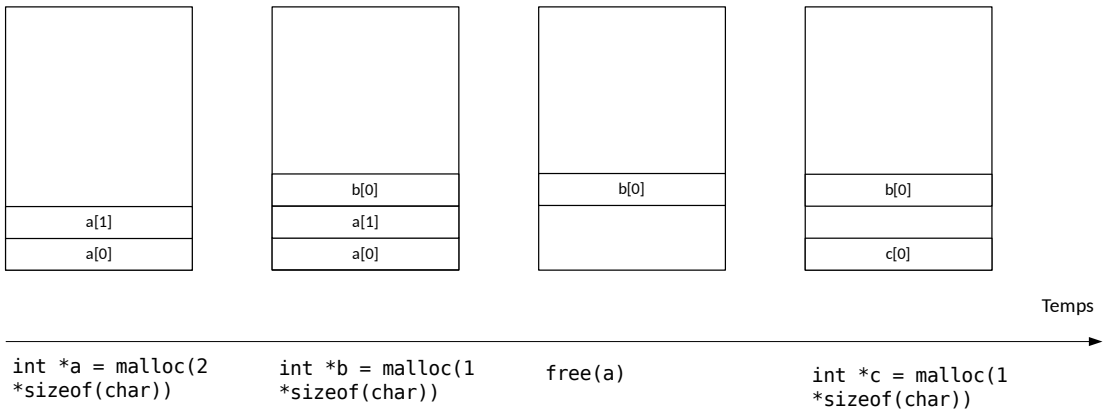


Fig. 13.1 – Allocation et libération mémoire

13.4 La pile

Lorsqu'un programme s'exécute, l'ordre dont les fonctions s'exécutent n'est pas connu à priori. L'ordre d'exécution des fonctions dans l'exemple suivant est inconnu par le programme et donc les éventuelles variables locales utilisées par ces fonctions doivent dynamiquement être allouées.

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

double square(double num) {
    return num * num
}

double cube(double num) {
    return num * num * num;
}

int main(void) {
    double num = 10;

    for (size_t i = 0; i < 10; i++) {
        if (rand() % 2) {
            num = square(num);
        } else {
            num = cube(num);
        }
    }

    printf("%f\n", num);
}
```

Lors d'un appel de fonction, le compilateur ajoute avant la première instruction du code caché permettant d'empiler sur un espace mémoire dédié (*stack*) les variables locales dont

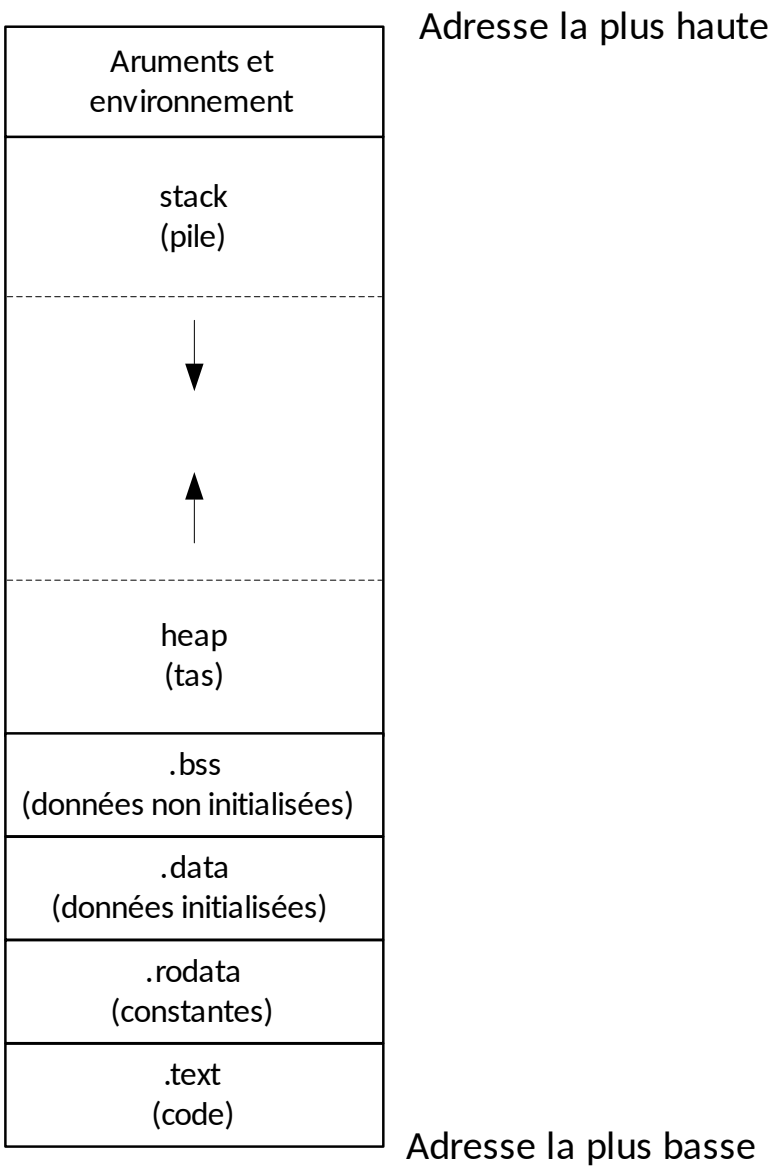


Fig. 13.2 – Organisation de mémoire d'un programme

il a besoin ainsi que certaines informations tel que l'adresse mémoire de retour.

13.5 Allocation dynamique sur le tas

L'allocation dynamique permet de réserver - lors de l'exécution - une zone mémoire dont on vient de calculer la taille. On utilisera la fonction *malloc* (memory allocation) pour réserver de la mémoire. Cette fonction n'initialise pas la zone réservée.

```
typedef unsigned int size_t;
void* malloc(size_t size);
```

Il est nécessaire d'inclure le fichier *stdlib.h* pour utiliser les fonctions d'allocation mémoire. Par exemple, pour réserver un tableau de n valeurs de type *double* :

```
int n;
double * zone_acquisition; // pointeur sur la zone à réserver

n = 100;

zone_acquisition = (double*)malloc(n * sizeof(double));
```

13.5.1 Allocation dynamique sur le tas avec mise à zéro

On utilisera la fonction *calloc* (memory allocation) pour réserver de la mémoire avec initialisation automatique de la zone réservée.

```
void * calloc (size_t count, size_t size);
```

Cette fonction réserve *count* x *size* octets en mémoire et l'initialise à zéro.

13.5.2 Modification de la taille d'une zone déjà allouée sur le tas

Si l'on veut agrandir une zone déjà allouée avec *malloc* ou *calloc*, on utilisera la fonction suivante :

```
void * realloc (void * ptr, size_t size);
```

Elle permet de :

- réallouer un bloc de mémoire avec une nouvelle taille
- si ptr est NULL, créer un nouveau bloc
- si la réallocation échoue, retourner NULL ; le bloc passé en paramètre reste alors inchangé
- en cas de succès, l'adresse retournée peut être différente de ptr ; le bloc initialement pointé par ptr a alors été libéré

— le bloc réalloué est initialisé avec le contenu du bloc ptr ; l'espace supplémentaire est non initialisé

13.5.3 Libération

Le tas n'étant pas extensible à l'infini, il faut libérer la mémoire dés que l'on n'en a plus l'utilité.

```
void free(void *memblock);
```

Une fois libérée, la mémoire (donc son pointeur) ne doit plus être utilisée sous peine de corrompre des données du système.

```
int n;
double * zone_acquisition; // pointeur sur la zone à réserver

n=100;

zone_acquisition = (double*) malloc ( n * sizeof(double) );

// utilisation...

free(zone_acquisition); // libère la mémoire
```

De la même manière, il ne faut pas libérer un bloc qui n'a pas été alloué. Si on ne libère pas la mémoire, elle reste allouée pour l'application et la zone disponible diminue. Il peut arriver qu'il ne reste plus d'espace disponible pour l'allocation dynamique ; cela peut entraver la bonne marche de l'ordinateur. Ce problème est souvent dû à des erreurs de conception des applications qui ne libèrent pas tous les blocs alloués ; on observe alors un phénomène de fuite mémoire qui cause le plantage de la machine. Selon les fréquences d'allocation et de non libération, ces problèmes peuvent survenir immédiatement, ou après plusieurs jours de fonctionnement, ce qui complique grandement les opérations de debug...

13.5.4 Allocation dynamique sur la pile

L'allocation dynamique sur la pile est équivalente à l'allocation sur le tas sauf qu'elle est plus rapide (pas de recherche par le système d'un espace suffisant et continu) et qu'elle ne nécessite pas de libération.

On utilisera la fonction *alloca* (memory allocation) pour réserver de la mémoire. Cette fonction n'initialise pas la zone réservée.

```
void* alloca(size_t size);
```

Il est nécessaire d'inclure le fichier *malloc.h* pour utiliser cette fonction d'allocation mémoire sur la pile. L'espace est libéré à la sortie de la fonction appelante. On veillera tout

particulièrement à ce que le pointeur ayant reçu l'adresse de la zone mémoire réservé ne soit pas exploité en dehors de la fonction (puisque la zone est libérée quand on en sort).

13.5.5 Limite d'utilisation de la pile

L'espace mémoire utilisé par la pile est une zone dont l'usage est uniquement dédié au programme. Si plusieurs programmes cohabitent en mémoire, ils auront chacun leur propre pile.

Cet espace mémoire dédié à la pile est de taille fixe et définie lors de la compilation du programme.

La pile reçoit les éléments suivants :
- les variables locales aux fonctions,
- les variables déclarées comme paramètres dans les fonctions,
- les informations liées au mécanismes d'appel et de retour des fonctions,
- les données retournées par les fonctions,
- les zone allouées par la fonction `alloca`.

Étant donné que la taille de la pile est fixe, il y a un risque qu'elle soit trop petite pour supporter toutes les informations que votre programme doit y placer. Si cela se produit, il y a corruption de la mémoire puisque la pile 'déborde' et que vous dépassez la zone qui lui est dédiée.

Les événements suivants peuvent générer des débordements de pile :
- trop de variables locales (par exemple un grand tableau),
- trop d'appels de fonctions en cascade,
- utilisation de fonctions récursives (qui s'autoappellent).

Dans le jargon informatique, on appelle ça du *jardinage* puisque vous allez piétiner les zones mémoires voisines sans en avoir la permission.

Le compilateur (en réalité l'éditeur de liens - le *linker*) vous permet de spécifier la taille de la pile ; c'est une de ses nombreuses options.

13.6 Variables automatiques

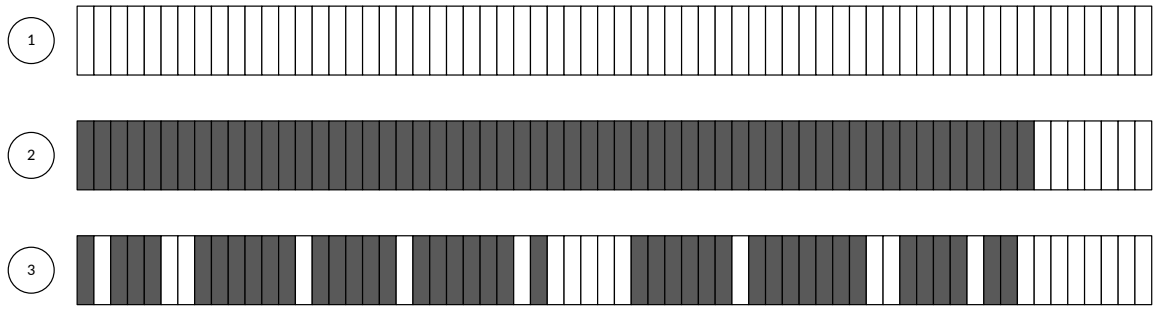
Une variable est dite *automatique* lorsque sa déclaration est faite au sein d'une fonction. La variable d'itération `int i` dans une boucle `for` est dite automatique. C'est à dire que le compilateur à le choix de placer cette variable :
- sur la pile ;
- dans un registre mémoire processeur.

Jadis, le mot clé `register` était utiliser pour forcer le compilateur à placer une variable locale dans un registre processeur pour obtenir de meilleures performances. Aujourd'hui, les compilateurs sont assez malins pour déterminer automatiquement les variables souvent utilisées.

13.7 Fragmentation mémoire

On peut observer à la figure Fig. 13.1 qu'après un appel successif de `malloc` et de `free` des espaces mémoires non utilisés peuvent apparaître entre des régions utilisées. Ces *trous* sont appelés fragmentation mémoire.

Dans la figure suivante, on suit l'évolution de l'utilisation du *heap* au cours de la vie d'un programme. Au début , la mémoire est libre. Tant que de la mémoire est allouée sans libération (`free`), aucun problème de fragmentation . Néanmoins, après un certain temps la mémoire devient fragmentée ; il reste dans cet exemple 2 emplacements de taille 2, un emplacement de taille 5 et un emplacement de taille 8. Il est donc impossible de réserver un espace de taille 9 malgré que l'espace cumulé libre est suffisant.



Dans une petite architecture, l'allocation et la libération fréquente d'espaces mémoire de taille arbitraire est malvenue. Une fois que la fragmentation mémoire est installée, il n'existe aucun moyen de soigner le mal si ce n'est au travers de l'ultime solution de l'informatique : éteindre puis redémarrer.

13.7.1 MMU

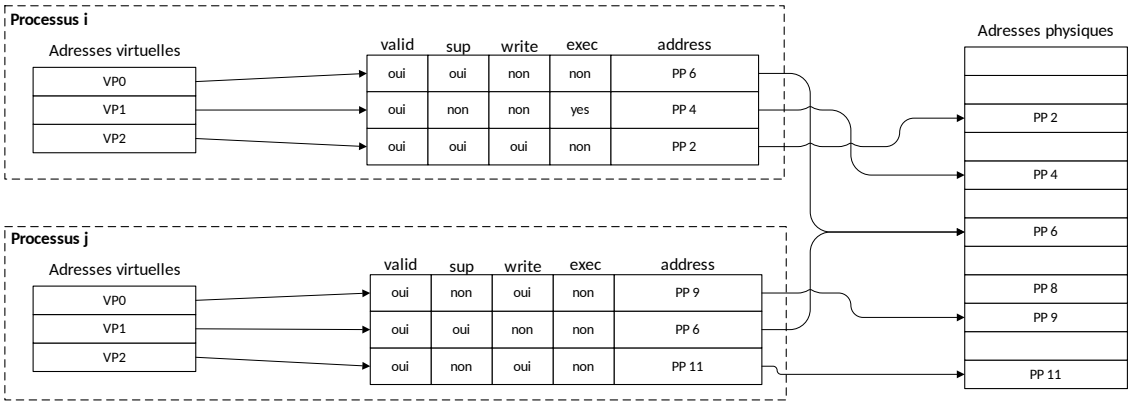
Les systèmes d'exploitations modernes (Windows, Linux, macOS...) utilisent tous un dispositif matériel nommé MMU pour *Memory Management Unit*. La MMU est en charge de créer un espace mémoire **virtuel** entre l'espace physique. Cela crée une indirection supplémentaire mais permet de réorganiser la mémoire physique sans compromettre le système.

En pratique l'espace de mémoire virtuelle est toujours beaucoup plus grand que l'espace physique. Cela permet de s'affranchir dans une large mesure de problèmes de fragmentation car si l'espace virtuel est suffisament grand, il y aura statistiquement plus de chance d'y trouver un emplacement non utilisé.

La programmation sur de petites architectures matérielles (microcontrôleurs, DSP) ne possèdent pas de MMU et dès lors l'allocation dynamique est généralement à proscrire à moins qu'elle soit faite en connaissance de cause et en utilisant des mécanisme comme les *memory pool*.

Dans la figure ci-dessous. La mémoire physique est représentée à droite en termes de pages mémoires physiques (*Physical Pages* ou **PP**). Il s'agit de blocs mémoires contigus d'une

taille fixe, par exemple 64 kB. Chaque page physique est mappée dans une table propre à chaque processus (programme exécutable). On y retrouve quelques proriétés utiles à savoir est-ce que la page mémoire est accessible en écriture, est-ce qu'elle peut contenir du code exécutable ? Une propriété peut indiquer par exemple si la page mémoire est valide. Chacune de ces entrées est considérée comme une page mémoire virtuelle (*virtual page* **VP**).



Erreurs de segmentation (*segmentation fault*)

Lorsqu'un programme tente d'accéder à un espace mémoire qui n'est pas mappé dans la MMU, ou que cet espace mémoire ne permet pas le type d'accès souhaité : par exemple une écriture dans une page en lecture seule. Le système d'exploitation tue le processus avec une erreur *Segmentation Fault*. C'est la raison pour laquelle, il n'est pas systématique d'avoir une erreur de segmentation en cas de jarinage mémoire. Tant que les valeurs modifées sont localisées au sein d'un bloc mémoire autorisé, il n'y aura pas d'erreur.

L'erreur de segmentation est donc générée par le système d'exploitation en lèvant le signal **SIGSEGV** (Violation d'accès à un segment mémoire, où erreur de segmentation).

13.7.2 Memory Pool

Un *memory pool* est une méthode faisant appel à de l'allocation dynamique de blocs de taille fixe. Lorsqu'un programme doit très régulièrement allouer et désalouer de la mémoire, il est préférable que les blocs mémoire ait une taille fixe. De cette façon, après un `free`, la mémoire libérée est assez grande pour une allocation ultérieure.

Lorsqu'un programme est exécuté sous Windows, macOS ou Linux, l'allocation dynamique standard `malloc`, `calloc`, `realloc` et `free` sont performants et le risque de crash dû à une fragmentation mémoire est rare.

En revanche lors de l'utilisation sur de petites architectures (microcontrôleurs) qui n'ont pas de système sophistiqués pour gérer la mémoire, il est parfois nécessaire d'écrire son propre système de gestion de mémoire.

Chapitre 14

Pointeurs

Attention les vélos, on s'attaque à un sujet délicat, difficile, scabreux, mais nécessaire. Un sujet essentiel, indispensable et fantastique : les pointeurs.

Les pointeurs sont des **variables** qui, au lieu de stocker une valeur, stockent une **adresse mémoire**. Dans quel but me direz-vous ? Pour créer des indirections, simplifier l'exécution du code.

Prenons un exemple concret. Le Vicomte de Valmont décide d'écrire à la marquise de Merteuil et il rédige une lettre. Il cachette sa lettre et la dépose dans sa boîte aux lettres pour enlèvement par le facteur moyennant quelques sous. En des termes programmatiques, on a :

```
char lettre[] = "Chère Maquise, ...";
```

Cette variable `lettre` est dès lors enregistrée en mémoire à l'adresse `0x12345abc` qui pourrait correspondre à l'emplacement de la boîte aux lettres du Vicomte.

Le facteur qui ne craint pas la besogne prend connaissance du courrier à livrer, mais constate avec effroi que le cachet de cire à fondu sous l'effet du réchauffement climatique. La lettre est collée au fond de la boîte et il ne parvient pas à la détacher. Pire encore, et ajoutant à la situation déjà cocasse un dramatisme sans équivoque, à forcer de ses maigres doigts le papier de l'enveloppe se déchire et le contenu de lettre indécollable lui est révélée.

Je l'admets volontiers, il me faut bien faire quelques pirouettes pour justifier qu'une valeur en mémoire ne peut être transportée d'un lieu à un autre à simple dos de facteur. Aussi, notre facteur qui est si bon, mais qui n'a plus la mémoire de sa jeunesse, ni papier d'ailleurs, décide de mémoriser la lettre et de la retranscrire chez madame la Marquise qu'il connaît bien. Or comme il est atteint de la maladie de *64-bits* il n'arrive à mémoriser que 8 caractères `Chère Ma`. Sur son bolide, il arrive à destination et retranscrit dans le fond de la boîte de madame de Merteuil les huit caractères fidèlement retranscrits. Comme il est bonnet, mais assidu, il consacre le restant de sa journée en des allers-retours usant la gomme de son tout nickelé jusqu'à ce que toute la lettre ait été retranscrite.

On se retrouve avec une **copie** de la lettre chez madame de Merteil :

```
char valmont_mailbox[] = "Chère Maquise, ...";
char merteil_mailbox[] = "Chère Maquise, ...";
```

La canicule n'étant pas finie, et cette physique discutable ne pouvant être déjouée, la marquise décide de résoudre le problème et se rends à Tarente (un très mauvais choix par jour de canicule) et formule sa réponse sur le mur sud du Castello Aragonese ayant préalablement pris soin de noter la position GPS du mur avec exactitude (`0x30313233`) :

```
char castello_wall[] = "Cher Vicomte, ...";
char (*gps_position)[] = &castello_wall;
```

De retour chez elle, elle prie le facteur de transmettre au vicomte de Valmont ce simple mot : `0x30313233`. Le facteur parvient sans mal à mémoriser les 4 octets du message.

La variable `gps_position` ne contient donc pas le message, mais seulement l'adresse mémoire de ce message. Il s'agit ici d'un **pointeur sur un tableau de caractères**.

Entre temps, le vicomte qui est paresseux s'est équipé d'un téléscripteur capable d'exécuter du code C et il parvient à lire le message de sa complice la marquise.

```
printf("%s", *gps_position);
```

S'il avait oublié l'astérisque (*, U+002A) dans cette dernière ligne il n'aurait pas vu le message espéré, mais simplement `0123` qui correspond au contenu à l'adresse mémoire ou se trouve l'adresse du message (et non le message).

L'astérisque agit donc comme un **déréférencement**, autrement dit, la demande expresse faite au dévoué facteur d'aller à l'adresse donnée récupérer le contenu du message.

Oui, mais, on utilise un astérisque pour déréférencer, mais dans l'exemple précédant on a utilisé l'esperluette (&, U+0026) : `&castello_wall`, pourquoi ? L'esperluette quant elle préfixe une variable peut être traduite par **l'adresse de**. Cela revient à l'étape pendant laquelle la marquise a mesuré la position GPS du mur sur à Tarente.

Il manque encore une chose, il y a aussi une astérisque sur `(*gps_position)[]`. Cela vaudrait-il dire qu'on déréférence la position gps pour affecter l'adresse du mur ? Non, pas du tout... Et c'est d'ailleurs à cette étape que les novices perdent le fil. Où en étais-je ?

Notons qu'il y a plusieurs interprétations de l'astérisque en C :
- Opérateur de multiplication : `a * b`
- Déréférencement d'un pointeur : `*ptr`
- Déclaration d'un pointeur : `int * ptr`

Donc ici, on déclare un pointeur. En appliquant la règle gauche-droite que l'on verra plus bas :

```
char (*gps_position)[]
      ^^^^^^^^^^^^          1. gps_position est
                  ^         2. ...
     ^                      3. un pointeur sur
                   ^^       4. un tableau de
```

(suite sur la page suivante)

(suite de la page précédente)

```
^^^^                          5. caractères
                              6. PROFIT...
```

Résumons :
- Un pointeur est une **variable**
- Il contient une **adresse mémoire**
- Il peut être **déréférencé** pour en obtenir la valeur de l'élément qu'il pointe
- **L'adresse d'une variable** peut être obtenue avec une esperluette

14.1 Pointeur simple

Le format le plus simple d'un pointeur sur un entier s'écrit avec l'asterix * :

```
int* ptr = NULL;
```

La valeur `NULL` corresponds à l'adresse nulle `0x00000000`. On utilise cette convention pour bien indiquer qu'il s'agit d'une adresse et non d'une valeur scalaire.

À tout moment la valeur du pointeur peut être assignée à l'adresse d'un entier puisque nous avons déclaré un pointeur sur un entier :

```
int boiling = 100;
int freezing = 0;

for (char i = 0; i < 10; i++) {
    ptr = i % 2 ? &boiling : &freezing;
    printf("%d", *ptr);
}
```

Lorsque nous avions vu les tableaux, nous écrivions :

```
int array[10] = {0,1,2,3,4,5,6,7,8,9};
```

Vous ne le saviez pas, mais *plot twist* la variable `array` est un pointeur, et la preuve est que `array` peut être déréférencé :

```
printf("%d", *array);
```

La différence entre un **tableau** et un **pointeur** est la suivante :
- Il n'est pas possible d'assigner une adresse à un tableau
- Il n'est pas possible d'assigner des valeurs à un pointeur

D'ailleurs, l'opérateur crochet `[]` n'est rien d'autre qu'un sucre syntaxique :

```
a[b] == *(a + b);
```

14.2 Arithmétique de pointeurs

Fondamentalement un pointeur est une variable qui contient un ordinal, c'est-à-dire qu'il peut être imaginé l'ajout à un pointeur une grandeur finie :

```
char str[] =␣
 ↪"Le vif zéphyr jubile sur les kumquats du clown gracieux";

for (char* ptr = str; *ptr; ptr++) {
    putchar(*ptr);
}
```

Imaginons que l'on souhaite représenter le carré magique suivant :

```
┌───┬───┬───┐
│ 4 │ 9 │ 2 │
├───┼───┼───┤
│ 3 │ 5 │ 7 │
├───┼───┼───┤
│ 8 │ 1 │ 6 │
└───┴───┴───┘
```

On peut le représenter en mémoire linéairement et utiliser de l'arithmétique de pointeur pour le dessiner :

```
char magic[] = "492" "357" "816";

char* ptr = magic;

for (size_t row = 0; row < 3; row++) {
    for (size_t col = 0; col < 3; col++)
        putchar(*(ptr + row * 3 + col));
    putchar('\n');
}
```

Mais ? N'est-ce pas là ce que fait le compilateur lorsque l'adresse les éléments d'un tableau multi dimensionnel ?

```
char magic[][3] = {"792", "357", "816"};

for (size_t row = 0; row < 3; row++) {
    for (size_t col = 0; col < 3; col++)
        putchar(magic[row][col]);
    putchar('\n');
}
```

Oui très exactement, les deux codes sont similaires, mais l'un est plus élégant que l'autre, lequel d'après vous ?

L'arithmétique de pointeur est donc chose courante avec les tableaux. À vrai dire, les

deux concepts sont interchangeables :

Élement	Premier	Deuxième	Troisième	n ième
Accès tableau	`a[0]`	`a[1]`	`a[2]`	`a[n - 1]`
Accès pointeur	`*a`	`*(a + 1)`	`*(a + 2)`	`*(a + n - 1)`

De même, l'exercice peut être répété avec des tabelaux à deux dimensions :

Élement	Premier	Deuxième	n ligne m colonne
Accès tableau	`a[0][0]`	`a[1][1]`	`a[n - 1][m - 1]`
Accès pointeur	`*(*(a+0)+0)`	`*(*(a+1)+1)`	`*(*(a+i-1)+j-1)`

14.3 Pointeur et chaînes de caractères

```
static const char* conjonctions[] = {
    "mais", "ou", "est", "donc", "or", "ni", "car"
};
```

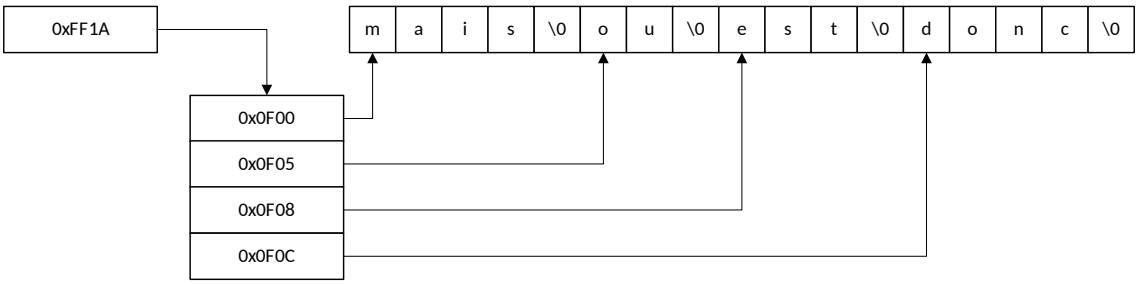


Fig. 14.1 – Pointeur sur une chaîne de caractère

Cette structure est très exactement la même que pour les arguments transmis à la fonction `main` : la définition `char *argv[]`.

14.4 Structures et pointeurs

14.4.1 Initialisation d'un pointeur sur une structure

De la même manière qu'avec les types standards, on peut définir un pointeur sur une structure de donnée.

```
typedef struct {

  unsigned char jour;
```

(suite sur la page suivante)

(suite de la page précédente)

```
  unsigned char mois;
  unsigned int  annee;

} sDate, *pDate;
```

L'exemple précédent définit un type de donnée *sDate* ainsi qu'un pointeur sur le même type de donnée : *pDate*. On pourrait donc initialiser un pointeur sur une structure de la façon suivante :

```
sDate date_depart;
pDate p;            // pointeur sur un type sDate

p=&date_depart;     // initialisation du pointeur sur un type␣
↪structuré
```

14.4.2 Utilisation d'un pointeur sur une structure

On a vu que les champs d'une structure sont accessibles au travers du . faisant la liaison entre la variable et le champs. Cela est valable si la variable est du type structuré. Si la variable est du type pointeur sur une structure, on remplacera le . par $->$.

```
sDate date_depart;
pDate p;            // pointeur sur un type sDate

p=&date_depart;     // initialisation du pointeur sur un type␣
↪structuré

p->jour=29;         // accès aux champs de la structure
p->mois=12;         // depuis  un pointeur
p->annee=1964;
```

14.4.3 Utilisation d'un pointeur récursif sur une structure

Lorsqu'on utilise des listes chaînées, on a besoin de créer une structure contenant des données ainsi qu'un pointeur sur un élément précédent et un autre sur l'élément suivant. Ces pointeurs sont du même type que la structure dans laquelle ils sont déclarés et cela impose un style d'écriture spécifique :

```
typedef struct sElement {

  struct sElement *precedent; // pointeur sur l'élément précédent
  struct sElement *suivant;   // pointeur sur l'élément suivant

  unsigned long data;  // donnée de la liste chaînée

} sElement, *pElement;
```

Exemple d'utilisation :

```
sElement e[3]; // 3 éléments dans la liste

// premier élément de la liste
e[0].precedent = NULL;
e[0].suivant   = &e[1];

// second élément de la liste
e[1].precedent = &e[0];
e[1].suivant   = &e[2];

// troisième élément de la liste
e[2].precedent = &e[1];
e[2].suivant   = NULL;
```

14.4.4 Pointeurs et paramètres de fonctions

Les fonctions comportent une liste de paramètres permettant de retourner une information au programme appelant. Il est souvent indispensable de pouvoir fournir à une fonction des paramètres qu'elle peut modifier lors de son exécution. Pour se faire, on passera par l'utilisation de pointeurs.

14.4.5 Paramètres sous la forme de pointeurs

Le prototype d'une fonction recevant un (ou plusieurs) pointeurs s'écrit de la manière suivante :

```
type fonction(type * param);
```

Cette fonction reçoit un paramètre (*param*) qui est un pointeur sur un type particulier.

Exemple de prototype :

```
int calcul(double x, double * pres);
```

La fonction *calcul* prend 2 paramètres. Le premier (x) est du type double. Le second (*pres*) est un pointeur sur un double. Il sera donc possible, lors de l'appel de la fonction, de lui donner l'adresse d'une variable dans laquelle la fonction placera le résultat du calcul.

```
int calcul(double x, double * pres) {

  *pres = x * 2.;  // calcul du double de x
                   // place le resultat à l'adresse pres

  return 0;        // code retour = 0 (int)
}
```

(suite sur la page suivante)

(suite de la page précédente)

```c
int main() {

  double valeur = 7.;
  double r = 0.;
  int code_ret=0;

  code_ret=calcul (valeur, &r);
  // r vaut maintenant 14.
  return 0;
}
```

Lors de l'appel d'une fonction recevant un pointeur comme paramètre, on placera le symbole & pour lui donner l'adresse de la variable.

14.5 Transtypage de pointeurs (cast)

Le `cast` de pointeur s'avère nécessaire lorsqu'un pointeur du type `void` est déclaré, comme c'est le cas pour la fonction de copie mémoire `memcpy`. En effet, cette fonction accepte en entrée un pointeur vers une région mémoire source, et un pointeur vers une région mémoire de destination. D'un cas d'utilisation à un autre, le format de ces régions mémoires peut être de nature très différente :

```c
char message[] = "Mind the gap, please!";

int array[128];

struct { int a; char b; float c[3] } elements[128];
```

Il faudrait donc autant de fonction `memcpy` que de type possible, ce qui n'est ni raisonnable, ni même imaginable. Face à ce dilemme, on utilise un pointeur neutre, celui qui n'envie personne et que personne n'envie `void` et qui permet sans autre :

```c
void *ptr;

ptr = message;
ptr = array;
ptr = elements;
```

Que pensez-vous que `sizeof(void)` devrait retourner ? Formellement ceci devrait mener à une erreur de compilation, car `void` n'a pas de substance, et donc aucune taille associée. Néanmoins `gcc` est très permissif de base et (à ma grande surprise), il ne génère par défaut ni *warning*, ni erreurs sans l'option `-Wpointer-arith` sur laquelle nous aurons tout le loisir de revenir.

L'intérêt d'un pointeur, c'est justement de pointer une région mémoire et le plus souvent, de la balayer grâce à l'arithmétique de pointeurs. Notre fonction de copie mémoire doit

en somme pouvoir parcourir toute la région mémoire de source et de destination et de ce fait incrémenter le pointeur. Mais, n'ayant aucune taille l'arithmétique de pointeur n'est pas autorisée avec le pointeur `void` et nous voilà bien avancés, notre pointeur ne nous est guère d'usage que son utilité éponyme : rien.

Or, le titre de cette section étant le transtypage, il doit donc y avoir moyen de s'en sortir par une pirouette programmatique dans laquelle je déclare un nouveau pointeur du type char auquel j'associe la valeur de ptr par un **cast explicite**.

```
char *iptr = (char*)ptr;
```

Dès lors, l'arithmétique est redevient possible `iptr++`. Pourquoi ne pas avoir utilisé ce subterfuge plus tôt me direz-vous ? En effet, il m'aurait été possible d'écrire `char *ptr = (char*)elements;` directement et sans détour, mais ceci aurait alors mené à ce prototype-ci :

```
void *memcpy(char* dest, const char* src, size_t n);
```

La clé est dans le standard ISO/IEC 9899 :2011 section 6.3.2.3 page 55 :

> A pointer to void may be converted to or from a pointer to any object type. A pointer to any object type may be converted to a pointer to void and back again ; the result shall compare equal to the original pointer.

Autrement dit, il n'est pas nécessaire, ni recommandé de faire un transtypage explicite pour convertir vers et en provenance d'un pointeur sur `void`. Et donc, l'astuce de memcpy est que la fonction accepte n'importe quel type de pointeur et c'est le message autodocumenté du code.

Et quant à l'implémentation de cette fonction me direz-vous ? Une possibilité serait :

```
void memcpy(void *dest, void *src, size_t n)
{
    char* csrc = src;
    char* cdest = dest;

    for (size_t i = 0; i < n; i++)
        cdest[i] = csrc[i];
}
```

Où plus concis :

```
void memcpy(void *dest, void *src, size_t n)
{
    for (size_t i = 0; i < n; i++)
        ((char*)dst)[i] = ((char*)src)[i];
}
```

Or, rien de tout ceci n'est juste. `memcpy` est une fonction fondamentale en C, ce pourquoi nous nous y attardons temps. Elle est constamment utilisée et doit être extrêmement performante. Aussi, si le compilateur cible une architecture 64-bits pourquoi diable copier les éléments par paquet de 8-bits. C'est un peu comme si notre facteur, au début de ce

chapitre, aurait fait ses allers-retours avec en tête qu'un octet par trajet. L'implémentation dépend donc de l'architecture cible et doit tenir compte des éventuels effets de bords. Par exemple s'il faut copier un tableau de 9 x 32 bits. Une architecture 64-bits aura une grande facilité à copier les 8 premiers octets, mais quant au dernier, il s'agit d'un cas particulier et selon la taille de la copie et l'architecture du processeur, l'implémentation devra être ajustée. C'est pourquoi ce type très bas niveau de fonction est l'affaire d'une cuisine interne du compilateur et dont le développeur ne doit pas se soucier. Vous êtes comme Thomas l'apôtre, et ne me croyez pas ? Alors, digressons et essayons :

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

int main(void)
{
    char a[] = "La Broye c'est fantastique!";
    char b[sizeof(a)];

    memcpy(a, b, sizeof(a));

    printf("%s %s", a, b);
}
```

On observe qu'il n'y a aucun appel de fonction à `memcpy` comme c'est le cas pour `printf` (`bl printf`). La copie tient place en 6 instructions.

```
main :
    // Entry
    str     lr, [sp, #-4]!
    sub     sp, sp, #60

    // Inline memcpy
    mov     ip, sp                  // Destination address
    add     lr, sp, #28             // Source address (char b␣
 ↪located 28 octets after a)

    ldmia   lr!, {r0, r1, r2, r3}   // Load 4 x 32-bits
    stmia   ip!, {r0, r1, r2, r3}   // Store 4 x 32-bits

    ldm     lr, {r0, r1, r2}        // Load 3 x 32-bits
    stm     ip, {r0, r1, r2}        // Store 3 x 32-bits

    // Display (printf)
    add     r2, sp, #28
    mov     r1, sp
    ldr     r0, .L4
    bl      printf

    // Exit
    mov     r0, #0
    add     sp, sp, #60
```

(suite sur la page suivante)

(suite de la page précédente)

```
    ldr     pc, [sp], #4
.L4 :
    .word   .LC0
.LC0 :
    .ascii  "La Broye c'est fantastique!\000"
```

Vous pouvez jouer avec cet exemple ici.

14.6 Pointeurs de fonctions

Un pointeur peut pointer n'importe ou en mémoire, et donc il peut également pointer non pas sur une variable, mais sur une fonction. Les pointeurs de fonctions sont très utiles pour des fonctions de rappel (callback).

Par exemple on veut appliquer une transformation sur tous les éléments d'un tableau, mais la transformation n'est pas connue à l'avance. On pourrait écrire :

```c
int is_odd(int n)
{
    return !(n % 2);
}

void map(int array[], int (*callback)(int), size_t length)
{
    for (size_t i = 0; i < length; i++) {
        array[i] = callback(array[i]);
    }
}

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

    map(array, is_odd);
}
```

Avec la règle gauche droite on parvient à décortiquer la déclaration :

```
int (*callback)(int)
      ^^^^^^^^          callback is
              ^
     ^                  a pointer on
               ^^^^^    a function taking an int
^^^                     and returning an int
```

14.7 La règle gauche-droite

Cette règle est une recette magique permettant de correctement décortiquer une déclaration C contenant des pointeurs.

Il faut tout d'abord lire :

Symbole	Traduction	Direction
`* [] ()`	`pointeur sur tableau de fonction retournant`	Toujours à gauche Toujours à droite Toujours à droite

14.7.1 Première étape

Trouver l'identifiant et se dire `L'identifiant est`.

14.7.2 Deuxième étape

Chercher le symbole à droite de l'identifiant. Si vous trouvez un `()`, vous savez que cet identifiant est une fonction et vous avez `L'identifiant est une fonction retournant`. Si vous trouvez un `[]` vous dites alors `L'identifiant est un tableau de`. Continuez à droite jusqu'à ce que vous êtes à court de symboles, **OU** que vous trouvez une parenthèse fermante `)`.

14.7.3 Troisième étape

Regardez le symbole à gauche de l'identifiant. S'il n'est aucun des symboles précédents, dites quelque chose comme `int`. Sinon, convertissez le symbole en utilisant la table de correspondance. Continuez d'aller à **gauche** jusqu'à ce que vous êtes à court de symboles **OU** que vous rencontrez une parenthèse ouvrante `(`.

Continuez les étapes 2 et 3 jusqu'à ce que vous avez une déclaration complète.

14.7.4 Exemples

```
int *p[];
```

1. Trouver l'identifiant : `p` : `p est`

   ```
   int *p[];
        ^
   ```

2. Se déplacer à **droite** : `p est un tableau de`

```
int *p[];
     ^^
```

3. Se déplacer à **gauche** : `p est un tableau de pointeurs sur`

```
int *p[];
    ^
```

4. Continuer à **gauche** : `p est un tableau de pointeurs sur un int`

```
int *p[];
^^^
```

14.7.5 cdecl

Il existe un programme nommé cdecl qui permet de décoder de complexes déclaration c :

```
$ cdecl 'char (*(*x[3])())[5]'
declare x as array 3 of pointer to function returning pointer to␣
 ↪array 5 of char
```

Une version en-ligne est également disponible.

14.8 Initialisation par transtypage

L'utilisation de structure peut être utile pour initialiser un type de donnée en utilisant un autre type de donnée. Nous citons ici deux exemples.

```
int i = *(int*)(struct { char a; char b; char c; char d; }){'a',␣
 ↪'b', 'c', 'd'};
```

```
union {
    int matrix[10][10];
    int vector[100];
} data;
```

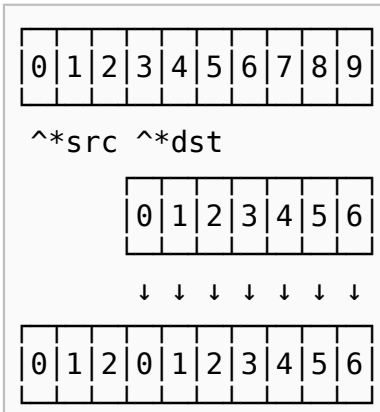
14.9 Enchevêtrement ou *Aliasing*

Travailler avec les pointeurs demande une attention particulière à tous les problème d'*alisasing* dans lesquels différents pointeurs pointent sur une même région mémoire.

Mettons que l'on souhaite simplement déplacer une région mémoire vers une nouvelle région mémoire. On pourrait implémenter le code suivant :

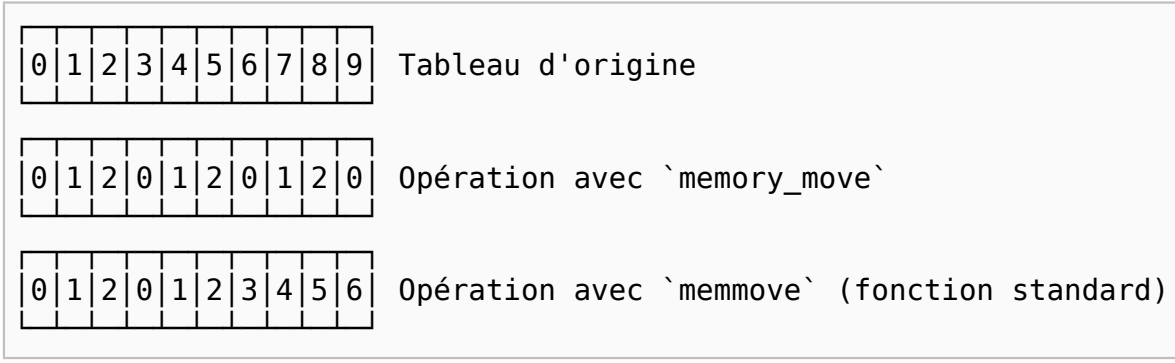
```c
void memory_move(char *dst, char*src, size_t size) {
    for (int i = 0; i < size; i++)
        *dst++ = *src++;
}
```

Ce code est très simple mais il peut poser problème selon les cas. Imaginons que l'on dispose d'un tableau simple de dix éléments et de deux pointeurs `*src` et `*dst`. Pour déplacer la région du tableau de 4 éléments vers la droite. On se dirait que le code suivant pourrait fonctionner :



Naïvement l'exécution suivante devrait fonctionner, mais les deux pointeurs source et destination s'enchevêtrent et le résultat n'est pas celui escompté.

```c
char array[10] = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9};
char *src = &array[0];
char *dst = &array[3];

memory_move(b, a, 7);
```



Notre simple fonction de déplacement mémoire ne fonctionne pas avec des régions mémoire qui s'enchevêtrent. En revanche, la fonction standard `memmove` de `<stdlib.h>` fonctionne car elle autorise, au détriment d'une plus grande complexité, de gérer ce type de situation.

Notons que sa fonction voisine `memcpy` ne dois **jamais** être utilisée en cas d'*aliasing*. Cette fonction se veut performante, c'est à dire qu'elle peut être implémentée en suivant le même principe que notre exemple `memory_move`. Le standard **C99** ne défini pas le

comportement de `memcpy` pour des pointeurs qui se chevauchent.

Exercice

Quel est le type de :

```
*&*&*&*&*&*&(int)x;
```

Exercice

Donnez lesvaleurs affichées par ce programme pour les variables `a` à `e`.

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

int test(int a, int * b, int * c, int * d) {
    a = *b;
    *b = *b + 5;
    *c = a + 2;
    d = c;
    return *d;
}

int main(void) {
    int a = 0, b = 100, c = 200, d = 300, e = 400;
    e = test(a, &b, &c, &d);
    printf("a:%d, b:%d, c:%d, d:%d, e:%d\n", a, b, c, d, e);
}
```

```
a:0, b:105, c:102, d:300, e:102
```

Chapitre 15

Bibliothèques

Fig. 15.1 – Bibliothèque du Trinity College de Dublin

Une bibliothèque informatique est une collection de fichiers comportant des fonctionnalités logicielles prêtes à l'emploi. La fonction `printf` est une de ces fonctionnalités et offerte par le header `<stdio.h>` faisant partie de la bibliothèque `libc6`.

L'anglicisme *library*, plus court à prononcer et à écrire est souvent utilisé en lieu et place de bibliothèque tant il est omniprésent dans le monde logiciel. Le terme `<stdlib.h>` etant la concaténation de *standard library* par exemple. Notez que librairie n'est pas la traduction correcte de *library* qui est un faux ami.

Une *library*, à l'instar d'une bibliothèque, contient du contenu (livre écrit dans une langue donnée) et un index (registre). En informatique il s'agit d'un fichier binaire compilé pour une architecture donnée ainsi qu'un ou plusieurs fichiers d'en-tête (*header*) contenant les définitions de cette bibliothèque.

Dans ce chapitre on donnera plusieurs exemples sur un environnement POSIX. Sous Windows, les procédures choses sont plus compliquées, mais les concepts restent les mêmes.

15.1 Exemple : libgmp

Voyons ensemble le cas de libgmp. Il s'agit d'une bibliothèque de fonctionnalités très utilisée et permettant le calcul arithmétique multiprécision en C. En observant le détail du paquet logiciel Debian on peut lire que `libgmp` est disponible pour différentes architectures `amd64`, `arm64`, `s390x`, `i386`, ... Un développement sur un Raspberry-PI nécessitera `arm64` alors qu'un développement sur un PC utilisera `amd64`. En cliquant sur l'architecture désirée on peut voir que ce paquet se compose des fichiers suivants (list réduite aux fichiers concernant C :

```
# Fichier d'en-tête C
/usr/include/x86_64-linux-gnu/gmp.h

# Bibliothèque compilée pour l'architecture visée (ici amd64)
/usr/lib/x86_64-linux-gnu/libgmp.a
/usr/lib/x86_64-linux-gnu/libgmp.so

# Documentation de la libgmp
/usr/share/doc/libgmp-dev/AUTHORS
/usr/share/doc/libgmp-dev/README
/usr/share/doc/libgmp-dev/changelog.gz
/usr/share/doc/libgmp-dev/copyright
```

On a donc :

`gmp.h` Fichier d'en-tête à include dans un fichier source pour utiliser les fonctionnalités

`libgmp.a` Bibliothèque **statique** qui contient l'implémentation en langage machine des fonctionnalités à référer au *linker* lors de la compilation

`libgmp.so` Bibliothèque **dynamique** qui contient aussi l'implémentation en langage machine des fonctionnalités

Imaginons que l'on souhaite bénéficier des fonctionnalités de cette bibliothèque pour le calcul d'orbites pour un satellite d'observation de Jupyter. Pour prendre en main cet *libary* on écrit ceci :

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

int main(void)
{
    unsigned int radix = 10;
    char a[] = "19810983098510928501928599999999999990";

    mpz_t n;
```

(suite sur la page suivante)

(suite de la page précédente)

```
    mpz_init(n);
    mpz_set_ui(n, 0);

    mpz_set_str(n, a, radix);

    mpz_out_str(stdout, radix, n);
    putchar('\n');

    mpz_add_ui(n, n, 12); // Addition

    mpz_out_str(stdout, radix, n);
    putchar('\n');

    mpz_mul(n, n, n); // Square

    mpz_out_str(stdout, radix, n);
    putchar('\n');

    mpz_clear(n);
}
```

Puis on compile :

```
$ gcc gmp.c
gmp.c:1:10: fatal error: gmp.h: No such file or directory
#include <gmp.h>
         ^~~~~~~
compilation terminated.
```

Aïe ! La bibliothèque n'est pas installée.

Debian/Ubuntu

```
$ sudo apt-get install libgmp-dev
```

Mac OS X

```
$ brew install gmp
```

Windows

```
ERREUR 404
```

Deuxième tentative :

```
$ gcc gmp.c
/tmp/cc2FxDSy.o: In function `main':
gmp.c:(.text+0x6f): undefined reference to `__gmpz_init'
gmp.c:(.text+0x80): undefined reference to `__gmpz_set_ui'
```

(suite sur la page suivante)

(suite de la page précédente)

```
gmp.c:(.text+0x96): undefined reference to `__gmpz_set_str'
gmp.c:(.text+0xb3): undefined reference to `__gmpz_out_str'
gmp.c:(.text+0xd5): undefined reference to `__gmpz_add_ui'
gmp.c:(.text+0xf2): undefined reference to `__gmpz_out_str'
gmp.c:(.text+0x113): undefined reference to `__gmpz_mul'
gmp.c:(.text+0x130): undefined reference to `__gmpz_out_str'
gmp.c:(.text+0x146): undefined reference to `__gmpz_clear'
collect2: error: ld returned 1 exit status
```

Cette fois-ci on peut lire que le compilateur à fait sont travail, mais ne parvient pas à trouver les symboles des fonctions que l'on utilise p.ex. `__gmpz_add_ui`. C'est normal parce que l'on n'a pas renseigné la bibliothèque à utiliser.

```
$ gcc gmp.c -lgmp

$ ./a.out
19810983098510928501928599999999999990
19810983098510928501928600000000000002
39247505132948566943624895793968860349316343035404371400771440000000000000004
```

Cette manière de faire utilise le fichier `libgmp.so` qui est la bibliothèque **dynamique**, c'est-à-dire que ce fichier est nécessaire pour que le programme puisse fonctionner. Si je donne mon exécutable à un ami qui n'as pas install libgmp sur son ordinateur, il ne sera pas capable de l'exécuter.

Alternativement on peut compiler le même programme en utilisant la librairie **statique**

```
$ gcc gmp.c /usr/lib/x86_64-linux-gnu/libgmp.a
```

c'est à dire qu'à la compilation toutes les fonctionnalités ont été intégrées à l'exécutable et il ne dépend de plus rien d'autre que le système d'exploitation. Je peux prendre ce fichier le donner à quelqu'un qui utilise la même architecture et il pourra l'exécuter. En revanche, la taille du programme est plus grosse :

```
# ~167 KiB
$ gcc gmp.c -l:libgmp.a
$ size a.out
text    data     bss     dec     hex filename
155494     808      56  156358   262c6 ./a.out

# ~8.5 KiB
$ gcc gmp.c -lgmp
$ size a.out
text    data     bss     dec     hex filename
2752     680      16    3448     d78 ./a.out
```

15.2 Exemple : ncurses

La bibliothèque ncurses traduction de *nouvelles malédictions* est une évolution de curses développé originellement par Ken Arnold . Il s'agit d'une bibliothèque pour la création d'interfaces graphique en ligne de commande, toujours très utilisée.

La bibliothèque permet le positionnement arbitraire dans la fenêtre de commande, le dessin de fenêtres, de menus, d'ombrage sous les fenêtres, de couleurs ...

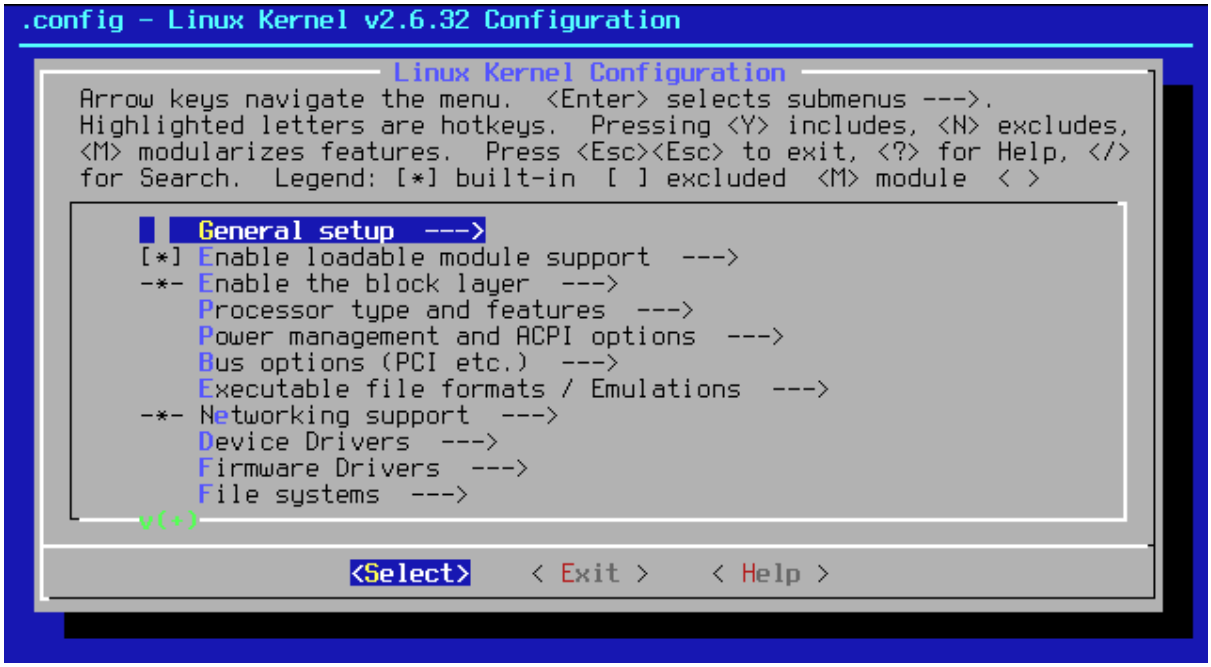


Fig. 15.2 – Exemple d'interface graphique écrite avec *ncurses*. Ici la configuration du noyau Linux.

L'écriture d'un programme Hello World avec cette bibliothèque pourrait être :

```c
#include <ncurses.h>

int main()
{
    initscr();              // Start curses mode
    printw("hello, world"); // Print Hello World
    refresh();              // Print it on to the real screen
    getch();                    // Wait for user input
    endwin();               // End curses mode

    return 0;
}
```

La compilation n'est possible que si :

1. La bibliothèque est installée sur l'ordinateur
2. Le lien vers la bibliothèque dynamique est mentionné à la compilation

```
$ gcc ncurses-hello.c -ohello -lncurses
```

15.3 Bibliothèques statiques

Une *static library* est un fichier binaire compilé pour une architecture donnée et portant les extensions :

- `.a` sur un système POSIX (Android, Mac OS, Linux, Unix)
- `.lib` sous Windows

Une bibliothèque statique n'est rien d'autre qu'une archive d'un ou plusieurs objets. Rappelons-le un objet est le résultat d'une compilation.

Par exemple si l'on souhaite écrire une bibliothèque statique pour le code de César on écrira un fichier source *caesar.c* :

```c
#include <stdio.h>

void caesar(char str[], unsigned key)
{
    key %= 26;

    for (size_t i = 0; str[i]; i++)
    {
        char c = str[i];

        if (c >= 'a' && c <= 'z')
        {
            str[i] = ((c + key > 'z') ? c - 'z' + 'a' - 1 : c) +␣
↪key;
        }
        else if (c >= 'A' && c <= 'Z')
        {
            str[i] = ((c + key > 'Z') ? c - 'Z' + 'A' - 1 : c) +␣
↪key;
        }
    }
}
```

Ainsi qu'un fichier d'en-tête *caesar.h* :

```c


/**
 * Function that compute the Caesar cipher
 * @param str input string
 * @param key offset to add to each character
 */
void caesar(char str[], unsigned key);
```

Pour créer une bibliothèque statique rien de plus facile. Le compilateur crée l'objet, l'archiveur crée l'amalgame :

```
$ gcc -c -o caesar.o caesar.c
$ ar rcs caesar.a caesar.o
```

Puis il suffit d'écrire un programme pour utiliser cette bibliothèque :

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

#define KEY 13

int main(int argc, char *argv[])
{
    for (int i = 1; i < argc; i++)
    {
        caesar(argv[i], KEY);
        printf("%s\n", argv[i]);
    }
}
```

Et de compiler le tout. Ici on utilise `-I.` et `-L.` pour dire au compilateur de chercher le fichier d'en-tête et la bibliothèque dans le répertoire courant.

```
$ gcc encrypt.c -I. -L. -l:caesar.a
```

La procédure sous Windows est plus compliquée et ne sera pas décrite ici.

15.4 Bibliothèques dynamiques

Une *dynamic library* est un fichier binaire compilé pour une architecture donnée et portant les extensions :

— `.so` sur un système POSIX (Android, Mac OS, Linux, Unix)
— `.dll` sous Windows

L'avantage principal est de ne pas charger pour rien chaque exécutable compilé de fonctionnalités qui pourraient très bien être partagées. L'inconvénient est que l'utilisateur du programme doit impérativement avoir installé la bibliothèque. Dans un environnement POSIX les bibliothèques dynamiques disposent d'un emplacement spécifique ou elles sont toute stockées. Malheureusement sous Windows le consensus est plus partagé et il n'est pas rare de voir plusieurs applications différentes héberger une copie des *dll* localement si bien que l'avantage de la bibliothèque dynamique est anéanti par un défaut de cohérence.

Reprenant l'exemple de César vu plus haut, on peut créer une bibliothèque dynamique :

```
$ gcc -shared -o libcaesar.so caesar.o
```

Puis compiler notre programme pour utiliser cette bibliothèque. Avec une bibliothèque dynamique, il faut spécifier au compilateur quels sont les chemins vers lesquels il pourra

trouver les bibliothèques installées. Comme ici on ne souhaite pas **installer** la bibliothèque et la rendre disponible pour tous les programmes, il faut ajouter aux chemins par défaut, le chemin local `$(pwd .)`, en créant une **variable d'environnement** nommée `LIBRARY_PATH`.

```
$ LIBRARY_PATH=$(pwd .) gcc encrypt.c -I. -lcaesar
```

Le problème est identique à l'exécution, car il faut spécifier (ici avec `LD_LIBRARY_PATH`) le chemin ou le système d'exploitation s'attendra à trouver la bibliothèque.

```
$ LD_LIBRARY_PATH=$(pwd .) ./a.out ferrugineux
sreehtvarhk
```

Car sinon c'est l'erreur :

```
$ LIBRARY_PATH=$(pwd .) ./a.out Hey?
./a.out: error while loading shared libraries: libcaesar.so :
cannot open shared object file: No such file or directory
```

15.5 Bibliothèques standard

Les bibliothèques standard (C standard library) sont une collection normalisée d'en-têtes portables. C'est à dire que quelque soit le compilateur et l'architecture cible, cette collection sera accessible.

Le standard **C99** définit un certain nombre d'en-têtes dont les plus utilisés (et ceux utilisés dans ce cours) sont :

<assert.h> Contient la macro `assert` pour valider certains prérequis.

<complex.h> Pour manipuler les nombres complexes

<float.h> Contient les constantes qui définissent la précision des types flottants sur l'architecture cible. `float` et `double` n'ont pas besoin de cet en-tête pour être utilisés.

<limits.h> Contient les constantes qui définissent les limites des types entiers.

<math.h> Fonctions mathématiques `sin`, `cos`, ...

<stdbool.h> Défini le type booléen et les constantes `true` et `false`.

<stddef.h> Défini certaines macros comme `NULL`

<stdint.h> Défini les types standard d'entiers (`int32_t`, `int_fast64_t`, ...).

<stdio.h> Permet l'accès aux entrées sorties standard (`stdin`, `stdout`, `stderr`). Défini entre autre la fonction `printf`.

<stdlib.h> Permet l'allocation dynamique et défini `malloc`

<string.h> Manipulation des chaînes de caractères

<time.h> Accès au fonctions lecture et de conversion de date et d'heure.

Exercice

La fonction Arc-Cosinus `acos` est-elle définie par le standard et dans quel fichier d'en-tête est-elle déclarée ? Un fichier d'en-tête se termine avec l'extension `.h`.

En cherchant `man acos header` dans Google, on trouve que la fonction `acos` est définie dans le header `<math.h>`.

Une autre solution est d'utiliser sous Linux la commande `apropos` :

```
$ apropos acos
acos (3)     - arc cosine function
acosf (3)    - arc cosine function
acosh (3)    - inverse hyperbolic cosine function
acoshf (3)   - inverse hyperbolic cosine function
acoshl (3)   - inverse hyperbolic cosine function
acosl (3)    - arc cosine function
cacos (3)    - complex arc cosine
cacosf (3)   - complex arc cosine
cacosh (3)   - complex arc hyperbolic cosine
cacoshf (3)  - complex arc hyperbolic cosine
cacoshl (3)  - complex arc hyperbolic cosine
cacosl (3)   - complex arc cosine
```

Le premier résultat permet ensuite de voir :

```
$ man acos | head -10
ACOS(3)    Linux Programmer's Manual         ACOS(3)

NAME
    acos, acosf, acosl - arc cosine function

SYNOPSIS
    #include <math.h>

    double acos(double x);
    float acosf(float x);
```

La réponse est donc *<math.h>*.

Sous Windows avec Visual Studio, il suffit d'écrire `acos` dans un fichier source et d'appuyer sur `F1`. L'IDE redirige l'utilisateur sur l'aide Microsoft acos-acosf-acosl qui indique que le header source est `<math.h>`. Exercice

Lors du formatage d'une date, on y peut y lire `%w`, par quoi sera remplacé ce *token* ?

15.5.1 Fonctions d'intérêt

Il serait inutile ici de lister toutes les fonctions, les bibliothèques standard étant largement documentées sur internet. Il ne fait aucun doute que le développeur sera trouver comment calculer un sinus avec la fonction `sin`. Néanmoins l'existence de certaines fonctions peut passer inaperçues et c'est de celles-ci don't j'aimerais parler.

Math

Constantes	Description
`M_PI`	Valeur de π
`M_E`	Valeur de e
`M_SQRT1_2`	Valeur de $1/\sqrt{(2)}$

Fonction	Description
`exp(x)`	e^x
`ldexp(x,n)`	$x \cdot 2^n$
`log(x)`	$\log_2(x)$
`log10(x)`	$\log_{10}(x)$
`pow(x,y)`	x^y
`sqrt(x)`	$\sqrt{(x)}$
`cbrt(x)`	$\sqrt[3]{(x)}$
`hypot(x,y)`	$\sqrt{(x^2+y^2)}$
`ceil`	Arrondi à l'entier supérieur
`floor`	Arrondi à l'entier inférieur

Chaînes de caractères

Fonction	Description
`strcopy(dst, src)`	Identique à `memcpy` mais sans nécessité de donner la taille de la chaîne puisqu'elle se termine par `\0`
`memmove(dst, src, n)`	Identique à `memcpy` mais traite les cas particuliers lorsque les deux régions mémoire se superposent.

Types de données

Test d'une propriété d'un caractère passé en paramètre

Fonction	Description
`isalnum`	une lettre ou un chiffre
`isalpha`	une lettre
`iscntrl`	un caractère de commande
`isdigit`	un chiffre décimal
`isgraph`	un caractère imprimable ou le blanc
`islower`	une lettre minuscule
`isprint`	un caractère imprimable (pas le blanc)
`ispunct`	un caractère imprimable pas isalnum
`isspace`	un caractère d'espace blanc
`isupper`	une lettre majuscule
`isxdigit`	un chiffre hexadécimal

Limites

Valeurs limites pour les entiers signés et non signés

Constante	Valeur
`SCHAR\_MIN`	-128
`SCHAR\_MAX`	+127
`CHAR\_MIN`	0
`CHAR\_MAX`	255
`SHRT\_MIN`	-32768
`SHRT\_MAX`	+32767
`USHRT\_MAX`	65535
`LONG\_MIN`	-2147483648
`LONG\_MAX`	+2147483647
`ULONG\_MAX`	+4294967295
`DBL\_MAX`	1E+37 ou plus
`DBL\_EPSILON`	1E-9 ou moins

15.6 Autres bibliothèques

— **GNU C Library (glibc)**
  — C11
  — POSIX.1-2008
  — IEEE 754-2008

15.6.1 POSIX C Library

Le standard C ne définit que le minimum vital et qui est valable sur toutes les architectures pour autant que la *toolchain* soit compatible **C99**. Il existe néanmoins toute une collection d'autres fonctions manquantes :

- **La communication entre les processus (deux programmes qui souhaitent communic**
  - `<sys/socket.h>`
  - `<sharedmemory.h>`
- **La communicaton sur le réseau e.g. internet**
  - `<sys/socket.h>`
  - `<arpa/inet.h>`
  - `<net/if.h>`
- **Les tâches**
  - `<thread.h>`
- **Les traductions de chaînes p.ex. français vers anglais**
  - `<iconv.h>`
- **Les fonctions avancées de recherche de texte**
  - `<regex.h>`
- **Le log centralisé des messages (d'erreur)**
  - `<syslog.h>`

Toutes ces bibliothèques additionnelles ne sont pas nécessairement disponibles sur votre ordinateur ou pour le système cible, surtout si vous convoitez une application *bare-metal*. Elles dépendent grandement du système d'exploitation utilisé, mais une tentative de normalisation existe et se nomme POSIX (ISO/IEC 9945).

Généralement la vaste majorité des distributions Linux et Unix sont compatibles avec le standard POSIX et les bibliothèques ci-dessus seront disponibles à moins que vous ne visiez une architecture différente de celle sur laquelle s'exécute votre compilateur.

Le support POSIX sous Windows (Win32) n'est malheureusement que partiel et il n'est pas standardisé.

Un point d'entrée de l'API POSIX est la bibliothèque `<unistd.h>`.

15.6.2 GNU GLIBC

La bibliothèque portable GNULIB est la bibliothèque standard référencée sous Linux par `libc6`.

15.6.3 Windows C library

La bibliothèque Windows Windoes API offre une interface au système de fichier, au registre windows, aux imprimantes, à l'interface de fenêtrage, à la console et au réseau.

L'accès à cet API est offert par un unique point d'entrée windows.h qui regroupe certains en-têtes standards (`<stdarg.h>`, `<string.h>`, ...), mais pas tous ( ) ainsi que les en-têtes spécifiques à Windows tels que :

`<winreg.h>` Pour l'accès au registre Windows

`<wincon.h>` L'accès à la console

La documentation est disponible en ligne depuis le site de Microsoft, mais n'est malheureusement pas complète et souvent il est difficile de savoir sur quel site trouver la bonne version de la bonne documentation. Par exemple, il n'y a aucune documentation claire de *LSTATUS* pour la fonction *RegCreateKeyExW <https ://docs.microsoft.com/en-us/windows/win32/api/winreg/nf-winreg-regcreatekeyexw>* permettant de créer une entrée dans la base de registre.

Un bon point d'entrée est le Microsoft API and reference catalog.

Quelques observations :

- Officiellement Windows est compatible avec C89 (ANSI C) (c.f. C Language Reference)
- L'API Windows n'est pas officiellement compatible avec C99, mais elle s'en approche, il n'y pas ou peu de documents expliquant les différences.
- Microsoft n'a aucune priorité pour développer son support C, il se focalise davantage sur C++ et C#, c'est pourquoi certains éléments du langage ne sont pas ou peu documentés.
- Les types standards Windows différent de ceux proposés par C99. Par exemple `LONG32` remplace `int32_t`.

15.7 MacOS X Library

/usr/lib/libSystem.B.dylib Exercice

La fonction Arc-Cosinus `acos` est-elle définie par le standard et dans quel fichier d'en-tête est-elle déclarée ? Un fichier d'en-tête se termine avec l'extension `.h`.

En cherchant `man acos header` dans Google, on trouve que la fonction `acos` est définie dans le header `<math.h>`.

Une autre solution est d'utiliser sous Linux la commande `apropos` :

```
$ apropos acos
acos (3)     - arc cosine function
acosf (3)    - arc cosine function
acosh (3)    - inverse hyperbolic cosine function
acoshf (3)   - inverse hyperbolic cosine function
acoshl (3)   - inverse hyperbolic cosine function
acosl (3)    - arc cosine function
cacos (3)    - complex arc cosine
cacosf (3)   - complex arc cosine
cacosh (3)   - complex arc hyperbolic cosine
cacoshf (3)  - complex arc hyperbolic cosine
cacoshl (3)  - complex arc hyperbolic cosine
cacosl (3)   - complex arc cosine
```

Le premier résultat permet ensuite de voir :

```
$ man acos | head -10
ACOS(3)    Linux Programmer's Manual         ACOS(3)

NAME
    acos, acosf, acosl - arc cosine function

SYNOPSIS
    #include <math.h>

    double acos(double x);
    float acosf(float x);
```

La réponse est donc *<math.h>*.

Sous Windows avec Visual Studio, il suffit d'écrire `acos` dans un fichier source et d'appuyer sur `F1`. L'IDE redirige l'utilisateur sur l'aide Microsoft acos-acosf-acosl qui indique que le header source est `<math.h>`.

Chapitre 16

Préprocesseur

Comme nous l'avons vu en introduction (c.f. Section 1.6), le langage C est basé sur une double grammaire, c'est-à-dire qu'avant la compilation du code, un autre processus est appelé visant à préparer le code source avant la compilation.

Le coeur de cette opération est appelé **préprocesseur**. Les instructions du préprocesseur C sont faciles à reconnaître, car elles débutent toutes par le croisillon (**#**), *hash* en anglais et utilisé récemment comme hashtag sur les réseaux sociaux. Notons au passage que ce caractère était historiquement utilisé par les Anglais sous le dénominatif *pound* (livre). Lorsqu'il est apparu en Europe, il a été confondu avec le caractère dièse (♯) présent sur les pavés numériques de téléphone.

Le vocabulaire du préprocesseur est le suivant :

`#include` Inclu un fichier dans le fichier courant

`#define` Crée une définition (Macro)

`#undef` Détruis une définition existante

`#if defined` ou **`#ifdef`** Teste si une définition existe

`#if` .. **`#elif`** .. **`#else`** .. **`#endif`** Test conditionnel (similaire à l'instruction `if` du langage C)

`#` Opérateur de conversion en chaîne de caractères

`##` Opérateur de concaténation de chaînes

`#error "error message"` Génère une erreur

`#pragma` Directive spécifique au compilateur.

Le préprocesseur C est indépendant du langage C, c'est-à-dire qu'il peut être exécuté sur n'importe quel type de fichier. Prenons l'exemple d'une lettre générique d'un cabinet dentaire :

```
#ifdef FEMALE
#    define NAME Madame
#else
#    define NAME Monsieur
#endif
Bonjour NAME,
```

(suite sur la page suivante)

(suite de la page précédente)

```
Veuillez noter votre prochain rendez-vous le DATE, à HOUR heure.

Veuillez agréer, NAME, nos meilleures salutations,

#ifdef IS_BOSS
Le directeur
#elif defined IS_ASSISTANT
La secrétaire du directeur
#elif defined OWNER_NAME
OWNER_NAME
#else
#    error "Lettre sans signature"
#endif
```

Il est possible d'appeler le préprocesseur directement avec l'option `-E`. Des directives `define` peuvent être renseignées depuis la ligne de commande :

```
$ gcc -xc -E test.txt \
    -DDATE=22 -DHOUR=9:00 \
    -DFEMALE \
    -DOWNER_NAME="Adam" -DPOSITION=employee
# 1 "test.txt"
# 1 "<built-in>"
# 1 "<command-line>"
# 1 "/usr/include/stdc-predef.h" 1 3 4
# 1 "<command-line>" 2
# 1 "test.txt"
Bonjour Madame,

Veuillez noter votre prochain rendez-vous le 22, à 9:00 heure.

Veuillez agréer, Madame, nos meilleures salutations,

Adam
```

Notez que les instructions du préprocesseur (à l'exception des opérateurs de concaténation de conversion en chaîne de caractère) sont des instructions de ligne (*line-wise*), et doivent se terminer par un caractère de fin de ligne.

16.1 Phases de traduction

Le standard décrit 4 phases de pré-processing :

1. Remplacement des caractères spéciaux, décodage des trigraphes, traitement des fin de lignes.
2. Fusionne les lignes utilisant un retour virtuel `\`.
3. Supprime les commentaires, décompose les symboles du préprocesseur
4. Exécute les directives du préprocesseur (`#define` et `#include`)

16.2 Extensions des fichiers

Par convention, et selon le standard GNU, les extensions suivantes sont en vigueur :

`.h` Fichier d'en-tête ne comportant que des définitions préprocesseur, des déclarations (structures, unions, ...) et des prototypes de fonction, mais aucun code exécutable. Ce fichier sera soumis au préprocesseur.

`.c` Fichier source C comportant les implémentations de fonction et les variables globales. Ce fichier sera soumis au préprocesseur.

`.i` Fichier source C qui ne sera pas soumis au préprocesseur : `gcc -E -o foo.i foo.c`

`.s` Fichier assembleur non soumis au préprocesseur.

`.S` Fichier assembleur soumis au préprocesseur. Notons toutefois que cette convention n'est pas applicable sous Windows, car le système de fichier n'est pas sensible à la casse.

16.3 Inclusion de fichiers

16.3.1 #include

La directive include peut prendre deux formes, l'inclusion locale et l'inclusion globale. Il s'agit d'ailleurs de l'une des questions les plus posées (c.f. cette question.).

`#include <filename>` Le préprocesseur va chercher le chemin du fichier à inclure dans les chemins de l'implémentation.

`#include "filename"` Le préprocesseur cherche le chemin du fichier à partir du chemin courant et les chemins donnés par les des directives `-I`.

L'inclusion de fichier est simplement du remplacement de chaînes :

```
$ echo "Ce début de phrase est ici" > head.h
$ echo ", mais cette fin est là." > tail.h
$ echo -e '#include "head.h"\n#include "tail.h"\n' > main.c
$ gcc -E main.c -o-
```

(suite sur la page suivante)

(suite de la page précédente)

```
# 1 "main.c"
# 1 "<built-in>"
# 1 "<command-line>"
# 31 "<command-line>"
# 1 "/usr/include/stdc-predef.h" 1 3 4
# 32 "<command-line>" 2
# 1 "main.c"
# 1 "head.h" 1
Ce début de phrase est ici
# 2 "main.c" 2
# 1 "tail.h" 1
, mais cette fin est là.
# 3 "main.c" 2
```

La directive `#include` est principalement utilisée pour include des fichiers d'en-tête (*header*), mais rarement (jamais), des fichiers C.

16.4 Définitions

16.4.1 #define

Les définitions sont des symboles généralement écrits en majuscule et qui sont remplacés par le préprocesseur. Ces définitions peuvent être utiles pour définir des constantes globales qui sont définies à la compilation :

```
#ifndef WINDOW_SIZE
#    define WINDOW_SIZE 10
#endif

int tab[WINDOW_SIZE];

void init(void) {
    for(size_t i = 0; i < WINDOW_SIZE; i++)
        tab[i] = i;
}
```

Il est ainsi possible de définir la taille du tableau à la compilation avec :

```
$ gcc main.c -DWINDOW_SIZE=42
```

Notons qu'au pré-processing, toute occurrence d'un symbole défini est remplacé par le contenu de sa définition. **C'est une remplacement de chaîne bête, idiot et naïf**. Il est par conséquent possible d'écrire :

```
#define MAIN int main(
#define BEGIN ) {
```

(suite sur la page suivante)

(suite de la page précédente)

```
#define END return 0; }
#define EOF "\n"

MAIN
BEGIN
    printf("Hello" EOF);
END
```

On relèvera qu'il est aussi possible de commettre certaines erreurs :

```
#define ADD a + b

int a = 12;
int b = 23;
int c = ADD * ADD
```

Après pré-processing on aura un comportement non désiré, car la multiplication est plus prioritaire que l'addition.

```
#define ADD a + b

int a = 12;
int b = 23;
int c = a + b * a + b
```

Pour se prémunir contre ces éventuelles coquilles, on protègera toujours les définitions avec des parenthèses `#define ADD (a + b)`.

16.4.2 #undef

Un symbole défini soit par la ligne de commande `-DFOO=1`, soit par la directive `#define FOO 1` ne peut pas être redéfini. C'est pourquoi il est possible d'utiliser `#undef` pour supprimer une directive préprocesseur :

```
#ifdef FOO
#   undef FOO
#endif
#define FOO 1
```

Généralement on évitera de faire appel à `#undef` car le bon programmeur aura forcé la définition d'une directive en amont pour contraindre le développement en aval.

16.5 Macros

On appel **macro** une fonction définie au niveau du préprocesseur.

16.6 Debogage

16.6.1 #error

Cette directive génère une erreur avec le texte qui suit la directive :

```
#if !(KERNEL_SIZE % 2)
#    error Le noyau du filtre est pair
#endif
```

16.6.2 Directives spéciales

Le standard définit certains symboles utiles pour le débogage :

__LINE__ Est remplacé par le numéro de la ligne sur laquelle est placé ce symbole

__FILE__ Est remplacé par le nom du fichier sur lequel est placé ce symbole

__func__ Est remplacé par le nom de la fonction du bloc dans lequel la directive se trouve

__STDC__ Est remplacé par 1 pour indiquer que l'implémentation est compatible avec C90

__DATE__ Est remplacé par la date sous la forme `"Mmm dd yyyy"`

__TIME__ Est remplacé par l'heure au moment du pr-processing `"hh:mm:ss"`

16.7 Caractère d'échappement

L'anti-slash (`backslash`) est interprété par le préprocesseur comme un saut de ligne virtuel. Il permet par exemple de casser les longues lignes :

```
#define TRACE printf("Le programme est passé " \
    " dans le fichier %s" \
    " ligne %d\n", \
    __FILE__, __LINE__);
```

16.8 Macros

Une macro est une définition qui prend des arguments en paramètre :

```
#define MIN(x, y) ((x) < (y) ? (x) : (y))
```

De la même manière que pour les définissions simple, il s'agit d'un remplacement de chaîne :

```
$ cat test.c
#define MIN(x, y) ((x) < (y) ? (x) : (y))

int main(void) {
    return MIN(23, 12);
}

$ gcc -E test.c -o-
int main(void) {
    return ((23) < (12) ? (23) : (12));
}
```

Notez que l'absence d'espace entre le nom de la macro et la parenthèse est importante. L'exemple suivant le démontre :

```
$ cat test.c
#define ADD (x, y) ((x) + (y))

int main(void) {
    return ADD(23, 12);
}

$ gcc -E test.c -o-
int main(void) {
    return (x, y) ((x) + (y))(23, 12);
}
```

16.9 Directives conditionnelles

Les directives `#if`, `#else`, `#elif` et `#endif` sont utiles pour rendre conditionnelle une section de code. Cela peut être utilisé pour définir une structure selon le boutisme de l'architecture cible :

```
#ifdef BIG_ENDIAN
typedef struct {
    int header;
    int body;
    int tail;
```

(suite sur la page suivante)

(suite de la page précédente)

```
} Dataframe;
#else
typedef struct {
    int tail;
    int body;
    int header;
} Dataframe;
#endif
```

16.9.1 Désactivation de code

On voit souvent des développeurs commenter des sections de code pour le débogage. Cette pratique n'est pas recommandée, car les outils de refactoring (réusinage de code), ne parviendront pas à interpréter le code en commentaire jugeant qu'il ne s'agit pas de code, mais de texte insignifiant. Une méthode plus robuste et plus sure consiste à utiliser une directive conditionnelle :

```
#if 0 // TODO: Check if this code is still required.
if (x < 0) {
    x = 0;
}
#endif
```

16.9.2 Include guard

La protection des fichiers d'en-tête permet d'éviter d'inclure un fichier s'il a déjà été inclus.

Imaginons que la constante `M_PI` soit définie dans le header `<math.h>` :

```
#define M_PI        3.14159265358979323846
```

Si ce fichier d'en-tête est inclus à nouveau, le préprocesseur génèrera une erreur, car le symbole est déjà défini. Pour éviter ce genre d'erreur, les fichiers d'en-tête sont protégés par un garde :

```
#ifndef MATH_H
#define MATH_H

...

#endif
```

Si le fichier a déjà été inclus, la définition `MATH_H` sera déjà déclarée et le fichier d'en-tête ne sera pas réinclus.

On préfèrera utiliser la directive #pragma once qui est plus simple à l'usage et évite une collision de nom. Néanmoins et bien que cette directive ne soit pas standardisée par l'ISO, elle est compatible avec la très grande majorité des compilateurs C.

```
#pragma once

...
```

16.10 Commentaires

Les commentaires C du type :

```
// Blabla

/**
 * Le corbeau et le renard.
 */
```

sont aussi des directives du préprocesseur. Ils seront retirés par le préprocesseur.

Exercice

Que pensez-vous de cette définition ?

```
#define IS_OCTAL(c) ((c) >= '0' && (c) <= '8')
```

Chapitre 17

Algorithmes et conception

L'algorithmique est le domaine scientifique qui étudie les algorithmes, une suite finie et non ambiguë d'opérations ou d'instructions permettant de résoudre un problème ou de traiter des données.

Un algorithme peut être également considéré comme étant n'importe quelle séquence d'opérations pouvant être simulées par un système Turing-complet. Un système est déclaré Turing-complet s'il peut simuler n'importe quelle machine de Turing. For heureusement, le langage C est Turing-complet puisqu'il possède tous les ingrédients nécessaires à la simulation de ces machines, soit compter, comparer, lire, écrire...

Dans le cas qui concerne cet ouvrage, un algorithme est une recette exprimée en une liste d'instructions et permettant de résoudre un problème informatique. Cette recette contient à peu de choses près les éléments programmatiques que nous avons déjà entre aperçus : des structures de contrôle, des variables, etc.

Généralement un algorithme peut être exprimé graphiquement en utilisant un organigramme (*flowchart*) ou un structogramme (*Nassi-Shneiderman diagram*) afin de s'affranchir du langage de programmation cible.

La **conception** aussi appelée Architecture logicielle est l'art de penser un programme avant son implémentation. La phase de conception fait bien souvent appel à des algorithmes.

Pour être qualifié d'algorithme certaines propriété doivent être respectées :

1. **Entrées**, un algorithme doit posséder 0 ou plus d'entrées en provenance de l'extérieur de l'algorithme.
2. **Sorties**, un algorithme doit posséder au moins une sortie.
3. **Rigueur**, chaque étape d'un algorithme doit être claire et bien définie.
4. **Finitude**, un algorithme doit comporté un nombre fini d'étapes.
5. **Répétable**, un algorithme doit fournir un résultat répétable.

17.1 Complexité d'un algorithme

Il est souvent utile de savoir quelle est la complexité d'un algorithme afin de le comparer à un autre algorithme équivalent. Il existe deux indicateurs :

— La complexité en temps
— La complexité en mémoire

Pour l'un, l'idée est de savoir combine de temps CPU consomme un algorithme. Pour l'autre, on s'intéresse à l'utilisation de mémoire tampon.

La complexité en temps et en mémoire d'un algorithme est souvent exprimée en utilisant la notation en O (*big O notation*). Par exemple, la complexité en temps d'un algorithme qui demanderait 10 étapes pour être résolu s'écrirait :

$$O(10)$$

Un algorithme qui ferait une recherche dichotomique sur un tableau de n éléments à une complexité $O(log(n))$.

Quelques points à retenir :

— La complexité d'un algorithme considère toujours le cas le moins favorable.
— Le meilleur algorithme est celui qui présente le meilleur compromis entre sa complexité en temps et sa complexité en mémoire.

À titre d'exemple, le programme suivant qui se charge de remplacer les valeurs paires d'un tableau par un 0 et les valeurs impaires par un 1 à une complexité en temps de $O(n)$ où n est le nombre d'éléments du tableau.

```
void discriminate(int* array, size_t length)
{
    for (size_t i = 0; i < length; i++)
    {
        array[i] %= 2;
    }
}
```

D'une manière générale, la plupart des algorithmes que l'ingénieur écrira appartiendront à ces catégories exprimées du meilleur au plus mauvais :

Complexité	$n = 100000$	i7 (100'000 MIPS)
$O(log(n))$	11	0.11 ns
$O(n)$	100'000	1 us
$O(nlog(n))$	1'100'000	11 us
$O(n^2)$	10'000'000'000	100 ms (un battement de cil)
$O(2^n)$	très très grand	Le soleil devenu géante rouge aura ingurgité la terre
$O(n!)$	trop trop grand	La galaxie ne sera plus que poussière

Les différentes complexité peuvent être résumées sur la figure suivante :

Un algorithme en $O(n^2)$, doit éveiller chez le développeur la volonté de voir s'il n'y a pas moyen d'optimiser l'algorithme en réduisant sa complexité, souvent on s'aperçoit

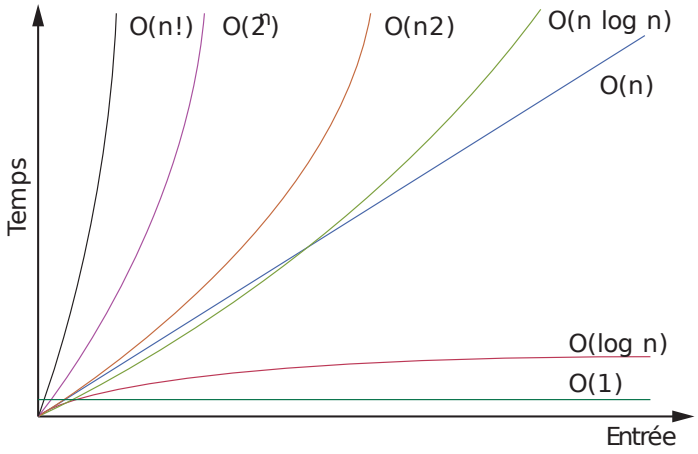


Fig. 17.1 – Différentes complexités d'algorithmes

qu'un algorithme peut être optimisé et s'intéresser à sa complexité est un excellent point d'entrée.

Attention toutefois à ne pas mal évaluer la complexité d'un algorithme. Voyons par exemple les deux algorithmes suivants :

```c
int min = MAX_INT;
int max = MIN_INT;

for (size_t i = 0; i < sizeof(array) / sizeof(array[0]); i++) {
    if (array[i] < min) {
        min = array[i];
    }
    if (array[i] > min) {
        max = array[i];
    }
}
```

```c
int min = MAX_INT;
int max = MIN_INT;

for (size_t i = 0; i < sizeof(array) / sizeof(array[0]); i++)
{
    if (array[i] < min) {
        min = array[i];
    }
}

for (size_t i = 0; i < sizeof(array) / sizeof(array[0]); i++)
{
    if (array[i] > min) {
        max = array[i];
    }
}
```

Exercice

Quel serait l'algorithme permettant d'afficher :

et dont la taille peut varier ? Exercice

On vous donne un gros fichier de 3'000'000'000 entiers positifs 32-bits, il vous faut générer un entier qui n'est pas dans la liste. Le hic, c'est que vous n'avez que 500 MiB de mémoire de travail. Quel algorithme proposez-vous ?

Une fois le travail terminé, votre manager vient vous voir pour vous annoncer que le cahier des charges a été modifié. Le client dit qu'il n'a que 10 MiB. Pensez-vous pouvoir résoudre le problème quand même ?

17.2 Machines d'états

17.3 Diagrammes visuels

— Diagrammes en flux
— Structogrammes
— Diagramme d'activités

17.4 Récursivité

La récursivité est une autoréférence. Il peut s'agit en C d'une fonction qui s'appelle elle-même. Exercice

Soit deux tableaux d'entiers, trouver la paire de valeurs (une dans chaque tableau) ayant la plus petite différence (positive).

Exemple :

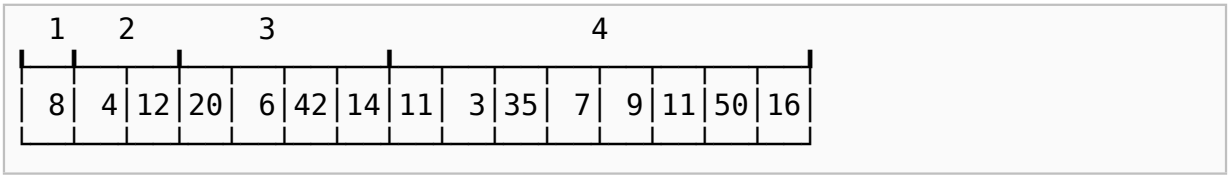
```
int a[] = {5, 3, 14, 11, 2};
int b[] = {24, 128, 236, 20, 8};

int diff = 3 // pair 11, 8
```

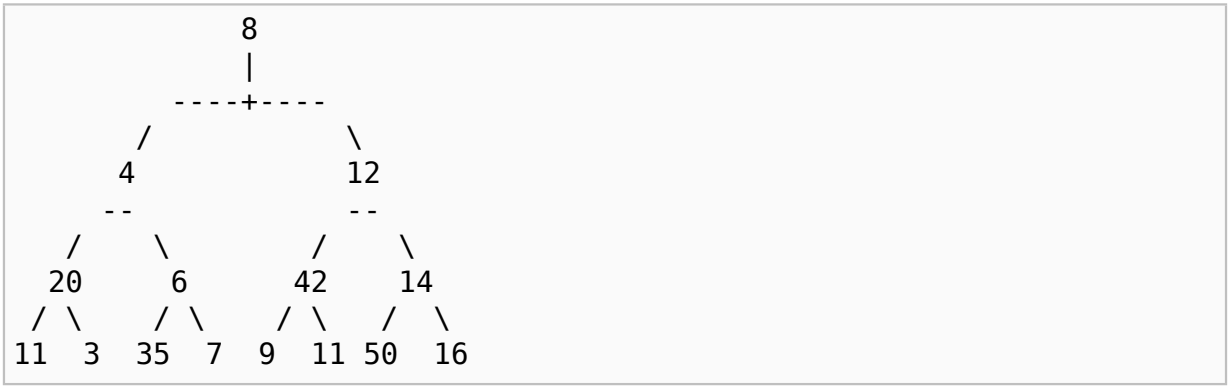
1. Proposer une implémentation
2. Quelle est la complexité de votre algorithme ?

17.5 Programmation dynamique

17.6 Algorithmes de tris

17.6.1 Heap Sort





17.7 Type d'algorithmes

17.7.1 Algorithmes en ligne (incrémental)

Un algorithme incrémental ou online est un algorithme qui peut s'exécuter sur un flux de données continu en entrée. C'est à dire qu'il est en mesure de prendre des décisions sans avoir besoin d'une visibilité complète sur le set de données.

Un exemple typique est le problème de la secrétaire. On souhaite recruter une nouvelle secrétaire et le recruteur voit défiler les candidats. Il doit décider à chaque entretient s'il engage ou non le candidat et ne peut pas attendre la fin du processus d'entretiens pour obtenir le score attribué à chaque candidat. Il ne peut comparer la performance de l'un qu'à celle de deux déjà entrevus. L'objectif est de trouver la meilleure stratégie.

La solution à ce problème est de laisser passer 37% des candidats sans les engager. Ceci correspond à une proportion de $1/e$. Ensuite il suffit d'attendre un ou une candidate meilleure que toutes ceux/celles du premier échantillon.

Exercice

L'intégrateur de Kahan (Kahan summation algorithm) est une solution élégante pour palier à la limite de résolution des types de données.

L'algorithme pseudo-code peut être exprimé comme :

```
function kahan_sum(input)
    var sum = 0.0
    var c = 0.0
    for i = 1 to input.length do
        var y = input[i] - c
        var t = sum + y
        c = (t - sum) - y
        sum = t
    next i
    return sum
```

1. **Implémenter cet algorithme en C compte tenu du prototype :**

```c
float kahan_sum(float value, float sum, float c);
```

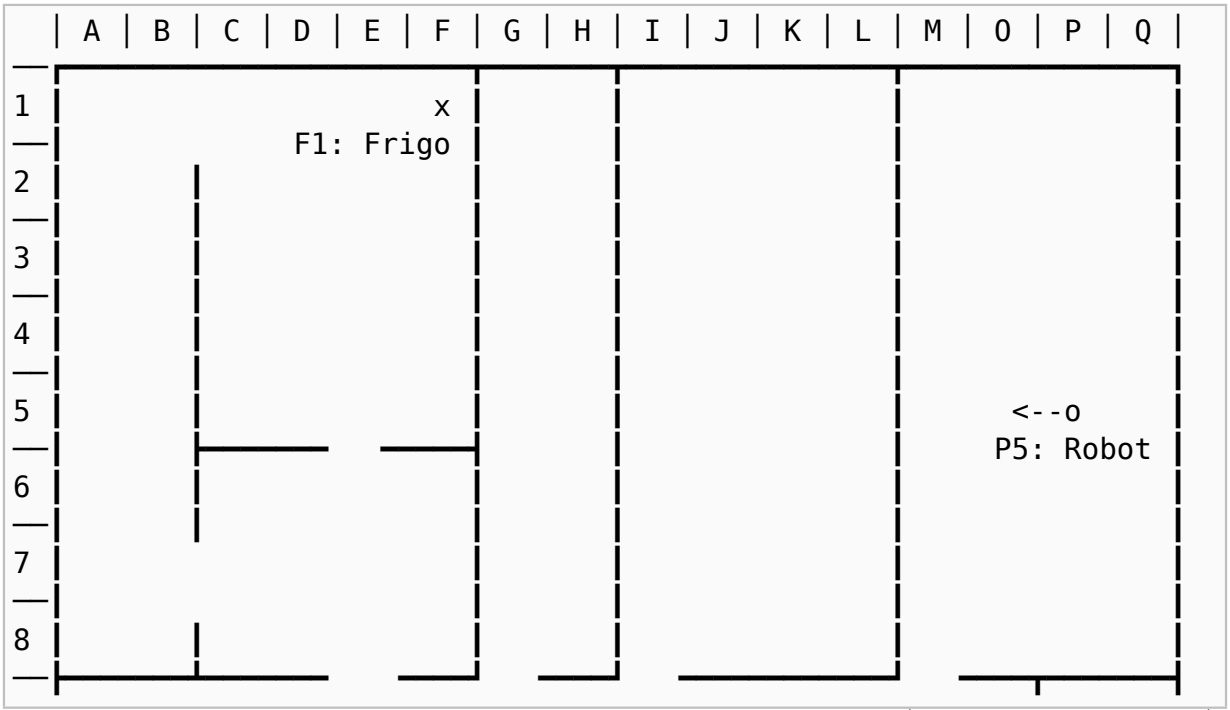
2. Expliquer comment fonctionne cet algorithme.
3. Donner un exemple montrant l'avantage de cet algorithme sur une simple somme.

Exercice

Un robot aspirateur souhaite se rassasier et cherche le frigo, le problème c'est qu'il ne sait pas où il est. Elle serait la stratégie de recherche du robot pour se rendre à la cuisine ?

Le robot dispose de plusieurs fonctionnalités :
- Avancer
- Tourner à droite de 90°
- Détection de sa position absolue p.ex. P5

Élaborer un algorithme de recherche.



(suite sur la page suivante)

(suite de la page précédente)

Chapitre 18

Compilation séparée

18.1 Translation unit

En programmation, on appelle *translation unit* (unité de traduction), un code qui peut être **compilé** en un **objet** sans autre dépendance externe. Le plus souvent, une unité de traduction correspond à un fichier C.

18.2 Diviser pour mieux régner

De même qu'un magasine illustré est divisé en sections pour accroître la lisibilité (sport, news, annonces, météo) de même un code source est organisé en éléments fonctionnels le plus souvent séparés en plusieurs fichiers et ces derniers parfois maintenus par différents développeurs.

Rappelons-le (et c'est très important) :
- une fonction ne devrait pas dépasser un écran de haut (~50 lignes) ;
- un fichier ne devrait pas dépasser 1000 lignes ;
- une ligne ne devrait pas dépasser 80 caractères.

Donc à un moment, il est essentiel de divisier son travail en créant plusieurs fichiers.

Ainsi, lorsque le programme commence à être volumineux, sa lecture, sa compréhension et sa mise au point deviennent délicates même pour le plus aguéri des développeur. Il est alors essentiel de scinder le code source en plusieurs fichiers. Prenons l'exemple d'un programme qui effectue des calculs sur les nombres complexes. Notre projet est donc constitué de trois fichiers :

```
$ tree
.
├── complex.c
├── complex.h
└── main.c
```

Le programme principal et la fonction `main` est contenu dans `main.c` quant au module *complex* il est composé de deux fichiers : `complex.h` l'en-tête et `complex.c`, l'implémentation du module.

Le fichier `main.c` devra inclure le fichier `complex.h` afin de pourvoir utiliser correctement les fonctions du module de gestion des nombres complexes. Exemple :

```c
// fichier main.c
#include "complex.h"

int main() {
    Complex c1 = { .real = 1., .imag = -3. };
    complex_fprint(stdout, c1);
}
```

```c
// fichier complex.h
#ifndef COMPLEX_H
#define COMPLEX_H

#include <stdio.h>

typedef struct Complex {
    double real;
    double imag;
} Complex, *pComplex;

void complex_fprint(FILE *fp, const Complex c);

#endif // COMPLEX_H
```

```c
// fichier complex.c
#include "complex.h"

void complex_fprint(FILE* fp, const Complex c) {
    fprintf(fp, "%+.3lf + %+.3lf\n", c.real, c.imag);
}
```

Un des avantages majeurs à la création de modules est qu'un module logiciel peut être réutilisé pour d'autres applications. Plus besoin de réinventer la roue à chaque application !

Cet exemple sera compilé dans un environnement POSIX de la facon suivante :

```
gcc -c complex.c -o complex.o
gcc -c main.c -o main.o
gcc complex.o main.o -oprogram -lm
```

Nous verrons plus bas les éléments théoriques vous permettant de mieux comprendre ces lignes.

18.3 Module logiciel

Les applications modernes dépendent souvent de nombreux modules logiciels externes aussi utilisés dans d'autres projets. C'est avantageux à plus d'un titre :

— les modules externes sont sous la responsabilité d'autres développeurs et le programme a développer comporte moins de code ;
— les modules externes sont souvent bien documentés et testés et il est facile de les utiliser ;
— la lisibilité du programme est accrue car il est bien découpé en des ensembles fonctionnels ;
— les modules externes sont réutilisables et indépendants, ils peuvent donc être réutilisés sur plusieurs projets.

Lorsque vous utiliser la fonction `printf`, vous dépendez d'un module externe nommé `stdio`. En réalité l'ensemble des modules `stdio`, `stdlib`, `stdint`, `ctype`... sont tous groupé dans une seule bibliothèque logicielle nommée `libc` disponible sur tous les systèmes compatibles POSIX. Sous Linux, le pendant libre `glibc` est utilisée. Il s'agit de la biblothèque GNU C Library.

Un module logiciel peut se composer de fichiers sources, c'est à dire un ensemble de fichiers `.c` et `.h` ainsi qu'une documentation et un script de compilation (`Makefile`). Alternativement, un module logiciel peut se composer de bibliothèques déjà compilées sous la forme de fichiers `.h`, `.a` et `.so`. Sous Windows on rencontre fréquemment l'extension `.dll`. Ces fichiers compilés ne donnent pas accès au code source mais permettent d'utiliser les fonctionnalités quelles offrent dans des programmes C en mettant à disposition un ensemble de fonctions documentées.

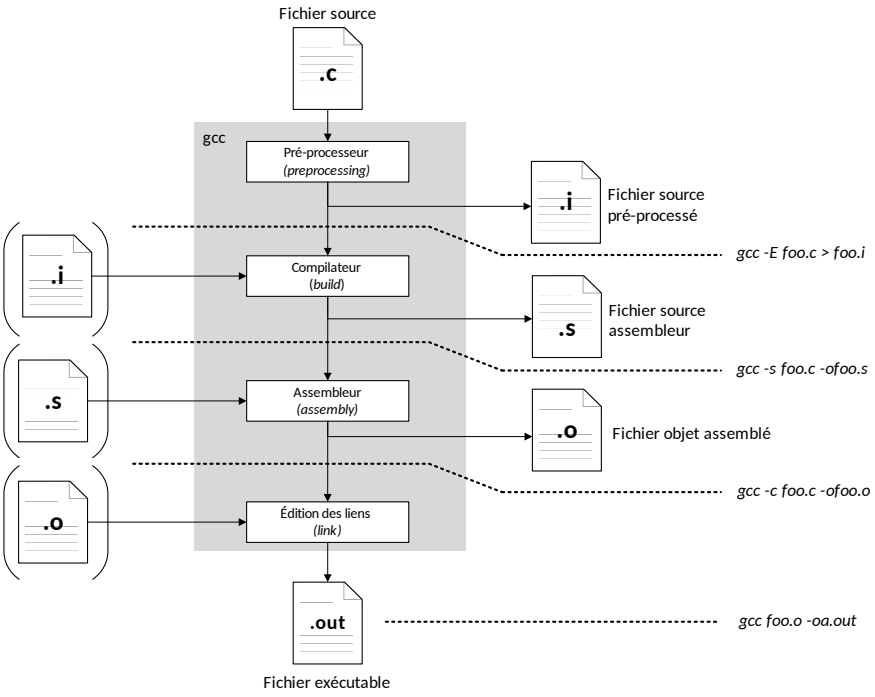
18.4 Compilation avec assemblage différé

Lorsque nous avions compilé notre premier exemple Hello World nous avions simplement appelé `gcc` avec le fichier source `hello.c` qui nous avait créé un exécutable `a.out`. En réalité, GCC est passé par plusieurs sous étapes de compilation :

1. **Préprocessing** : les commentaires sont retirés, les directives pré-processeur sont remplacées par leur équivalent C.
2. **Compilation** : le code C d'une seule *translation unit* est converti en langage machine en un fichier objet `.o`.
3. **Édition des liens** : aussi nommé *link*, les différents fichiers objets sont réunis en un seul exécutable.

Lorsqu'un seul fichier est fourni à GCC, les trois opérations sont effectuées en même temps mais ce n'est plus possible aussitôt que le programme est composé de plusieurs unités de translation (plusieurs fichiers C). Il est alors nécessaire de compiler manuellement chaque fichier source et d'en créer.

La figure suivante résume les différentes étapes de GCC. Les pointillés indiquent à quel niveau les opérations peuvent s'arrêter. Il est dès lors possible de passer par des fichiers intermédiaires assembleur (`.s`) ou objets (`.o`) en utilisant la bonne commande.



Notons que ces étapes existent quelque soit le compilateur ou le système d'exploitation. Nous retrouverons ces exactes mêmes étapes avec Microsoft Visual Studio mais le nom des commandes et les extensions des fichiers peuvent varier s'ils ne respectent pas la norme POSIX (et GNU).

Notons que généralement, seul deux étapes de GCC sont utilisées :

1. Compilation avec `gcc -c <fichier.c>`, ceci génère automatiquement un fichier `.o` du même nom que le fichier d'entrée.
2. Édition des liens avec `gcc <fichier1.o> <fichier2.o> ...`, ceci génère automatiquement un fichier exécutable `a.out`.

18.5 Fichiers d'en-tête (*header*)

Les fichiers d'en-tête (`.h`) sont des fichiers écrits en langage C mais qui ne contienne pas d'implémentation de fonctions. Un tel fichier ne contient donc pas de `while`, de `for` ou même de `if`. Par convention ces fichiers ne contienne que :

— Des prototypes de fonctions (ou de variables).
— Des déclaration de types (`typedef`, `struct`).
— Des définitions pré-processeur (`#include`, `#define`).

Nous l'avons vu dans le chapitre sur le pré-processeur, la directive `#include` ne fait qu'inclure le contenu du fichier cible à l'emplacement de la directive. Il est donc possible (mais fort déconseillé), d'avoir la situation suivante :

```c
// main.c
int main() {
   #include "foobar.def"
}
```

Et le fichier `foobar.def` pourrait cotenir :

```c
// foobar.def
#ifdef FOO
printf("hello foo!\n");
#else
printf("hello bar!\n");
#endif
```

Vous noterez que l'extension de `foobar` n'est pas `.h` puisque le contenu n'est pas un fichier d'en-tête. `.def` ou n'importe quelle autre extension pourrait donc faire l'affaire ici.

Dans cet exemple, le pré-processeur ne fait qu'inclure le contenu du fichier `foobar.def` à l'emplacement de la définition `#include "foobar.def"`. Voyons le en détail :

```
$ cat << EOF > main.c
→ int main() {
→     #include "foobar.def"
→     #include "foobar.def"
→ }
→ EOF

$ cat << EOF > foobar.def
→ #ifdef FOO
→ printf("hello foo!\n");
→ #else
→ printf("hello bar!\n");
→ #endif
→ EOF

$ gcc -E main.c | sed '/^#/ d'
int main() {
printf("hello bar\n");
printf("hello bar\n");
}
```

Lorsque l'on observe le résultat du pré-processeur, on s'aperçois que toutes les directives préprocesseur ont disparues et que la directive `#include` a été remplacée par de contenu de `foobar.def`. Remarquons que le fichier est inclus deux fois, nous verrons plus loin comme éviter cela.

Nous avons vu au chapitre sur les prototypes de fonctions qu'il est possible de ne déclarer que la première ligne d'une fonction. Ce prototype permet au compilateur de savoir combien d'arguments est composé une fonction sans nécessairement disposer de l'implémentation de cette fonction. Aussi on trouve dans tous les fichiers d'en-tête des déclaration

en amont (*forward declaration*). Dans le fichier d'en-tête `stdio.h` on trouvera la ligne : `int printf( const char *restrict format, ... );`.

Notons qu'ici le prototype est précédé par le mot clé `extern`. Il s'agit d'un mot clé **optionnel** permettant de renforcer l'intention du développeur que la fonction déclarée n'est pas inclue dans fichier courant mais qu'elle est implémentée ailleurs, dans un autre fichier. Et c'est le cas car `printf` est déjà compilée quelque part dans la bibliothèque `libc` inclue par défaut lorsqu'un programme C est compilé dans un environnement POSIX.

Un fichier d'en-tête contiendra donc tout le nécessaire utile à pouvoir utiliser une bibliothèque externe.

18.5.1 Protection de réentrance

La protection de réentrence aussi nommée *header guards* est une solution au problème d'inclusion multiple. Si par exemple on défini dans un fichier d'en-tête un nouveau type et que l'on inclus ce fichier, mais que ce dernier est déjà inclu par une autre bibliothèque une erreur de compilation apparaîtera :

```
$ cat << EOF > main.c
→ #include "foo.h"
→ #include "bar.h"
→ int main() {
→    Bar bar = {0};
→    foo(bar);
→ }
→ EOF

$ cat << EOF > foo.h
→ #include "bar.h"
→
→ extern void foo(Bar);
→ EOF

$ cat << EOF > bar.h
→ typedef struct Bar {
→    int b, a, r;
→ } Bar;
→ EOF

$ gcc main.c
In file included from main.c:2:0 :
bar.h:1:16: error: redefinition of ‘struct Bar’
typedef struct Bar {
               ^~~
In file included from foo.h:1:0,
                 from main.c:1 :
bar.h:1:16: note: originally defined here
```

(suite sur la page suivante)

(suite de la page précédente)

```
typedef struct Bar {
                ^~~
In file included from main.c:2:0 :
bar.h:3:3: error: conflicting types for ‘Bar’
} Bar;
^~~
...
```

Dans cet exemple l'utilisateur ne sait pas forcément que `bar.h` est déjà inclus avec `foo.h` et le résultat après pré-processing est le suivant :

```
$ gcc -E main.c | sed '/^#/ d'
typedef struct Bar {
int b, a, r;
} Bar;

extern void foo(Bar);
typedef struct Bar {
int b, a, r;
} Bar;
int main() {
Bar bar = {0};
foo(bar);
}
```

On y retrouve la définition de `Bar` deux fois et donc, le compilateur génère une erreur.

Une solution à ce problème est d'ajouter des gardes d'inclusion multiple par exemple avec ceci :

```
#ifndef BAR_H
#define BAR_H

typedef struct Bar {
int b, a, r;
} Bar;

#endif // BAR_H
```

Si aucune définition du type `#define BAR_H` n'existe, alors le fichier `bar.h` n'a jamais été inclus auparavant et le contenu de la directive `#ifndef BAR_H` dans lequel on commence par définir `BAR_H` est exécuté. Lors d'une future inclusion de `bar.h`, la valeur de `BAR_H` aura déjà été définie et le contenu de la directive `#ifndef BAR_H` ne sera jamais exécuté.

Alternativement, il existe une solution **non standard** mais supportée par la plupart des compilateurs. Elle fait intervenir un pragma :

```
#pragma once
```

(suite sur la page suivante)

(suite de la page précédente)

```
typedef struct Bar {
int b, a, r;
} Bar;
```

Cette solution est équivalente à la méthode traditionnelle et présente plusieurs avantages. C'est tout d'abord une solution atomique qui ne nécessite pas un `#endif` à la fin du fichier. Il n'y a ensuite pas de conflit avec la règle SSOT car le nom du fichier `bar.h` n'apparaît pas dans le fichier `BAR_H`.

Chapitre 19

Portée et visibilité

Ce chapitre se concentre sur quatre caractéristiques d'une variable :
- La portée
- La visibilité
- La durée de vie
- Son qualificatif de type

Dans les quatre cas, elles décrivent l'accessibilité, c'est à dire jusqu'à ou/et jusqu'à quand une variable est accessible, et de quelle manière

19.1 Espace de nommage

L'espace de nommage ou `namespace` est un concept différent de celui existant dans d'autres langages tel que C++. Le standard **C99** décrit 4 types possibles pour un identifiant :
- fonction et *labels*
- noms de structures (`struct`), d'unions (`union`), d'énumération (`enum`),
- identifiants

19.2 Portée

La portée ou scope décrit jusqu'à où une variable est accessible.

Une variable est **globale**, c'est-à-dire accessible partout, si elle est déclarée en dehors d'une fonction :

```
int global_variable = 23;
```

Une variable est **locale** si elle est déclarée à l'intérieur d'un bloc, ou à l'intérieur d'une fonction. Elle sera ainsi visible de sa déclaration jusqu'à la fin du bloc courant :

Fig. 19.1 – Brouillard matinal sur le Golden Gate Bridge, San Francisco.

```c
int main(int)
{
    {
        int i = 12;

        i += 2; // Valide
    }

    i++; // Invalide, `i` n'est plus visible.
}
```

19.2.1 Variable shadowing

On dit qu'une variable est *shadowed* ou *masquée* si sa déclaration masque une variable préalablement déclarée :

```c
int i = 23;

for(size_t i = 0; i < 10; i++) {
    printf("%ld", i); // Accès à `i` courant et non à `i = 23`
}

printf("%d", i); // Accès à `i = 23`
```

19.3 Visibilité

Selon l'endoit où est déclaré une variable, elle ne sera pas nécessairement visible partout ailleurs. Une variable **locale** n'est accessible qu'à partir de sa déclaration et jusqu'à la fin du bloc dans laquelle elle est déclarée.

L'exemple suivant montre la visibilité de plusieurs variables :

```c
                    //   a
void foo(int a) {   //   ┬ b
    int b;          //   │ ┬
    ...             //   │ │
    {               //   │ │ c
       int c;       //   │ │ ┬
       ...          //   │ │ │ d
       int d;       //   │ │ │ ┬
       ...          //   │ │ │ │
    }               //   │ │ ┴ ┴
    ...             //   │ │
}                   //   ┴ ┴
```

Une variable déclarée globalement, c'est à dire en dehors d'une fonction à une durée de vie sur l'entier du module (*translation unit*) quelque soit l'endroit ou elle est déclarée, en revanche elle n'est visible que depuis l'endroit ou elle est déclarée. Les deux variables `i` et `j` sont globales au module, c'est à dire qu'elle peuvent être accédées depuis n'importe quelle fonction contenu dans ce module.

En revanche la variable `j`, bien qu'elle ait ait une durée de vie sur toute l'exécution du programme et que sa portée est globale, elle ne pourra être accédée depuis `main` car elle n'est pas visible.

```
#include <stdio.h>
                              //  i
int i;                        //  ┬
                              //  │
int main() {                  //  │
    printf("%d %d\n", i, j); //  │
}                             //  │
                              //  │ j
int j;                        //  ┴ ┬
```

Le mot clé `extern` permet non pas de déclarer la variable `j` mais de renseigner le compilateur qu'il existe *ailleurs* une variable `j`. C'est ce que l'on appelle une déclaration avancée ou *forward-declaration*. Dans ce cas, bien que `j` soit déclaré après la fonction principale, elle est maintenant visible.

```
#include <stdio.h>

                          // j
extern int j;             // ┬   Déclaration en amont de `j`
                          // │
int main() {              // │
    printf("%d\n", j);    // │
}                         // │
                          // │
int j;                    // │
                          // │
```

Une particularité en C est que tout symbol global (variable ou fonction) ont une accessibilité transversale. C'est à dire que dans le cas de la compilation séparée, une variable déclarée dans un fichier, peut être accédée depuis un autre fichier, il en va de même pour les fonctions.

L'exemple suivant implique deux fichiers `foo.c` et `main.c`. Dans l'un deux symbols sont déclarés, une variable et une fonction.

```
// foo.c

int foo;

void do_foo() {
```

(suite sur la page suivante)

(suite de la page précédente)

```
    printf("Foo does...");
}
```

Depuis le programme principal, il est possible d'accéder à symboles à condition de renseigner sur le prototype de la fonction et l'existence de la variable :

```
// main.c

extern int foo;
extern void do_foo(); // Non obligatoire

int main() {
    foo();
}
```

Dans le cas ou l'on voudrait restreindre l'accessibilité d'une variable au module dans lequel elle est déclarée, l'usage du mot clé `static` s'impose.

En écrivant `static int foo;` dans `foo.c`, la variable n'est plus accessible en dehors du module même avec une déclaration en avance. On dit que sa portée est réduite au module.

19.4 Qualificatif de type

Les variables en C peuvent être crées de différentes manières. Selon la manière dont elle pourront être utilisées, il est courant de les classer en catégories.

Une classe de stockage peut être implicite à une déclaration de variable ou explicite, en ajoutant un attribut devant la déclaration de celle-ci.

19.4.1 `auto`

Cette classe est utilisée par défaut lorsqu'aucune autre classe n'est précisée. Les variables automatiques sont visibles uniquement dans le bloc où elles sont déclarées. Ces variables sont habituellement créées sur la pile (*stack*) mais peuvent être aussi stockées dans les registres du processeur. C'est un choix qui incombe au compilateur.

```
auto type identificateur = valeur_initiale;
```

Pour les variables automatiques, le mot-clé *auto* n'est pas obligatoire, et n'est pas recommandé en **C99** car son utilisation est implicite.

19.4.2 `register`

Ce mot clé incite le compilateur à utiliser un registre processeur pour stocker la variable. Ceci permet de gagner en temps d'exécution car la variable n'a pas besoin d'être chargée depuis et écrite vers la mémoire.

Jadis, ce mot clé était utilisé devant toutes les variables d'itérations de boucles. La traditionnelle variable `i` utilisée dans les boucles `for` était déclarées `register int i = 0;`. Les compilateurs modernes savent aujourd'hui identifier les variables les plus souvent utilisées. L'usage de ce mot clé n'est donc plus recommandé depuis **C99**.

19.4.3 `const`

Ce mot clé rends une déclaration non modifiable par le programme lui même. Néanmoins il ne s'agit pas de constantes au sens strict du terme car une variable de type `const` pourrait très bien être modifiée par erreur en jardinant la mémoire. Quand ce mot clé est appliqué à une structure, aucun des champs de la structure n'est accessible en écriture. Bien qu'il puisse paraître étrange de vouloir rendre « constante » une « variable », ce mot clé a une utilité. En particulier, il permet de faire du code plus sûr.

19.4.4 `static`

Elle permet de déclarer des variables dont le contenu est préservé même lorsque l'on sort du bloc où elles ont été déclarées.

Elles ne sont donc initialisées qu'une seule fois. L'exemple suivant est une fonction qui retourne à chaque fois une valeur différente, incrémentée de 1. La variable `i` agit ici comme une variable globale, elle n'est initialisée qu'une seule fois à 0 et donc s'incrémente d'appel en appel. En revanche, elle n'est pas accessible en dehors de la fonction ; c'est donc une variable locale.

```c
int iterate() {
    static int i = 0;
    return i++;
}
```

Il n'est pas rare de voir des variables globales, ou des fonctions précédées du mot clé `static`. Ces variables sont dites *statiques au module*. Elle ne sont donc pas accessibles depuis un autre module (*translation unit*)

La fonction suivante est *statique* au module dans lequel elle est déclarée. Il ne sera donc pas possible d'y accéder depuis un autre fichier C.

```c
static int add(int a, int b) { return a + b; }
```

19.4.5 `volatile`

Cette classe de stockage indique au compilateur qu'il ne peut faire aucune hypothèse d'optimisation concernant cette variable. Elle indique que son contenu peut être modifié en tout temps en arrière plan par le système d'exploitation ou le matériel. Ce mot clé est davantage utilisé en programmation système, ou sur microcontrôleurs.

L'usage de cette classe de stockage réduit les performances d'un programme puisque qu'elle empêche l'optimisation du code et le contenu de cette variable devra être rechargé à chaque utilisation

19.4.6 `extern`

Cette classe est utilisée pour signaler que la variable ou la fonction associée est déclarée dans un autre module (autre fichier). Ainsi le code suivant ne déclare pas une nouvelle variable `foo` mais s'attends à ce que cette variable ait été déclarée dans un autre fichier.

```
extern int foo;
```

19.4.7 `restrict`

En C, le mot clé `restrict`, apparu avec **C99**, es utililsé uniquement pour des pointeurs. Ce qualificatif de type informe le compilateur que pour toute la durée de vie du pointeur, aucun autre pointeur ne pointera que sur la valeur qu'il pointe ou une valeur dérivée de lui même (p. ex : `p + 1`).

En d'autres termes, le qualificatif indique au compilateur que deux pointeurs différents ne peuvent pas pointer sur les même régions mémoire.

Prenons l'exemple simple d'une fonction qui met à jour deux pointeurs avec une valeur passée en paramètre :

```
void update_ptr(size_t *a, size_t *b, const size_t *value) {
    *a += *value;
    *b += *value;
}
```

Le compilateur, n'ayant aucune information sur les pointeurs fournis, ne peut faire aucune hypothèse d'optimisation. En effet, ces deux pointeurs `a` et `b` ainsi que `value` pourraient très bien pointer sur la même région mémoire, et dans ce cas `*a += *value` aurait pour effet d'incrémenter `value`. En revanche, dans le cas ou la fonction est déclarée de la façon suivante :

```
void update_ptr(size_t *restrict a, size_t * restrict b, const size_
↪t *restrict value) {
    *a += *value;
    *b += *value;
}
```

le compilateur est informé qu'il peut faire l'hypothèse que les trois pointeurs fournis en paramètres sont indépendant les uns des autres. Dans ce cas il peut optimiser le code. Voir restrict sur Wikipedia pour plus de détails.

Chapitre 20

Qualité et Testabilité

Surveiller et assurer la qualité d'un code est primordial dans toute institution et quelque soit le produit. Dans l'industrie automobile par exemple, un bogue qui serait découvert plusieurs années après la commercialisation d'un modèle d'automobile aurait des conséquences catastrophique.

Voici quelques exemples célèbres de ratés logiciels :

La sonde martienne Mariner En 1962, un bogue logiciel à causé l'échec de la mission avec la destruction de la fusée apès qu'elle ait divergé de sa trajectoire. Une formule a mal été retranscrite depuis le papier en code exécutable. Des tests suffisants auraient évité cet échec

Un pipeline sovietique de gaz En 1982, un bug a été introduit dans un ordinateur canadien acheté pour le control d'un pipeline de gas trans-sibérien. L'erreur est reportée comme la plus large explosion jamais enregistrée d'origine non nucléaire.

Le générateur de nombre pseudo-aléatoire Kerberos Kerberos est un système de sécurité utilisé par Microsoft pour chiffer les mot de passes des comptes Windows. Une erreur de code lors de la génération d'une graine aléatoire a permi de façon triviale pendant 8 ans de pénétrer n'importe quel ordinateur utilisant une authentification Kerberos.

La division entière sur Pentium En 1993, une erreur sur le silicium des processeurs Pentium, fleuron technologique de l'époque, menait à des erreurs de calcul en virgule flottante. Par exemple la division 4195835.0/3145727.0 menait à 1.33374 au lieu de 1.33382

20.1 Hacking

20.1.1 Buffer overflow

L'attaque par buffer overflow est un type d'attaque typique permettant de modifier le comportement d'un programme en exploitant "le jardinage mémoire". Lorsqu'un programme a mal été conçu et que les tests de dépassement n'ont pas été correctements implémentés, il est souvent possible d'accéder à des comportements de programmes imprévus.

Considérons le programme suivant :

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

int check_password(char *str) {
    if(strcmp(str, "starwars"))
    {
        printf ("Wrong Password \n");
        return 0;
    }

    printf ("Correct Password \n");
    return 1;
}

int main(void)
{
    char buffer[15];
    int is_authorized = 0;

    printf("Password: ");
    gets(buffer);
    is_authorized = check_password(buffer);

    if(is_authorized)
    {
        printf ("Now, you have the root access! \n");
    }
}
```

A priori, c'est un programme tout à fait correct. Si l'utilisateur entre le bon mot de passe, il se voit octroyé des privilèges administrateurs. Testons ce programme :

```
$ gcc u.c -fno-stack-protector
$ ./a.out
Password: starwars
Correct Password
Now, you have the root access!
```

Très bien, maintenant testons avec un mauvais mot de passe :

```
$ ./a.out
Password: startrek
Wrong Password
```

Et maintenant essayons avec un mot de passe magique...

20.2 Tests unitaires

20.3 Tests fonctionnels

20.4 Framework de tests

Unity

Chapitre 21

Structures de données

21.1 Types de données abstraits

Un type de donnée abstrait (**ADT** pour Abstract Data Type) cache généralement une structure dont le contenu n'est pas connu de l'utilisateur final. Ceci est rendu possible par le standard (C99 §6.2.5) par l'usage de types incomplets.

Pour mémoire, un type incomplet décrit un objet dont on ne connaît pas sa taille en mémoire.

L'exemple suivant déclare un nouveau type structure qui n'est alors pas (encore) connu dans le fichier courant :

```
typedef struct Unknown *Known;

int main() {
    Known foo; // Autorisé, le type est incomplet

    foo + 1; // Impossible car la taille de foo est inconnue.
    foo->key; // Impossible car le type est incomplet.
}
```

De façon générale, les types abstraits sont utilisés dans l'écriture de bibliothèques logicielles lorsqu'il est important que l'utilisateur final ne puisse pas compromettre le contenu du type et en forcant cet utilisateur à ne passer que par des fonctions d'accès.

Prenons le cas du fichier *foobar.c* lequel décrit une structure `struct Foo` et un type `Foo`. Notez que le type peut être déclaré avant la structure. `Foo` restera abstrait jusqu'à la déclaration complète de la structure `struct Foo` permettant de connaître sa taille. Ce fichier contient également trois fonctions :

- `init` permet d'initialiser la structure ;
- `get` permet de récupérer la valeur contenue dans `Foo` ;
- `set` permet d'assigner une valeur à `Foo`.

En plus, il existe un compteur d'accès `count` qui s'incrémente lorsque l'on assigne une valeur et se décrémente lorsque l'on récupère une valeur.

```c
#include <stdlib.h>

typedef struct Foo Foo;

struct Foo {
    int value;
    int count;
};

void init(Foo** foo) {
    *foo = malloc(sizeof(Foo)); // Allocation dynamique
    (*foo)->count = (*foo)->value = 0;
}

int get(Foo* foo) {
    foo->count--;
    return foo->value;
}

void set(Foo* foo, int value) {
    foo->count++;
    foo->value = value;
}
```

Evidemment, on ne souhaite pas qu'un petit malin compromette ce compteur en écrivant maladroitement :

```c
foo->count = 42; // Hacked this !
```

Pour s'en protéger on a recours à la compilation séparée (voir chapitre **TranslationUnits**___) dans laquelle le programme est découpé en plusieurs fichiers. Le fichier `foobar.h` contiendra tout ce qui doit être connu du programme principal, à savoir les prototypes des fonctions, et le type abstrait :

```c
#pragma once

typedef struct Foo Foo;

void init(Foo** foo);
int get(Foo* foo);
void set(Foo* foo, int value);
```

Ce fichier sera inclu dans le programme principal `main.c` :

```c
#include "foobar.h"
#include <stdio.h>

int main() {
    Foo *foo;
```

(suite sur la page suivante)

(suite de la page précédente)

```
    init(&foo);
    set(foo, 23);
    printf("%d\n", get(foo));
}
```

En résumé, un type abstrait impose l'ulisation de fonctions intermédiaires pour modifier le type. Dans la grande majorité des cas, ces types représentent des structures qui contiennent des informations internes qui ne sont pas destinées à être modifiées par l'utilisateur final.

21.2 Tableau dynamique

Un tableau dynamique aussi appelé *vecteur* est, comme son nom l'indique, alloué dynamiquement dans le *heap* en fonction des besoins. Vous vous rappelez que le *heap* grossit à chaque appel de `malloc` et diminue à chaque appel de `free`.

Un tableau dynamique est souvent spécifié par un facteur de croissance (rien à voir avec les hormones). Lorsque le tableau est plein et que l'on souhaite rajouter un nouvel élément, le tableau est réalloué dans un autre espace mémoire plus grand avec la fonction `realloc`. Cette dernière n'est rien d'autre qu'un `malloc` suivi d'un `memcopy` suivi d'un `free`. Un nouvel espace mémoire est réservé, les données sont copiées du premier espace vers le nouveau, et enfin le premier espace est libéré. Voici un exemple :

```
// Alloue un espace de trois chars
char *buffer = malloc(3);

// Rempli le buffer
buffer[0] = 'h';
buffer[1] = 'e';
buffer[2] = 'l'; // Le buffer est plein...

// Augmente dynamiquemenmt la taille du buffer à 5 chars
char *tmp = realloc(buffer, 5);
assert(tmp != NULL);
buffer = tmp;

// Continue de remplir le buffer
buffer[3] = 'l';
buffer[4] = 'o'; // Le buffer est à nouveau plein...

// Libère l'espace mémoire utilisé
free(buffer);
```

La taille du nouvel espace mémoire est plus grande d'un facteur donné que l'ancien espace. Selon les langages de programmation et les compilateurs, ces facteurs sont compris entre 3/2 et 2. C'est à dire que la taille du tableau prendra les tailles de 1, 2, 4, 8, 16, 32, etc.

Lorsque le nombre d'éléments du tableau devient inférieur du facteur de croissance à la taille effective du tableau, il est possible de faire l'opération inverse, c'est-à-dire réduire la taille allouée. En pratique cette opération est rarement implémentée, car peu efficace (c.f. cette réponse sur stackoverflow).

21.2.1 Anatomie

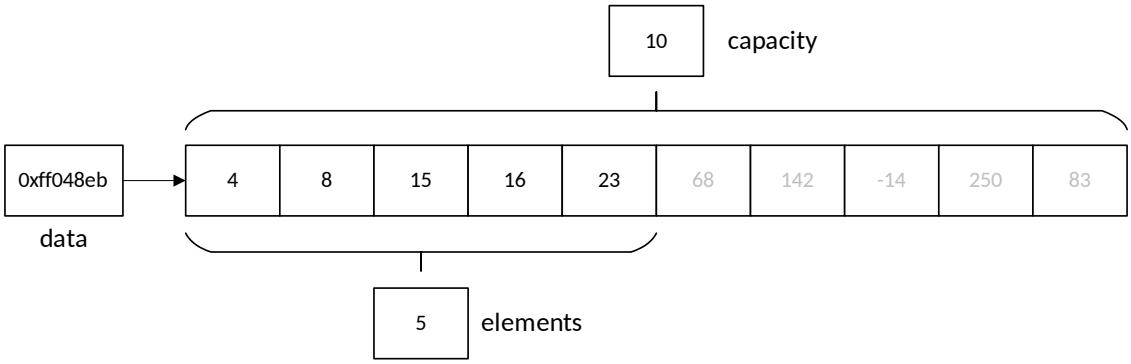
Un tableau dynamique est représenté en mémoire comme un contenu séquentiel qui possède un début et une fin. On appelle son début la **tête** ou *head* et la fin du tableau sa **queue** ou *tail*. Selon que l'on souhaite ajouter des éléments au début ou à la fin du tableau la complexité n'est pas la même.

Nous définirons par la suite le vocabulaire suivant :

Action	Terme technique
Ajout d'un élément à la tête du tableau	*unshift*
Ajout d'un élément à la queue du tableau	*push*
Suppression d'un élément à la tête du tableau	*shift*
Suppression d'un élément à la queue du tableau	*pop*

Nous comprenons rapidement qu'il est plus compliqué d'ajouter ou de supprimer un élément depuis la tête du tableau, car il est nécessaire ensuite de déplacer chaque élément (l'élément 0 devient l'élément 1, l'élément 1 devient l'élément 2...).

Un tableau dynamique peut être représenté par la figure suivante :

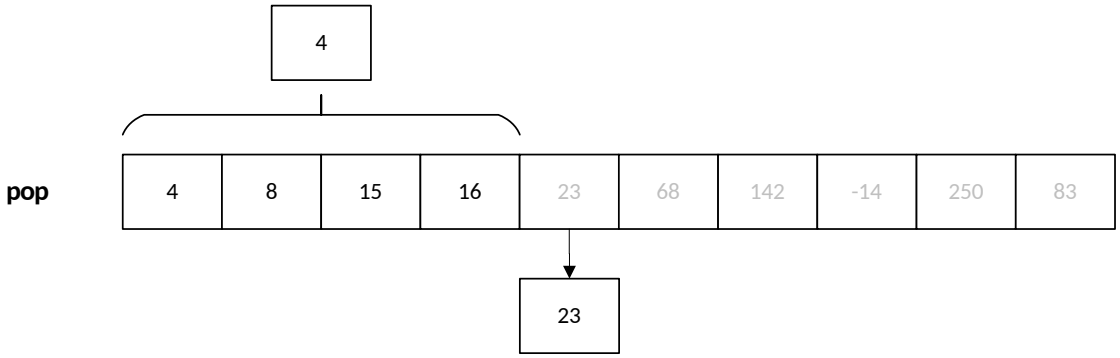


Un espace mémoire est réservé dynamiquement sur le tas. Comme `malloc` ne retourne pas la taille de l'espace mémoire alloué mais juste un pointeur sur cet espace, il est nécessaire de conserver dans une variable la capacité du tableau. Notons qu'un tableau de 10 `int32_t` représentera un espace mémoire de 4x10 bytes, soit 40 bytes. La mémoire ainsi réservée par `malloc` n'est généralement pas vide mais elle contient des valeurs, vestige d'une ancienne allocation mémoire d'un d'autre programme depuis que l'ordinateur a été allumé. Pour connaître le nombre d'éléments effectifs du tableau il faut également le mémoriser. Enfin, le pointeur sur l'espace mémoire est aussi mémorisé.

Les composants de cette structure de donnée sont donc :

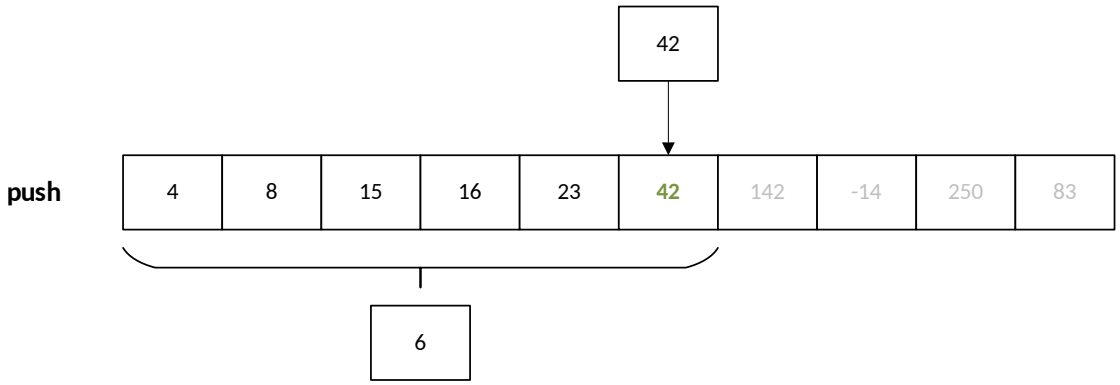
— Un entier non signé `size_t` représentant la capacité totale du tableau dynamique à un instant T.
— Un entier non signé `size_t` représentant le nombre d'éléments effectivement dans le tableau.
— Un pointeur sur un entier `int *` contenant l'adresse mémoire de l'espace alloué par `malloc`.
— Un espace mémoire alloué par `malloc` et contenant des données.

L'opération `pop` retire l'élément de la fin du tableau. Le nombre d'éléments est donc ajusté en conséquence.



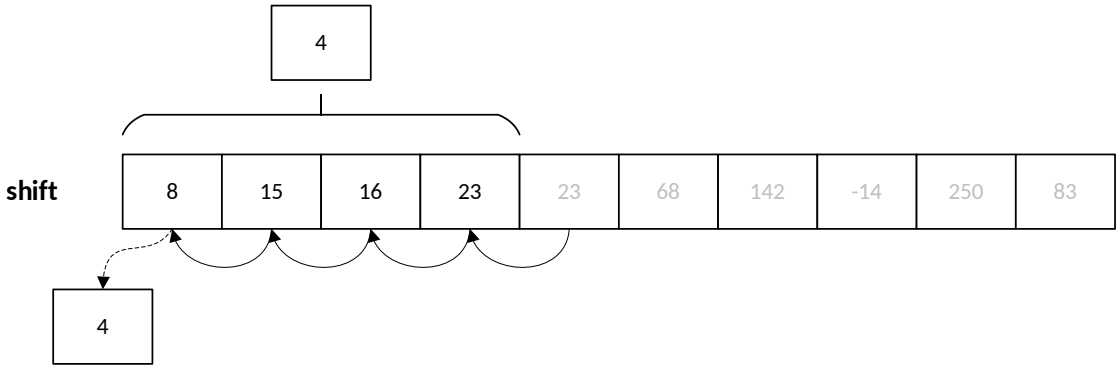

```
if (elements <= 0) exit(EXIT_FAILURE);
int value = data[--elements];
```

L'opération `push` ajoute un élément à la fin du tableau.




```
if (elements >= capacity) exit(EXIT_FAILURE);
data[elements++] = value;
```

L'opération `shift` retire un élément depuis le début. L'opération à une complexité de O(n) puisqu'à chaque opération il est nécessaire de déplacer chaque éléments qu'il contient.

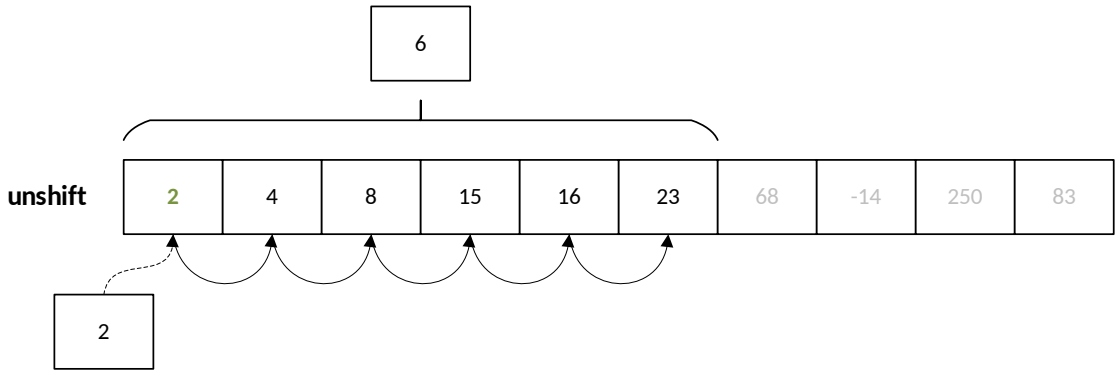



```
if (elements <= 0) exit(EXIT_FAILURE);
int value = data[0];
for (int k = 0; k < capacity; k++)
    data[k] = data[k+1];
```

Une optimisation peut être faite en déplacant le pointeur de donnée de 1 permettant de réduite la complexité à O(1) :

```
if (elements <= 0) exit(EXIT_FAILURE);
if (capacity <= 0) exit(EXIT_FAILURE);
int value = data[0];
data++;
capacity--;
```

Enfin, l'opération `unshift` ajoute un élément depuis le début du tableau :




```
for (int k = elements; k >= 1; k--)
    data[k] = data[k - 1];
data[0] = value;
```

Dans le cas ou le nombre d'éléments atteint la capacité maximum du tableau, il est nécessaire de réallouer l'espace mémoire avec `realloc`. Généralement on se contente de doubler l'espace alloué.

```
if (elements >= capacity) {
    data = realloc(data, capacity *= 2);
}
```

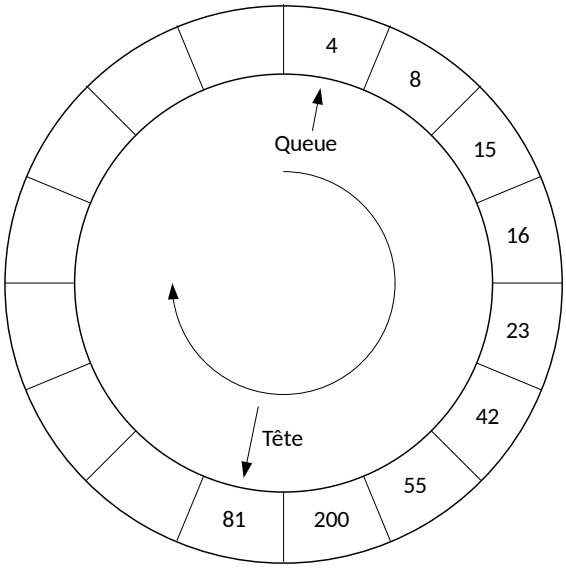
21.3 Buffer circulaire

Un tampon circulaire est généralement d'une taille fixe et possède deux pointeurs. L'un pointant sur le dernier élément (*tail*) et l'un sur le premier élément (*head*).

Lorsqu'un élément est supprimé du buffer, le pointeur de fin est incrémenté. Lorsqu'un élément est ajouté, le pointeur de début est incrémenté.

Pour permettre la circulation, les indices sont calculés modulo la taille du buffer.

Il est possible de représenter schématiquement ce buffer comme un cercle et ses deux pointeurs :



Le nombre d'éléments dans le buffer est la différence entre le pointeur de tête et le pointeur de queue, modulo la taille du buffer. Néanmoins, l'opérateur `%` en C ne fonctionne que sur des nombres positifs et ne retourne pas le résidu positif le plus petit. En sommes, `-2 % 5` devrait donner `3`, ce qui est le cas en Python, mais en C, en C++ ou en PHP la valeur retournée est `-2`. Le modulo vrai, mathématiquement correct peut être calculé ainsi :

```
((A % M) + M) % M
```

Les indices sont bouclés sur la taille du buffer, l'élément suivant est donc défini par :

```
(i + 1) % SIZE
```

Voici une implémentation possible du buffer circulaire :

```
#define SIZE 16
#define MOD(A, M) (((A % M) + M) % M)
#define NEXT(A) (((A) + 1) % SIZE)

typedef struct Ring {
    int buffer[SIZE];
    int head;
    int tail;
} Ring;

void init(Ring *ring) {
    ring->head = ring->tail = 0;
}

int count(Ring *ring) {
    return MOD(ring->head - ring->tail, size);
}

bool is_full(Ring *ring) {
    return count(ring) == SIZE - 1;
}

bool is_empty(Ring *ring) {
    return ring->tail == ring->head;
}

int* enqueue(Ring *ring, int value) {
    if (is_full(ring)) return NULL;
    ring->buffer[ring->head] = value;
    int *el = &ring->buffer[ring->head];
    ring->head = NEXT(ring->head);
    return el;
}

int* dequeue(Ring *ring) {
    if (is_empty(ring)) return NULL;
    int *el = &ring->buffer[ring->tail];
    ring->tail = NEXT(ring->tail);
    return el;
}
```

21.4 Listes chaînées

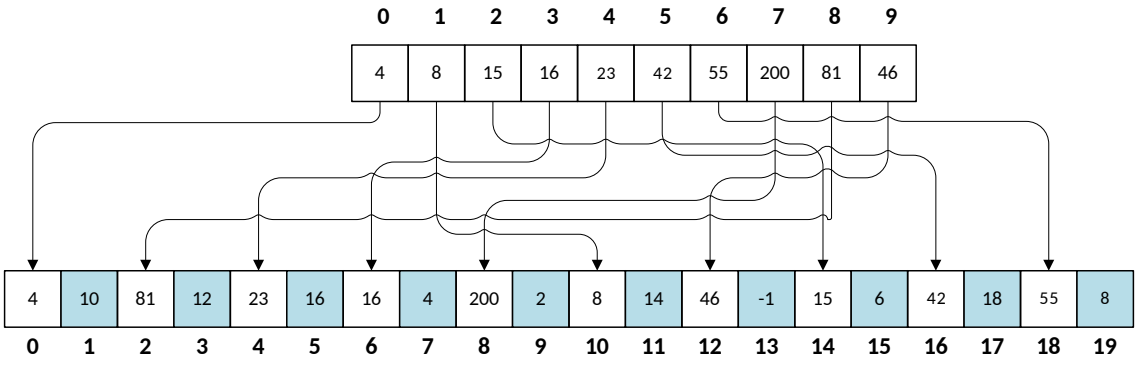
On s'aperçois vite avec les tableaux que certaines opérations sont plus coûteuses que d'autres. Ajouter ou supprimer un élément à la fin du tableau coûte $O(1)$ amorti, mais ajouter ou supprimer un élément à l'intérieur du tableau coûte $O(n)$ du fait qu'il est nécessaire de déplacer tous les éléments qui suivent l'élément concerné.

Une possible solution à ce problème serait de pouvoir s'affranchir du lien entre les éléments et leurs positions en mémoire relative les uns aux autres.

Pour illustrer cette idée, imaginons un tableau statique dans lequel chaque élément est décrit par la structure suivante :

```
struct Element {
    int value;
    int index_next_element;
};

struct Element elements[100];
```

Considérons les dix premiers éléments de la séquence de nombre A130826 dans un tableau statique. Ensuite répartissons ces valeurs aléatoirement dans notre tableau *elements* déclaré plus haut entre les indices 0 et 19.



On observe sur la figure ci-dessus que les éléments n'ont plus besoin de se suivre en mémoire car il est possible facilement de chercher l'élément suivant de la liste avec cette relation :

```
struct Element current = elements[4];
struct Element next = elements[current.index_next_element]
```

De même, insérer une nouvelle valeur *13* après la valeur *42* est très facile :

```
// Recherche de l'élément contenant la valeur 42
struct Element el = elements[0];
while (el.value != 42 && el.index_next_element != -1) {
    el = elements[el.index_next_element];
```

(suite sur la page suivante)

(suite de la page précédente)

```
}
if (el.value != 42) abort();

// Recherche d'un élément libre
const int length = sizeof(elements) / sizeof(elements[0]);
int k;
for (k = 0; k < length; k++)
    if (elements[k].index_next_element == -1)
        break;
assert(k < length && elements[k].index_next_element == -1);

// Création d'un nouvel élément
struct Element new = (Element){
    .value = 13,
    .index_next_element = -1
};

// Insertion de l'élément quelque part dans le tableau
el.index_next_element = k;
elements[el.index_next_element] = new;
```

Cette solution d'utiliser un lien vers l'élément suivant et s'appelle liste chaînée. Chaque élément dispose d'un lien vers l'élément suivant situé quelque part en mémoire. Les opérations d'insertion et de suppression au milieu de la chaîne sont maintenant effectuées en $O(1)$ contre $O(n)$ pour un tableau standard. En revanche l'espace nécessaire pour stocker ce tableau est doublé puisqu'il faut associer à chaque valeur le lien vers l'élément suivant.

D'autre part, la solution proposée n'est pas optimale :

— L'élément 0 est un cas particulier qu'il faut traiter différemment. Le premier élément de la liste doit toujours être positionné à l'indice 0 du tableau. Insérer un nouvel élément en début de tableau demande de déplacer cet élément ailleurs en mémoire.
— Rechercher un élément libre prend du temps.
— Supprimer un élément dans le tableau laisse une place mémoire vide. Il devient alors difficile de savoir où sont les emplacement mémoire disponibles

Une liste chaînée est une structure de données permettant de lier des éléments structurés entre eux. La liste est caractérisée par :

— un élément de tête (*head*),
— un élément de queue (*tail*).

Un élément est caractérisé par :

— un contenu (*payload*),
— une référence vers l'élément suivant et/ou précédent dans la liste.

Les listes chaînées réduisent la complexité liée à la manipulation d'éléments dans une liste. L'empreinte mémoire d'une liste chaînée est plus grande qu'avec un tableau, car à chaque élément de donnée est associé un pointeur vers l'élément suivant ou précédent.

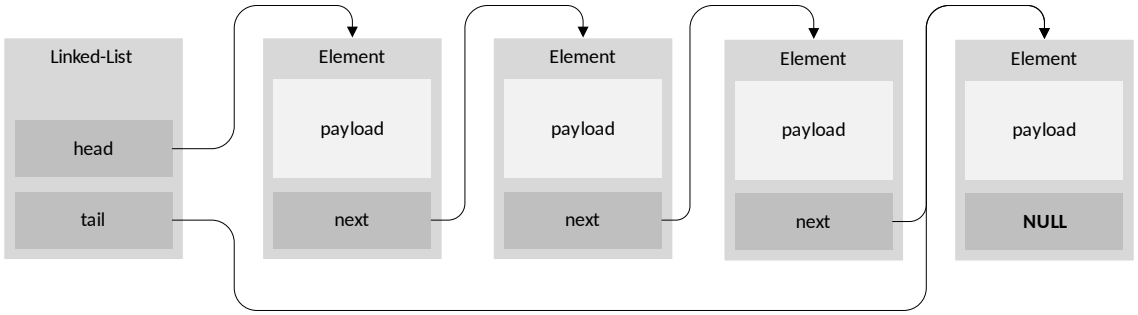
Ce surcoût est souvent part du compromis entre la complexité d'exécution du code et la mémoire utilisée par ce programme.

Structure de donnée	Pire cas			
	Insertion	Suppression	Recherche |	
			Trié	Pas trié
Tableau, pile, queue	$O(n)$	$O(n)$	$O(log(n))$	$O(n)$
Liste chaînée simple	$O(1)$	$O(1)$	$O(n)$	$O(n)$

21.4.1 Liste simplement chaînée (*linked-list*)

La figure suivante illustre un set d'éléments liés entre eux à l'aide d'un pointeur rattaché à chaque élément. On peut s'imaginer que chaque élément peut se situer n'importe où en mémoire et qu'il n'est alors pas indispensable que les éléments se suivent dans l'ordre.

Il est indispensable de bien identifier le dernier élément de la liste grâce à son pointeur associé à la valeur NULL.




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

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

struct Element
{
    struct Point point;
    struct Element* next;
};

int main(void)
```

(suite sur la page suivante)

(suite de la page précédente)

```
{
    struct Element a = {.point = {1,2,3}, .next = NULL};
    struct Element b = {.point = {4,5,6}, .next = &a};
    struct Element c = {.point = {7,8,9}, .next = &b};

    a.next = &c;

    struct Element* walk = &a;

    for (size_t i = 0; i < 10; i++)
    {
        printf("%d. P(x, y, z) = %0.2f, %0.2f, %0.2f\n",
            i,
            walk->point.x,
            walk->point.y,
            walk->point.z
        );

        walk = walk->next;
    }
}
```

21.4.2 Opérations sur une liste chaînée

- Création
- Nombre d'éléments
- Recherche
- Insertion
- Suppression
- Concaténation
- Destruction

Lors de la création d'un élément, on utilise principalement le mécanisme de l'allocation dynamique ce qui permet de récupérer l'adresse de l'élément et de faciliter sa manipulation au travers de la liste. Ne pas oublier de libérer la mémoire allouée pour les éléments lors de leur suppression…

Calcul du nombre d'éléments dans la liste

Pour évaluer le nombre d'éléments dans une liste, on effectue le parcours de la liste à partir de la tête, et on passe d'élément en élément grâce au champ *next* de la structure `Element`. On incrément le nombre d'éléments jusqu'à ce que le pointeur *next* soit égal à `NULL`.

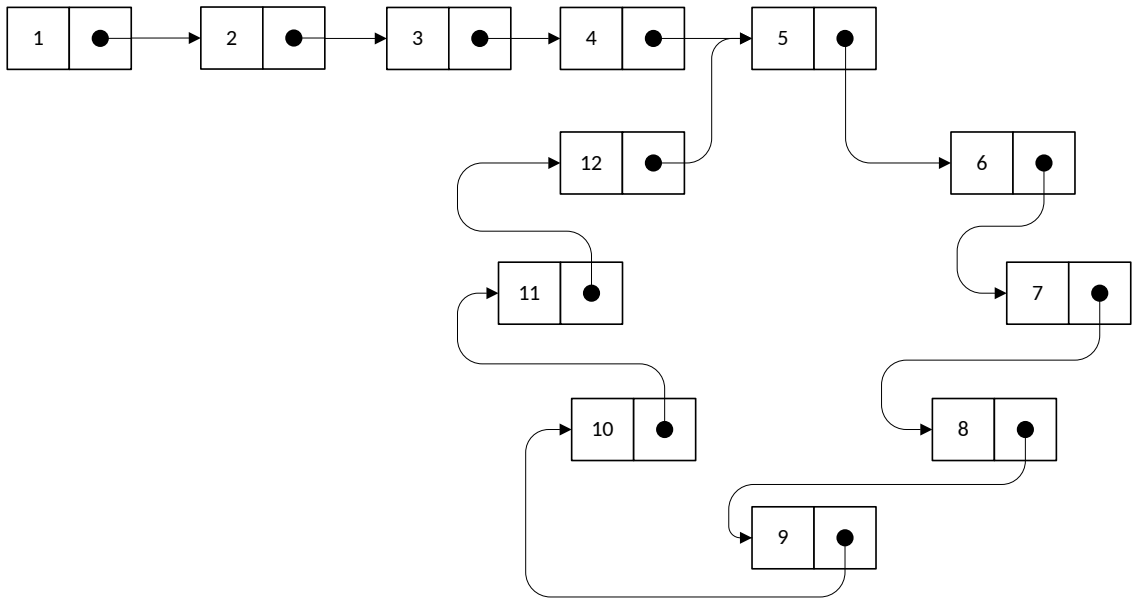
```
size_t count = 0;
```

(suite sur la page suivante)

(suite de la page précédente)

```
for (Element *e = &head; e != NULL; e = e->next)
    count++;
}
```

Attention, cette technique ne fonctionne pas dans tous les cas, spécialement lorsqu'il y a des boucles dans la liste chaînée. Prenons l'exemple suivant :

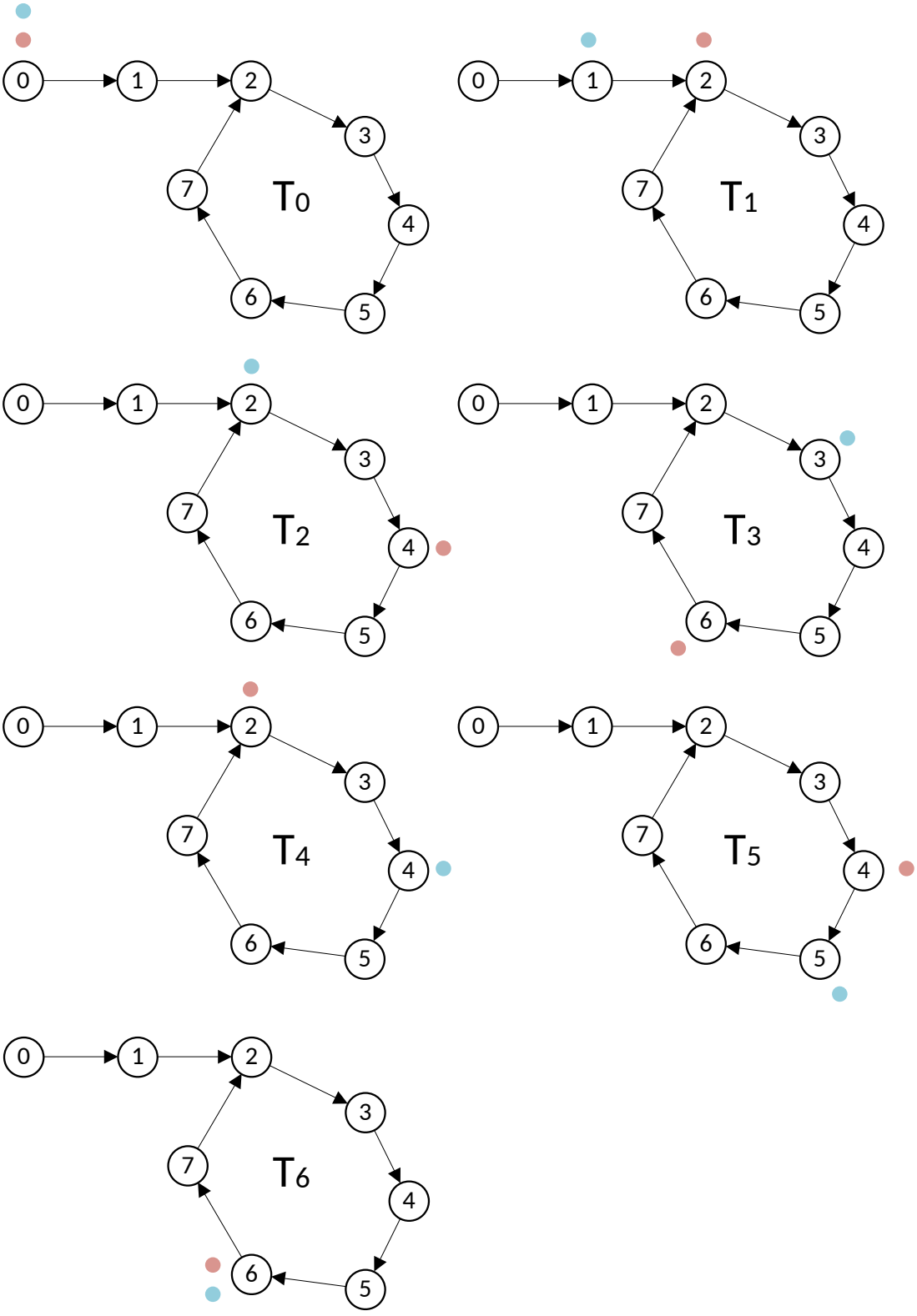


La liste se terminant par une boucle, il n'y aura jamais d'élément de fin et le nombre d'éléments calculé sera infini. Or, cette liste a un nombre fixe d'éléments. Comment donc les compter ?

Il existe un algorithme nommé détection de cycle de Robert W. Floyd aussi appelé *algorithme du lièvre et de la tortue*. Il consiste à avoir deux pointeurs qui parcourent la liste chaînée. L'un avance deux fois plus vite que le second.

```
size_t compute_length(Element* head)
{
    size_t count = 0;

    Element* slow = head;
    Element* fast = head;

    while (fast != NULL && fast->next != NULL) {
        slow = slow->next;
        fast = fast->next->next;

        count++;

        if (slow == fast) {
```

(suite sur la page suivante)


0
1
2
3
4
5
6
7
T0
T1
T2
T3
T4
T5
T6

(suite de la page précédente)

```
            // Collision
            break;
        }
    }

    // Case when no loops detected
    if (fast == NULL || fast->next == NULL) {
        return count;
    }

    // Move slow to head, keep fast at meeting point.
    slow = head;
    while (slow != fast) {
        slow = slow->next;
        fast = fast->next;

        count--;
    }

    return count;
}
```

Une bonne idée pour se simplifier la vie est simplement d'éviter la création de boucles.

Insertion

L'insertion d'un élément dans une liste chaînée peut-être implémentée de la façon suivante :

```
Element* insert_after(Element* e, void* payload)
{
    Element* new = malloc(sizeof(Element));

    memcpy(new->payload, payload, sizeof(new->payload));

    new->next = e->next;
    e->next = new;

    return new;
}
```

Suppression

La suppression implique d'accéder à l'élément parent, il n'est donc pas possible à partir d'un élément donné de le supprimer de la liste.

```
void delete_after(Element* e)
{
    e->next = e->next->next;
    free(e);
}
```

Recherche

Rechercher dans une liste chaînée est une question qui peut-être complexe et il est nécessaire de ce poser un certain nombre de questions :

— Est-ce que la liste est triée ?
— Combien d'espace mémoire puis-je utiliser ?

On sait qu'une recherche idéale s'effectue en $O(log(n))$, mais que la solution triviale en $O(n)$ est la suivante :

21.5 Liste doublement chaînée

21.5.1 Liste chaînée XOR

L'inconvénient d'une liste doublement chaînée est le surcoût nécessaire au stockage d'un élément. Chaque élément contient en effet deux pointeurs sur l'élément précédent (*prev*) et suivant (*next*).

```
...  A       B         C         D         E  ...
       ->  next ->  next  ->  next  ->
       <-  prev <-  prev  <-  prev  <-
```

Cette liste chaînée particulière compresse les deux pointeurs en un seul en utilisant l'opération XOR (dénotée).

```
...  A       B         C         D         E  ...
       <->  A⊕C  <->  B⊕D  <->  C⊕E  <->
```

Lorsque la liste est traversée de gauche à droite, il est possible de facilement reconstuire le pointeur de l'élément suivant à partir de l'adresse de l'élément précédent.

Les inconvénients de cette structure sont :

— Difficultés de débogage
— Complexité de mise en oeuvre

L'avantage principal étant le gain de place en mémoire.

21.6 Liste chaînée déroulée (Unrolled linked list)

21.7 Arbre binaire de recherche

L'objectif de cette section n'est pas d'entrer dans les détails des arbres binaires dont la théorie requiert un ouvrage dédié, mais de vous sensibiliser à l'existence de ces structures de données qui sont à la base de beaucoup de langage de haut niveau comme C++, Python ou C#.

L'arbre binaire, n'est rien d'autre qu'une liste chaînée comportant deux enfants un `left` et un `right` :

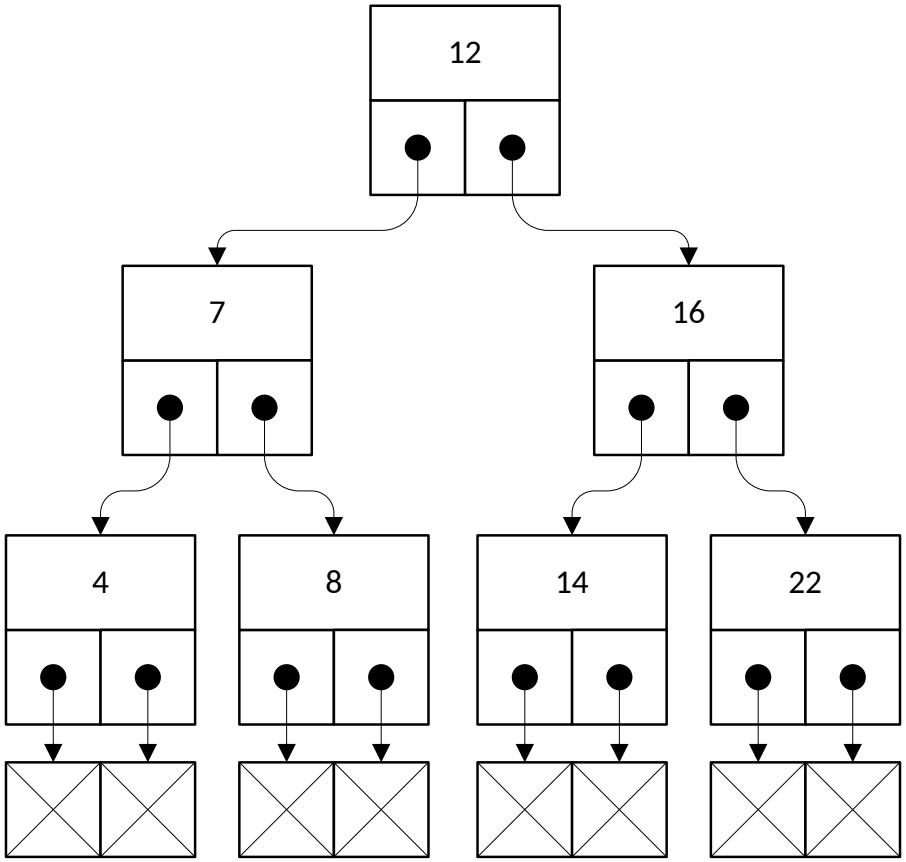


Fig. 21.1 – Arbre binaire équilibré

Lorsqu'il est équilibré, un arbre binaire comporte autant d'éléments à gauche qu'à droite et lorsqu'il est correctement rempli, la valeur d'un élément est toujours :

- La valeur de l'enfant de gauche est inférieure à celle de son parent
- La valeur de l'enfant de droite est supérieure à celle de son parent

Cette propriété est très appréciée pour rechercher et insérer des données complexes. Admettons que l'on a un registre patient du type :

```c
struct patient {
    size_t id;
    char firstname[64];
    char lastname[64];
    uint8_t age;
}

typedef struct node {
    struct patient data;
    struct node* left;
    struct node* right;
} Node;
```

Si l'on cherche le patient numéro 612, il suffit de parcourir l'arbre de façon dichotomique :

```c
Node* search(Node* node, size_t id)
{
    if (node == NULL)
        return NULL;

    if (node->data.id == id)
        return node;

    return search(node->data.id > id ? node->left : node->right,␣
↪id);
}
```

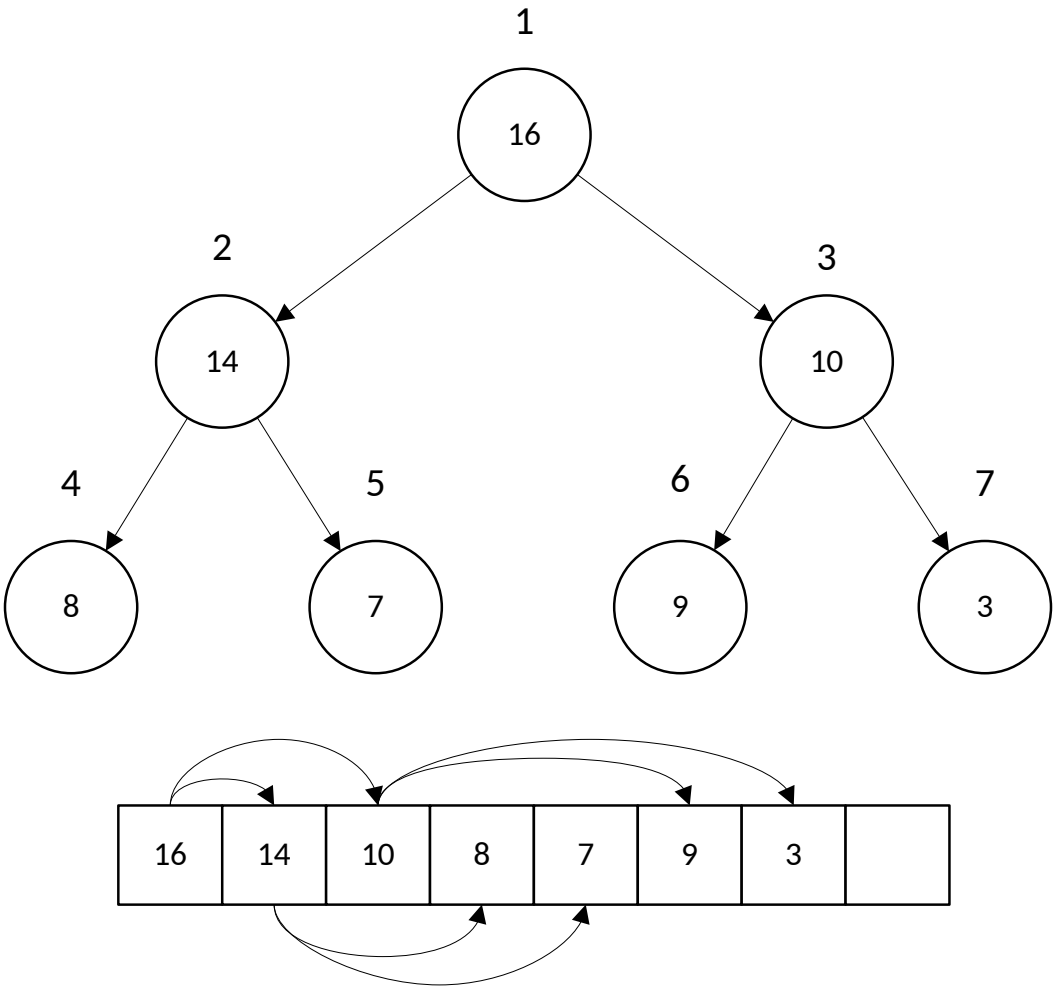
L'insertion et la suppression d'éléments dans un arbre binaire fait appel à des rotations, puisque les éléments doivent être insérés dans le correct ordre et que l'arbre, pour être performant doit toujours être équilibré. Ces rotations sont donc des mécanismes de ré-équilibrage de l'arbre ne sont pas triviaux, mais dont la complexité d'exécution reste simple, et donc performante.

21.8 Heap

La structure de donnée heap aussi nommée tas ne doit pas être confondue avec le tas utilisé en allocation dynamique. Il s'agit d'une forme particulière de l'arbre binaire dit "presque complet", dans lequel la différence de niveau entre les feuilles n'excède pas 1. C'est à dire que toutes les feuilles sont à une distance identique de la racine plus ou moins 1.

Un tas peut aisément être représenté sous forme de tableau en utilisant la règle suivante :

Cible	Début à 0	Début à 1
Enfant de gauche	$2 * k + 1$	$2 * k$
Enfant de droite	$2 * k + 2$	$2 * k + 1$
Parent	$floor(k - 1)/2$	$floor(k)/2$


1
16
2
14
3
10
4
8
5
7
6
9
7
3
16
14
10
8
7
9
3

21.9 Queue prioritaire

Une queue prioritaire ou *priority queue*, est une queue dans laquelle les éléments sont traités par ordre de priorité. Imaginons des personnalités, toutes atteintes d'une rage de dents et qui font la queue chez un dentiste aux moeurs discutables. Ce dernier ne prendra pas ses patients par ordre d'arrivée mais, par importance aristocratique.

```
typedef struct Person {
   char *name;
   enum SocialStatus {
       PEON;
       WORKER;
       ENGINEER;
       DOCTOR;
       PROFESSOR;
       PRESIDENT;
       SUPERHERO;
   } status;
} Person;

int main() {
    ProrityQueue queue;
    queue_init(queue);

    for(int i = 0; i < 100; i++) {
       queue_enqueue(queue, (Person) {
          .name = random_name(),
          .status = random_status()
       });

       Person person;
       queue_dequeue(queue, &person);
       dentist_heal(person);
    }
}
```

La queue prioritaire dispose donc aussi des méthodes `enqueue` et `dequeue` mais le `dequeue` retournera l'élément le plus prioritaire de la liste. Ceci se traduit par trier la file d'attente à chaque opération `enqueue` ou `dequeue`. L'une de ces deux opération pourrait donc avoir une complexité de $O(nlogn)$. Heureusement, il existe méthodes de tris performantes si un tableau est déjà trié et qu'un seul nouvel élément y est ajouté.

L'implémentation de ce type de structure de donnée s'appuie le plus souvent sur un *heap*, soit construit à partir d'un tableau statique, soit un tableau dynamique.

21.10 Tableau de Hachage

Les tableaux de hachage (*Hash Table*) sont une structure particulière dans laquelle une fonction dite de *hachage* est utilisée pour transformer les entrées en des indices d'un tableau.

L'objectif est de stocker des chaînes de caractères correspondant a des noms simples ici utilisés pour l'exemple. Une possible répartition serait la suivante :

Si l'on cherche l'indice correspondant à `Ada`, il convient de pouvoir calculer la valeur de l'indice correspondant à partir de la valeur de la chaîne de caractère. Pour calculer cet indice aussi appelé *hash*, il existe une infinité de méthodes. Dans ce exemple considérons une méthode simple. Chaque lettre est identifiée par sa valeur ASCII et la somme de toutes les valeurs ASCII est calculée. Le modulo 10 est ensuite calculé sur cette somme pour obtenir une valeur entre 0 et 9. Ainsi nous avons les calculs suivants :

```
Nom    Valeurs ASCII     Somme  Modulo 10
---    -------------     -----  ---------
Mia -> {77, 105, 97}  -> 279 -> 4
Tim -> {84, 105, 109} -> 298 -> 1
Bea -> {66, 101, 97}  -> 264 -> 0
Zoe -> {90, 111, 101} -> 302 -> 5
Jan -> {74, 97, 110}  -> 281 -> 6
Ada -> {65, 100, 97}  -> 262 -> 9
Leo -> {76, 101, 111} -> 288 -> 2
Sam -> {83, 97, 109}  -> 289 -> 3
Lou -> {76, 111, 117} -> 304 -> 7
Max -> {77, 97, 120}  -> 294 -> 8
Ted -> {84, 101, 100} -> 285 -> 10
```

Pour trouver l'indice de `"Mia"` il suffit donc d'appeler la fonction suivante :

```c
int hash_str(char *s) {
    int sum = 0;
    while (*s != '\0') sum += s++;
    return sum % 10;
}
```

L'assertion suivante est donc vraie :

```c
assert(strcmp(table[hash_str("Mia")], "Mia") == 0);
```

Rechercher `"Mia"` et obtenir `"Mia"` n'est certainement pas l'exemple le plus utile. Néanmoins il est possible d'encoder plus qu'une chaîne de caractère et utiliser plutôt une structure de donnée :

```
struct Person {
    char name[3 + 1 /* '\0' */];
    struct {
        int month;
        int day;
        int year;
    } born;
    enum {
        JOB_ASTRONOMER,
        JOB_INVENTOR,
        JOB_ACTRESS,
        JOB_LOGICIAN,
        JOB_BIOLOGIST
    } job;
    char country_code; // For example 41 for Switzerland
};
```

Dans ce cas, le calcul du hash se ferait sur la première clé d'un élément :

```
int hash_person(struct Person person) {
    int sum = 0;
    while (*person.name != '\0') sum += s++;
    return sum % 10;
}
```

L'accès à une personne à partir de la clé se résoud donc en `O(1)` car il n'y a aucune itération ou recherche à effectuer.

Cette vidéo YouTube explique bien le fonctionnement des tableaux de hachage.

21.10.1 Collisions

Lorsque la fonction de hachage est mal choisie, un certain nombre de collision peuvent apparaître. Si l'on souhaite par exemple ajouter les personnes suivantes :

```
Sue -> {83, 117, 101} -> 301 -> 4
Len -> {76, 101, 110} -> 287 -> 1
```

On voit que les positions `4` et `1` sont déjà occupées par Mia et Tim.

Une stratégie de résolution s'appelle Open adressing. Parmi les possibilités de cette stratégie le *linear probing* consiste à vérifier si la position du tableau est déjà occupée et en cas de collision, chercher la prochaine place disponible dans le tableau :

```
Person people[10] = {0}
```

(suite sur la page suivante)

(suite de la page précédente)

```
// Add Mia
Person mia = {.name="Mia", .born={.day=1,.month=4,.year=1991}};
int hash = hash_person(mia);
while (people[hash].name[0] != '\0') hash++;
people[hash] = mia;
```

Récupérer une valeur dans le tableau demande une comparaison supplémentaire :

```
char key[] = "Mia";
int hash = hash_str(key)
while (strcmp(people[hash], key) != 0) hash++;
Person person = people[hash];
```

Lorsque le nombre de collision est négligeable par rapport à la table de hachage la recherche d'un élément est toujours en moyenne égale à $O(1)$ mais lorsque le nombre de collision est prépondérant, la complexité se rapproche de celle de la recherche linéaire $O(n)$ et on perd tout avantage à cette structure de donnée.

Dans le cas extrême, pour garantir un accès unitaire pour tous les noms de trois lettres, il faudrait un tableau de hachage d'une taille $26^3 = 17576$ personnes. L'emprunte mémoire peut être considérablement réduite en stockant non pas une structure `struct Person` mais plutôt l'adresse vers cette structure :

```
struct Person *people[26 * 26 * 26] = { NULL };
```

Dans ce cas exagéré, la fonction de hachage pourrait être la suivante :

```
int hash_name(char name[4]) {
    int base = 26;
    return
        (name[0] - 'A') * 1 +
        (name[1] - 'a') * 26 +
        (name[2] - 'a') * 26 * 26;
}
```

21.10.2 Facteur de charge

Le facteur de charge d'une table de hachage est donné par la relation :

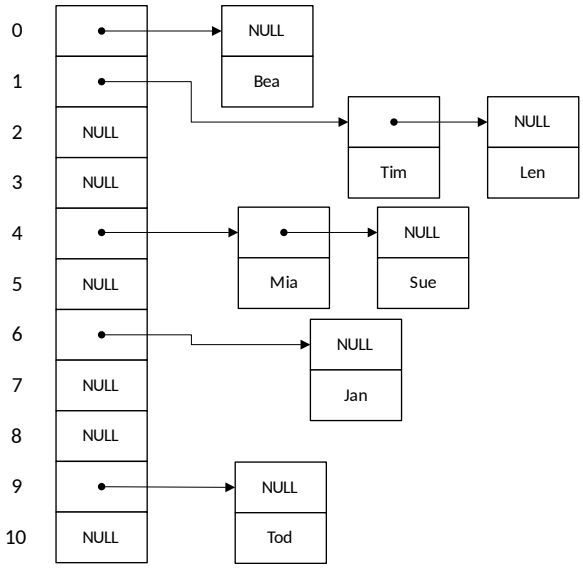
$$\text{Facteur de charge} = \frac{\text{Nombre total d'éléments}}{\text{Taille de la table}}$$

Plus ce facteur de charge est élevé, dans le cas du *linear probing*, moins bon sera performance de la table de hachage.

Certains algorithmes permettent de redimensionner dynamiquement la table de hachage pour conserver un facteur de charge le plus faible possible.

21.10.3 Chaînage

Le chaînage ou *chaining* est une autre méthode pour mieux gérer les collisions. La table de hachage est couplée à une liste chaînée.



21.10.4 Fonction de hachage

Nous avons vu plus haut une fonction de hachage calculant le modulo sur la somme des caractères ASCII d'une chaîne de caractères. Nous avons également vu que cette fonction de hachage est source de nombreuses collisions. Les chaînes `"Rea"` ou `"Rae"` auront les même *hash* puisqu'ils contiennent les même lettres. De même une fonction de hachage qui ne réparti pas bien les éléments dans la table de hachage sera mauvaise. On sait par exemple que les voyelles sont nombreuses dans les mots et qu'il n'y en a que six et que la probabilité que nos noms de trois lettres contiennent une voyelle en leur milieu est très élevée.

L'idée générale des fonctions de hachage est de répartir **uniformément** les clés sur les indices de la table de hachage. L'approche la plus courante est de mélanger les bits de notre clé dans un processus reproductible.

Une idée **mauvaise** et **à ne pas retenir** pourrait être d'utiliser le caractère pseudo-aléatoire de `rand` pour hacher nos noms :

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

int hash(char *str, int mod) {
    int h = 0;
    while(*str != '\0') {
        srand(h + *str++);
```

(suite sur la page suivante)

(suite de la page précédente)

```
        h = rand();
    }
    return h % mod;
}

int main() {
    char *names[] = {
        "Bea", "Tim", "Len", "Sam", "Ada", "Mia",
        "Sue", "Zoe", "Rae", "Lou", "Max", "Tod"
    };
    for (int i = 0; i < sizeof(names) / sizeof(*names); i++)
        printf("%s : %d\n", names[i], hash(names[i], 10));
}
```

Cette approche nous donne une assez bonne répartition :

```
$ ./a.out
Bea : 2
Tim : 3
Len : 0
Sam : 3
Ada : 4
Mia : 3
Sue : 6
Zoe : 5
Rae : 8
Lou : 0
Max : 3
Tod : 1
```

Dans la pratique, on utilisera volontier des fonctions de hachage utilisés en cryptographies tels que MD5 ou *SHA*. Considérons par exemple la première partie du poème Chanson de Pierre Corneille :

```
$ cat chanson.txt
Si je perds bien des maîtresses,
J'en fais encor plus souvent,
Et mes voeux et mes promesses
Ne sont que feintes caresses,
Et mes voeux et mes promesses
Ne sont jamais que du vent.

$ md5sum chanson.txt
699bfc5c3fd42a06e99797bfa635f410  chanson.txt
```

Le *hash* de ce texte est exprimé en hexadécimal (`0x699bfc5c3fd42a06e99797bfa635f410`). Converti en décimal `140378864046454182829995736237591622672` il peut être réduit en utilisant le modulo. Voici un exemple en C :

```c
#include <stdlib.h>
#include <stdio.h>
#include <openssl/md5.h>
#include <string.h>

int hash(char* str, int mod) {
    // Compute MD5
    unsigned int output[4];
    MD5_CTX md5;
    MD5_Init(&md5);
    MD5_Update(&md5, str, strlen(str));
    MD5_Final((char*)output, &md5);

    // 128-bits --> 32-bits
    unsigned int h = 0;
    for (int i = 0; i < sizeof(output)/sizeof(*output); i++) {
        h ^= output[i];
    }

    // 32-bits --> mod
    return h % mod;
}

int main() {
    char *text[] = {
        "La poule ou l'oeuf?",
        "Les pommes sont cuites!",
        "Aussi lentement que possible",
        "La poule ou l'oeuf.",
        "La poule ou l'oeuf!",
        "Aussi vite que nécessaire",
        "Il ne faut pas lâcher la proie pour l'ombre.",
        "Le mieux est l'ennemi du bien",
    };

    for (int i = 0; i < sizeof(text) / sizeof(*text); i++)
        printf("% 2d. %s\n", hash(text[i], 10), text[i]);
}
```

```
$ gcc hash.c -lcrypto
$ ./a.out
4. La poule ou l'oeuf?
3. Les pommes sont cuites!
7. Aussi lentement que possible
2. La poule ou l'oeuf.
5. La poule ou l'oeuf!
6. Aussi vite que nécessaire
8. Il ne faut pas lâcher la proie pour l'ombre.
1. Le mieux est l'ennemi du bien
```

On peut constater qu'ici les indices sont bien répartis et que la fonction de hachage choisie semble uniforme.

21.11 Piles ou LIFO (*Last In First Out*)

Une pile est une structure de donnée très similaire à un tableau dynamique mais dans laquelle les opérations sont limitées. Par exemple, il n'est possible que :

- d'ajouter un élément (*push*) ;
- retirer un élément (*pop*) ;
- obtenir le dernier élément ajouté (*peek*) ;
- tester si la pile est vide (*is_empty*) ;
- tester si la pile est pleine avec (*is_full*).

Une utilisation possible de pile sur des entiers serait la suivante :

```c
#include "stack.h"

int main() {
    Stack stack;
    stack_init(&stack);

    stack_push(42);
    assert(stack_peek() == 42);

    stack_push(23);
    assert(!stack_is_empty());

    assert(stack_pop() == 23);
    assert(stack_pop() == 42);

    assert(stack_is_empty());
}
```

Les piles peuvent être implémentées avec des tableaux dynamiques ou des listes chaînées (voir plus bas).

21.12 Queues ou FIFO (*First In First Out*)

Les queues sont aussi des structures très similaires à des tableaux dynamiques mais elle ne permettent que les opérations suivantes :

- ajouter un élément à la queue (*push*) aussi nommé *enqueue* ;
- supprimer un élément au début de la queue (*shift*) aussi nommé *dequeue* ;
- tester si la queue est vide (*is_empty*) ;
- tester si la queue est pleine avec (*is_full*).

Les queues sont souvent utilisées lorsque des processus séquentiels ou parallèles s'échangent des tâches à traiter :

```
#include "queue.h"
#include <stdio.h>

void get_work(Queue *queue) {
    while (!feof(stdin)) {
        int n;
        if (scanf("%d", &n) == 1)
            queue_enqueue(n);
        scanf("%*[^\n]%[\n]");
    }
}

void process_work(Queue *queue) {
    while (!is_empty(queue)) {
        int n = queue_dequeue(queue);
        printf("%d est %s\n", n, n % 2 ? "impair" : "pair";
    }
}

int main() {
    Queue* queue;

    queue_init(&queue);
    get_work(queue);
    process_work(queue);
    queue_free(queue);
}
```

21.13 Performances

Les différentes structures de données ne sont pas toutes équivalentes en termes de performances. Il convient, selon l'application, d'opter pour la structure la plus adaptée, et par conséquent il est important de pouvoir comparer les différentes structures de données pour choisir la plus appropriée. Est-ce que les données doivent être maintenues triées ? Est-ce que la structure de donnée est utilisée comme une pile ou un tas ? Quelle est la structure de donnée avec le moin d'*overhead* pour les opérations de `push` ou `unshift` ?

L'indexation (*indexing*) est l'accès à une certaine valeur du tableau par exemple avec `a[k]`. Dans un tableau statique et dynamique l'accès se fait par pointeur depuis le début du tableau soit : `*((char*)a + sizeof(a[0]) * k)` qui est équivalant à `*(a + k)`. L'indexation par arithmétique de pointeur n'est pas possible avec les listes chaînées dont il faut parcourir chaque élément pour découvrir l'adresse du prochain élément :

```
int get(List *list) {
    List *el = list->head;
    for(int i = 0; i < k; i++)
        el = el.next;
```

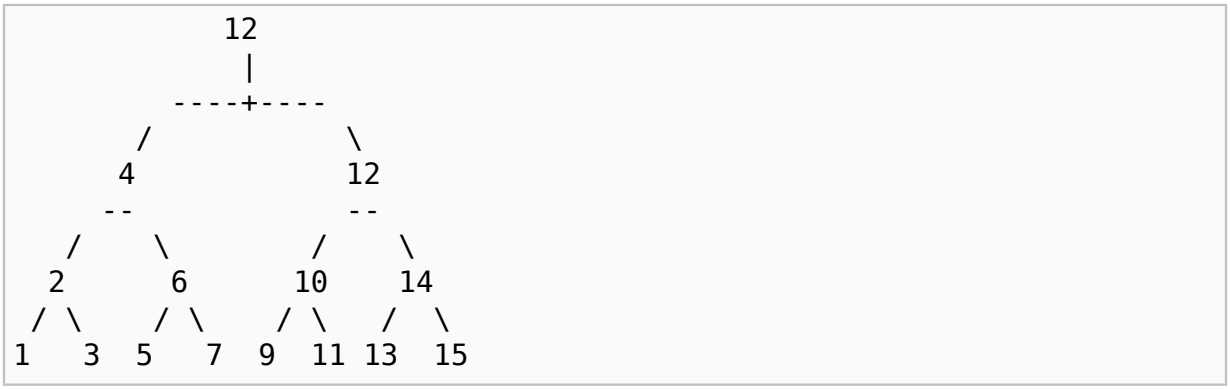
(suite sur la page suivante)

(suite de la page précédente)

```
    return el.value;
}
```

L'indexation d'une liste chaînée prends dans le cas le plus défavorable $O(n)$.

Les arbres binaires ont une structure qui permet naturellement la dichotomique. Chercher l'élément 5 prend 4 opérations : `12 -> 4 -> 6 -> 5`. L'indexation est ainsi possible en $O(logn)$.



Le tableau suivant résume les performances obtenues pour les différentes structures de données que nous avons vu dans ce chapitre :

Action	Tableau		Liste	Buffer	Arbre	Hash Map
‘ ‘	Statique	Dynamique	chaînée	circulaire		
Indexing	1	1	n	1	log n	1
Unshift/Shift	n	n	1	1	log n	n
Push/Pop	1	1 amorti	1	1	log n	1
Insert/Delete	n	n	1	n	log n	n
Search	n	n	n	n	log n	1
Sort	n log n	n log n	n log n	n log n	1	*n/a*

Chapitre 22

Avancé

Ce chapitre regroupe les sujets avancés dont la compréhension n'est pas requise pour le contrôle de connaissance.

22.1 Points de séquences

On appelle un point de séquence ou sequence point exprimé dans l'annexe C du standard C99 chaque élément de code dont l'exécution est garantie avant la séquence suivante. Ce qu'il est important de retenir c'est :

- L'appel d'une fonction est effectué après que tous ses arguments ont été évalués
- **La fin du premier opérande dans les opérations &&, ||, *?* et ,.**
  - Ceci permet de court-circuiter le calcul dans `a() && b() ``. La condition ``b()` n'est jamais évaluée si la condition `a()` est valide.
- Avant et après des actions associées à un formatage d'entrée sortie

L'opérateur d'assignation `=` n'est donc pas un point de séquence et l'exécution du code `(a = 2) + a + (a = 2)` est par conséquent indéterminée.

22.2 Complément sur les variables initialisées

Le fait de déclarer des variables dans en langage C implique que le logiciel doit réaliser l'initialisation de ces variables au tout début de son exécution. De fait, on peut remarquer deux choses. Il y a les variables initialisées à la valeur zéro et les variables initialisées à des valeurs différentes de zéro. Le compilateur regroupe en mémoire ces variables en deux catégories et ajoute un bout de code au début de votre application (qui est exécuté avant le *main*).

Ce code (que l'on n'a pas à écrire) effectue les opérations suivantes :

- mise à zéro du bloc mémoire contenant les variables ayant été déclarées avec une valeur d'initialisation à zéro
- recopie d'une zone mémoire contenant les valeurs initiales des variables ayant été déclarées avec une valeur d'initialisation différente de zéro vers la zone de ces mêmes variables.

Par ce fait, dès que l'exécution du logiciel est effectuée, on a, lors de l'exécution du *main*, des variables correctement initialisées.

22.3 Binutils

Les outils binaires (binutils) sont une collection de programmes installés avec un compilateur et permettant d'aider au développement et au débogage. Certains de ces outils sont très pratiques, mais nombreux sont les développeurs qui ne les connaissent pas.

nm Liste tous les symboles dans un fichier objet (binaire). Ce programme appliqué sur le programme hello world de l'introduction donne :

```
$ nm a.out
0000000000200dc8 d _DYNAMIC
0000000000200fb8 d _GLOBAL_OFFSET_TABLE_
00000000000006f0 R _IO_stdin_used
                 w _ITM_deregisterTMCloneTable
                 w _ITM_registerTMCloneTable

...

                 U __libc_start_main@@GLIBC_2.2.5
0000000000201010 D _edata
0000000000201018 B _end
00000000000006e4 T _fini
00000000000004f0 T _init
0000000000000540 T _start

...

000000000000064a T main
                 U printf@@GLIBC_2.2.5
00000000000005b0 t register_tm_clones
```

On observe notamment que la fonction `printf` est en provenance de la bibliothèque GLIBC 2.2.5, et qu'il y a une fonction `main`.

strings Liste toutes les chaînes de caractères imprimables dans un fichier binaire. On observe tous les symboles de débug qui sont par défaut intégrés au fichier exécutable. On lit également la chaîne de caractère `hello, world`. Attention donc à ne pas laisser les éventuels mots de passes ou numéro de licence en clair dans un fichier binaire.

```
$ strings a.out
/lib64/ld-linux-x86-64.so.2
libc.so.6
printf

...
```

(suite sur la page suivante)

(suite de la page précédente)

```
AUATL
[]A\A]A^A_
hello, world
;*3$"
GCC: (Ubuntu 7.4.0-1ubuntu1~18.04.1) 7.4.0

...

_IO_stdin_used
__libc_csu_init
__bss_start
main
__TMC_END__
_ITM_registerTMCloneTable
__cxa_finalize@@GLIBC_2.2.5
.symtab
.strtab

...

.data
.bss
.comment
```

size Liste la taille des segments mémoires utilisés. Ici le programme représente 1517 bytes, les données initialisées 8 bytes, les données variables 600 bytes, soit une somme décimale de 2125 bytes ou 84d bytes.

```
$ size a.out
text    data     bss     dec     hex filename
1517     600       8    2125     84d a.out
```

22.4 Format Q

Le format Q est une notation en virgule fixe dans laquelle le format d'un nombre est représenté par la lettre **Q** suivie de deux nombres :

1. Le nombre de bits entiers
2. Le nombre de bits fractionnaires

Ainsi, un registre 16 bits contenant un nombre allant de +0.999 à -1.0 s'exprimera **Q1.15** soit 1 + 15 valant 16 bits.

Pour exprimer la valeur pi (3.1415...) il faudra au minimum 3 bits pour représenter la partie entière, car le bit de signe doit rester à zéro. Le format sur 16 bits sera ainsi **Q4.12**.

La construction de ce nombre est facile :

1. Prendre le nombre réel

2. Le multiplier par 2 à la puissance du nombre de bits
3. Prendre la partie entière

```
1.    3.1415926535
2.    2**12 * 3.1415926535 = 12867.963508736
3.    12867
```

Pour convertir un nombre **Q4.12** en sa valeur réelle il faut :

1. Prendre le nombre encodé en **Q4.12**
2. Diviser sa valeur 2 à la puissance du nombre de bits

```
1.    12867
2.    12867 / 2**12 = 3.141357421875
```

On note une perte de précision puisqu'il n'est pas possible d'encoder un tel nombre dans seulement 16 bits. L'incrément positif minimal serait : $1/2^12 = 0.00024$. Il convient alors d'arrondir le nombre à la troisième décimale soit 3.141.

Les opérations arithmétiques sont possibles facilement entre des nombres de même types.

22.4.1 Addition

L'addition peut se faire avec ou sans saturation :

```c
typedef int16_t Q;
typedef Q Q12;

Q q_add(Q a, Q b) {
    return a + b;
}

Q q_add_sat(Q a, Q b) {
    int32_t res = (int32_t)a + (int32_t)b;
    res = res > 0x7FFF ? 0x7FFF : res
    res = res < -1 * 0x8000 ? -1 * 0x8000 : res;
    return (Q)res;
}
```

22.4.2 Multiplication

Soit deux nombres 0.9 et 3.141 :

```
┌─┬─┬─┬─┬─┬─┬─┬─┐ ┌─┬─┬─┬─┬─┬─┬─┬─┐
|0|0|0|0|1|1|1|0| |0|1|1|0|0|1|1|0| Q4.12 (0.9) 3686
└─┴─┴─┴─┴─┴─┴─┴─┘ └─┴─┴─┴─┴─┴─┴─┴─┘
```

(suite sur la page suivante)

(suite de la page précédente)

```
┌─┬─┬─┬─┬─┬─┬─┬─┐ ┌─┬─┬─┬─┬─┬─┬─┬─┐
│0│0│1│1│0│0│1│0│ │0│1│0│0│0│0│1│1│ Q4.12 (3.141) 12867
└─┴─┴─┴─┴─┴─┴─┴─┘ └─┴─┴─┴─┴─┴─┴─┴─┘
```

Multiplier ces deux valeurs revient à une multiplication sur 2 fois la taille. Le résultat doit être obtenu sur 32-bits sachant que les nombre **Q** s'additionnent comme **Q4.12** x **Q4.12** donnera **Q8.24**.

On voit immédiatement que la partie entière vaut 2, donc 90% de 3.14 donnera une valeur en dessous de 3. Pour reconstruire une valeur **Q8.8** il convient de supprimer les 16-bits de poids faible

```
3686 * 12867 = 47227762

┌─┬─┬─┬─┬─┬─┬─┬─┐ ┌─┬─┬─┬─┬─┬─┬─┬─┐ ┌─┬─┬─┬─┬─┬─┬─┬─┐ ┌─┬─┬─┬─┬─┬─┬─┬─┐
│0│0│0│0│0│0│1│0│ │1│1│0│1│0│0│0│0│ │1│0│1│0│0│0│1│1│ │0│1│1│1│0│0│1│0│␣
↪Q8.24
└─┴─┴─┴─┴─┴─┴─┴─┘ └─┴─┴─┴─┴─┴─┴─┴─┘ └─┴─┴─┴─┴─┴─┴─┴─┘ └─┴─┴─┴─┴─┴─┴─┴─┘

┌─┬─┬─┬─┬─┬─┬─┬─┐ ┌─┬─┬─┬─┬─┬─┬─┬─┐
│0│0│0│0│0│0│1│0│ │1│1│0│1│0│0│0│0│ Q8.8
└─┴─┴─┴─┴─┴─┴─┴─┘ └─┴─┴─┴─┴─┴─┴─┴─┘
```

```c
inline Q q_sat(int32_t x) {
    x = x > 0x7FFF ? 0x7FFF : x
    x = x < -1 * 0x8000 ? -1 * 0x8000 : x;
    return (Q)x;
}

inline int16_t q_mul(int16_t a, int16_t b, char q)
{
    int32_t c = (int32_t)a * (int32_t)b;
    c += 1 << (q - 1);
    return sat(c >> q);
}

inline int16_t q12_mul(int16_t a, int16_t b)
{
    return q_mul(a, b, 12);
}
```

22.5 Mémoire partagée

Nous le verrons plus loin au chapitre sur la MMU, mais la mémoire d'un processus mémoire (programme) ne peut pas être accedée par un autre programme. Le système d'exploitation l'en empêche.

Lorsque l'on souhaite communiquer entre plusieurs programmes, il est possible d'utiliser différentes méthodes :

— les flux (fichiers, stdin, stdout...)
— la mémoire partagée
— les sockets

Vous avez déjà vu les flux au chapitre précédant, et les sockets ne font pas partie de ce cours d'introduction.

Notons que la mémoire partagée est un mécanisme propre à chaque système d'exploitation. Sous POSIX elle est normalisée et donc un programme compatible POSIX et utilisant la mémoire partagée pourra fonctionner sous Linux, WSL ou macOS, mais pas sous Windows.

C'est principalement l'appel système `mmap` qui est utilisé. Il permet de mapper ou démapper des fichiers ou des périphériques dans la mémoire.

```
void *mmap(
    void *addr,
    size_t length, // Taille en byte de l'espace mémoire
    int prot,      // Protection d'accès (lecture, écriture,␣
↪exécution)
    int flags,     // Attributs (partagé, privé, anonyme...)
    int fd,
    int offset
);
```

Voici un exemple permettant de réserver un espace partagé en écriture et en lecture entre deux processus :

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/mman.h>

void* create_shared_memory(size_t size) {
    // Accessible en lecture et écriture
    int protection = PROT_READ | PROT_WRITE;

    // D'autres processus peuvent accéder à cet espace
    // lequel est anonyme
    // so only this process and its children will be able to use it:
    int visibility = MAP_SHARED | MAP_ANONYMOUS;

    // The remaining parameters to `mmap()` are not important for␣
↪this use case,
```

(suite sur la page suivante)

(suite de la page précédente)

```
    // but the manpage for `mmap` explains their purpose.
    return mmap(NULL, size, protection, visibility, -1, 0);
}
```

22.5.1 File memory mapping

Traditionnellement lorsque l'on souhaite travailler sur un fichier, il convient de l'ouvrir avec `fopen` et de lire son contenu. Lorsque cela est nécessaire, ce fichier est copié en mémoire :

```
FILE *fp = fopen("foo", "r");
fseek(fp, 0, SEEK_END);
int filesize = ftell(fp);
fseek(fp, 0, SEEK_SET);
char *file = malloc(filesize);
fread(file, filesize, sizeof(char), fp);
fclose(fp);
```

Cette copie n'est pas nécessairement nécessaire. Une approche **POSIX**, qui n'est donc pas couverte par le standard **C99** consiste à lier le fichier dans un espace mémoire partagé.

Ceci nécessite l'utilisation de fonctions bas niveau.

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/mman.h>

int main() {
    int fd = open("foo.txt", O_RDWR, 0600);
    char *addr = mmap(NULL, 100, PROT_READ | PROT_WRITE, MAP_SHARED,
↪ fd, 0);
    printf("Espace mappé à %p\n", addr);
    printf("Premiers caractères du fichiers : %.*s...\n", 20, addr);
}
```

Les avantages de cette méthode sont :

- pas nécessaire de copier l'intégralité du fichier en mémoire ;
- possibilité de partager le même fichier ouvert entre plusieurs processus ;
- possibilité laissée au système d'exploitation d'utiliser la RAM ou non si les ressources mémoires deviennent tendues.

22.6 Collecteur de déchets (*garbage collector*)

Le C est un langage primitif qui ne gère pas automatiquement la libération des ressources allouées dynamiquement. L'exemple suivant est évocateur :

```
int* get_number() {
    int *num = malloc(sizeof(int));
    *num = rand();
}

int main() {
    for (int i = 0; i < 100; i++) {
        printf("%d\n", *get_number());
    }
}
```

La fonction `get_number` alloue dynamiquement un espace de la taille d'un entier et lui assigne une valeur aléatoire. Dans le programme principal, l'adresse retournée est déréférencée pour être affichée sur la sortie standard.

A la fin de l'exécution de la boucle for, une centaine d'espaces mémoire sont maintenant dans les limbes. Comme le pointeur retourné n'a jamais été mémorisé, il n'est plus possible de libérer cet espace mémoire avec `free`.

On dit que le programme à une fuite mémoire. En admettant que ce programme reste résidant en mémoire, il peut arriver un moment où le programme peut aller jusqu'à utiliser toute la RAM disponible. Dans ce cas, il est probable que `malloc` retourne `NULL` et qu'une erreur de segmentaiton apparaisse lors du `printf`.

Allons plus loin dans notre exemple et considérons le code suivant :

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

int foo(int *new_value) {
    static int *values[10] = { NULL };
    static int count = 0;

    if (rand() % 5 && count < sizeof(values) / sizeof(*values) - 1)
↪{
        values[count++] = new_value;
    }

    if (count > 0)
        printf("Foo aime %d\n", *values[rand() % count]);
}

int bar(int *new_value) {
    static int *values[10] = { NULL };
    static int count = 0;
```

(suite sur la page suivante)

(suite de la page précédente)

```
    if (rand() % 5 && count < sizeof(values) / sizeof(*values) - 1)
↪{
        values[count++] = new_value;
    }

    if (count > 0)
        printf("Bar aime %d\n", *values[rand() % count]);
}

int* get_number() {
    int *number = malloc(sizeof(int));
    *number = rand() % 1000;
    return number;
}

int main() {
    int experiment_iterations = 10;
    for (int i = 0; i < experiment_iterations; i++) {
        int *num = get_number();
        foo(num);
        bar(num);
        #if 0 // ...
            free(num) ??
        #endif
    };
}
```

La fonction `get_number` alloue dynamiquement un espace mémoire et assigne un nombre aléatoire. Les fonctions `foo` et `bar` reçoivent en paramètre un pointeur sur un entier. Chacune à le choix de mémoriser ce pointeur et de clamer sur `stdout` qu'elle aime un des nombre mémorisés.

Au niveau du `#if 0` dans la fonction `main`, il est impossible de savoir si l'adresse pointée par `num` est encore utilisée ou non. Il se peut que `foo` et `bar` utilisent cet espace mémoire, comme il se peut qu'aucun des deux ne l'utilise.

Comment peut-on savoir si il est possible de libérer ou non `num` ?

Une solution courament utilsée en C++ s'appelle un *smart pointer*. Il s'agit d'un pointeur qui contient en plus de l'adresse de la valeur, le nombre de références utilisées. De cette manière il est possible en tout temps de savoir si le pointeur est référencé quelque part. Dans le cas ou le nombre de référence tombe à zéro, il est possible de libérer la ressource.

Dans un certain nombre de langage de programmation comme Python ou Java, il existe un mécanisme automatique nommé *Garbage Collector* et qui, périodiquement, fait un tour de toutes les allocations dynamique pour savoir si elle sont encore référencées ou non. Le cas échéant, le *gc* décide libérer la ressource mémoire. De cette manière il n'est plus nécessaire de faire la chasse aux ressources allouées.

En revanche en C, il n'existe aucun mécanisme aussi sophistiqués alors prenez garde à

bien libérer les ressources utilisée et à éviter d'écrire des fonctions qui allouent du contenu mémoire dynamiquement.

Chapitre 23

Pièges

Ce chapitre traite des pièges les plus courants dans lesquels l'apprenti programmeur ne manquera pas de tomber. Le C est un langage puissant, mais potentiellement dangereux, car il permet d'opérer à très bas niveau, ce qui peut occasionner des bogues retors à l'exécution.

23.1 Préprocesseur

Rappelons-le, les macros sont des simples remplacements de chaînes de caractères intervenant avant la compilation.

23.1.1 Macro avec paramètre

Le préprocesseur interprète une macro avec paramètres que si la parenthèse ouvrante suit directement et sans espace le nom de la macro. Ainsi considérant cet exemple :

```
#define f (x) ((x) + 1)

int u = f(1)
```

Le préprocesseur génère ceci et le compilateur retournera une erreur du type `identification x non déclaré`.

```
int u = (x) ((x) + 1)(1)
```

Observations :

- Ne jamais mettre d'espace entre le nom d'une macro et ses paramètres.
- Être toujours prêt à mettre en doute le code généré par une macro.

23.1.2 Paramètres de macro non protégés

On ne le répètera jamais assez : une macro est un remplacement de chaîne effectué par le préprocesseur. Donc écrire `#define m(x) x * 2` et `m(2 + 5)` sera remplacé en `2 + 5 * 2`.

Quel sera le problème dans le cas suivant ?

```
#define ABS(x) x >= 0 ? x: -x

int foo(void) {
    return ABS(5 - 8);
}
```

Plus difficile, quel serait le problème ici :

```
#define ERROR(str) printf("Erreur: %s\r\n", str); log(str);

if (y < 0)
    ERROR("Zero division");
else
    x = x / y;
```

Observations :

— Toujours protéger les paramètres des macros avec des parenthèses

```
#define ABS(x) ((x) >= 0 ? (x): -(x))
```

— Toujours protéger une macro à plusieurs instructions par une boucle vide :

```
#define ERROR(str) do { \
    printf("Erreur: %s\r\n", str); \
    log(str); \
} while (0)
```

23.1.3 Pré/Post incrémentation avec une macro

On pourrait se dire qu'avec toutes les précautions prises, il n'y aura plus d'ennuis possibles. Or, les post/pré incréments peuvent encore poser problème.

```
#define ABS(x) ((x) >= 0 ? (x) : -(x))

return ABS(x++)
```

On peut constater que x sera post-incrémenté deux fois au lieu d'une :

```
#define ABS(x) ((x) >= 0 ? (x) : -(x))

return ((x++) >= 0 ? (x++) : -(x++))
```

Observations :
— Éviter l'utilisation de la pre/post incrémentation/décrémentation dans l'appel de macros.

23.2 Erreurs de syntaxe

23.2.1 Confusion = et ==

L'erreur est si vite commise, mais souvent fatale :

```
if (c = 'o') {

}
```

L'effet contre-intuitif est que le test retourne toujours VRAI, car `'o' > 0`. Ajoutons que la valeur de `c` est modifié au passage.

Observations :
— Pour éviter toute ambiguïté, éviter les affectations dans les structures conditionnelles.

23.2.2 Confusion & et &&

Confondre le ET logique et le ET binaire est courant. Dans l'exemple suivant, le `if` n'est jamais exécuté :

```
int a = 0xA;
int b = 0x5;

if(a & b) {

}
```

23.2.3 Écriture déroutante

Selon la table de précédences on aura `i--` calculé en premier suivi de `- -j` :

```
k = i----j;
```

Observations :
— Éviter les formes ambigües d'écriture
— Favoriser la précédence explicite en utilisant des parenthèses
— Séparez vos opérations par des espaces pour plus de lisibilité : `k = i-- - -j`

23.2.4 Point virgule

L'erreur typique suivante est arrivée à tout programmeur débutant. Le `;` placé après le test `if` agis comme une instruction nulle si bien que la fusée sera lancée à tous les coups :

```
if (countdown == 0);
  launch_rocket();
```

Le même type d'erreur peut apparaître avec une boucle, ici causant une boucle infinie :

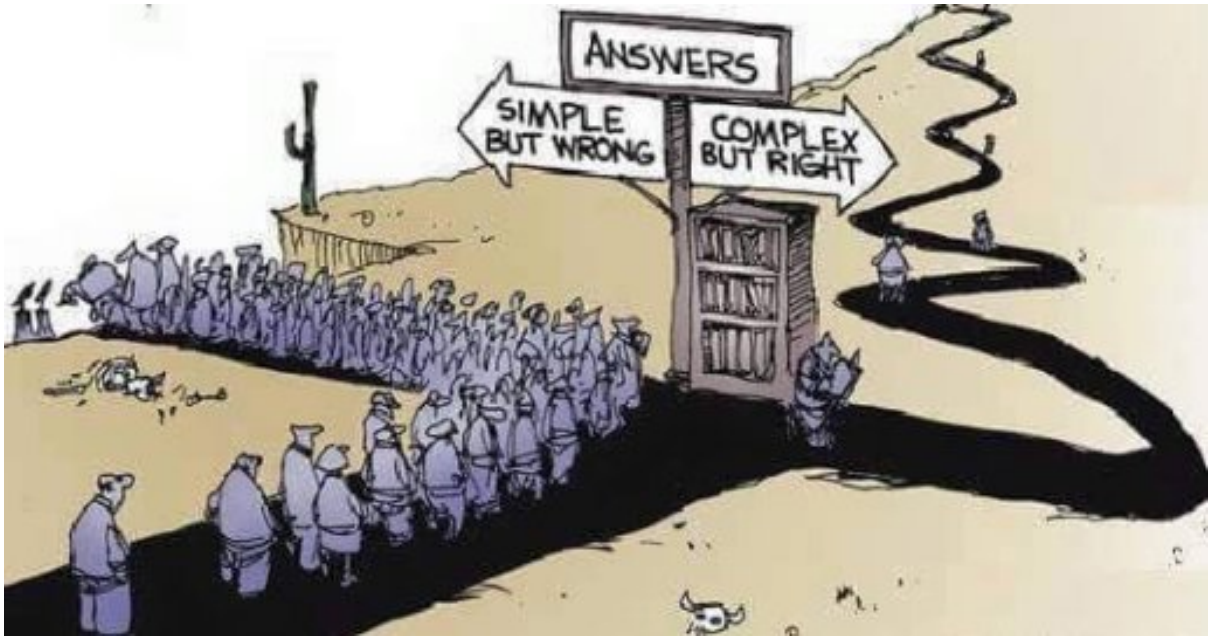
```
while(i > 0);
{
    i--;
}
```

Chapitre 24

Philosophie

La philosophie d'un bon développeur repose sur plusieurs principes de programmation relevant majoritairement du bon sens de l'ingénieur, les vaudois l'appelant parfois : **le bon sens paysan** comme l'aurait sans doute confirmé feu Jean Villard dit Gilles.

24.1 Rasoir d'Ockham



Le rasoir d'Ockham expose en substance que les multiples ne doivent pas être utilisés sans nécessité. C'est un principe d'économie, de simplicité et de parcimonie. Il peut être résumé par la devise Shadok : "Pourquoi faire simple quand on peut faire compliqué ?"

En philosophie un rasoir est un principe qui permet de *raser* des explications improbables d'un phénomène. Ce principe tient son nom de Guillaume d'Ockham (XIVe siècle) alors qu'il date probablement d'Empédocle ( ) vers 450 av. J.-C.

Il trouve admirablement bien sa place en programmation où le programmeur ne peut conserver une vue d'ensemble sur un logiciel qui est par nature invisible à ses yeux. Seuls

la simplicité et l'art de la conception logicielle sauvent un développeur de la noyade, car un programme peut rester simple quelque soit sa taille si chaque strate de conception reste évidente et simple à comprendre pour celui qui chercherait à contribuer au projet d'autrui.

24.2 Principes de programmation

24.2.1 DRY

Ne vous répétez pas (*Don't Repeat Yourself*) ! Je répète, **ne vous répétez pas** ! Il s'agit d'une philosophie de développement logiciel évitant la redondance de code. L'excellent livre The Pragmatic Programmer de Andrew Hunt et David Thomas décrit cette philosophie en ces termes :

> Dans un système, toute connaissance doit avoir une représentation unique, non ambiguë, faisant autorité.

En d'autres termes, le programmeur doit avoir sans cesse à l'esprit une sonnette d'alarme prête à vrombir lorsque qu'il presse machinalement `CTRL` (⌘) + `C` suivi de `CTRL` (⌘) + `V`. Dupliquer du code et quelque soit l'envergure de texte concerné est **toujours** une mauvaise pratique, car c'est le plus souvent le signe d'un code smell qui indique que le code peut être simplifié et optimisé.

24.2.2 KISS

Keep it simple, stupid est une ligne directrice de conception qui encourage la simplicité d'un développement. Il est similaire au rasoir d'Ockham, mais grandement plus commun en informatique. Énoncé par Eric Steven Raymond puis par le Zen de Python un programme ne doit faire qu'une chose, et une chose simple. C'est une philosophie grandement respectée dans l'univers Unix/Linux. Chaque programme de base du *shell* (`ls`, `cat`, `echo`, `grep`, ...) ne fait qu'une tâche simple, le nom est court et simple à retenir.

24.2.3 YAGNI

YAGNI est un anglicisme de *you ain't gonna need it* qui peut être traduit par : vous n'en aurez pas besoin. C'est un principe très connu en développent Agile XP (Extreme Programming) qui déclare qu'un développeur logiciel ne devrait pas implémenter une fonctionnalité à un logiciel tant que celle-ci n'est pas absolument nécessaire.

Ce principe combat le biais du développeur à vouloir sans cesse démarrer de nombreux chantiers sans se focaliser sur l'essentiel strictement nécessaire d'un programme et permettant de respecter le cahier des charges convenu avec le partenaire/client.

24.2.4 SSOT

Ce principe tient son acronyme de single source of truth. Il adresse principalement un défaut de conception relatif aux métadonnées que peuvent être les paramètres d'un algorithme, le modèle d'une base de données ou la méthode usitée d'un programme à collecter des données.

Un programme qui respecte ce principe évite la duplication des données. Des défauts courants de conception sont :

- Indiquer le nom d'un fichier source dans le fichier source
- Stocker la même image, le même document dans différents formats
- Stocker dans une base de données le nom *Doe*, prénom *John* ainsi que le nom complet *John Doe*
- Avoir un commentaire C du type : `int height = 206; // Size of Hafþór Júlíus Björnsson which is 205 cm` (deux sources de vérités contradictoires)
- Conserver une copie des mêmes données sous des formats différents (un tableau de données brutes et un tableau des mêmes données, mais triées)

24.3 Zen de Python

Python est un langage de programmation qui devient très populaire, il est certes moins performant que C, mais il se veut être de très haut niveau.

Le Zen de Python est un ensemble de 19 principes publiés en 1999 par Tim Peters. Largement accepté par la communauté de développeurs et il est connu sous le nom de **PEP 20**.

Voici le texte original anglais :

```
Beautiful is better than ugly.
Explicit is better than implicit.
Simple is better than complex.
Complex is better than complicated.
Flat is better than nested.
Sparse is better than dense.
Readability counts.
Special cases aren't special enough to break the rules.
Although practicality beats purity.
Errors should never pass silently.
Unless explicitly silenced.
In the face of ambiguity, refuse the temptation to guess.
There should be one—and preferably only one—obvious way to do it.
Although that way may not be obvious at first unless you're Dutch.
Now is better than never.
Although never is often better than right now.[n 1]
If the implementation is hard to explain, it's a bad idea.
If the implementation is easy to explain, it may be a good idea.
Namespaces are one honking great idea—let's do more of those!
```

24.4 The code taste

Dans une conférence TED en 2016, le créateur de Linux, Linus Torvald évoqua un principe nommé *code taste* traduisible par *avoir du goût pour le code*.

Il évoqua l'exemple C suivant et demanda à l'auditoire si ce code est de bon goût :

```c
void remove_list_entry(List* list, Entry* entry)
{
    Entry* prev = NULL;
    Entry* walk = list->head;

    while (walk != entry) {
        prev = walk;
        walk = walk->next;
    }

    if (!prev)
        list->head = entry->next;
    else
        prev->next = entry->next;
}
```

Il répondit que ce code est de mauvais goût, qu'il est *vilain* et *moche*, car ce test placé après la boucle `while` jure avec le reste du code et que parce que ce code semble laid, il doit y avoir une meilleure implémentation de meilleur goût. On dit dans ce cas de figure que le code *sent*, ce test est de trop, et il doit y avoir un moyen d'éviter de traiter un cas particulier en utilisant un algorithme meilleur.

Enlever un élément d'une liste chaînée nécessite de traiter deux cas :

- Si l'élément est au début de la liste, il faut modifier `head`
- Sinon il faut modifier l'élément précédent `prev->next`

Après avoir longuement questionné l'auditoire, il présente cette nouvelle implémentation :

```c
void remove_list_entry(List* list, Entry* entry)
{
    Entry** indirect = &head;

    while ((*indirect) != entry)
        indirect = &(*indirect)->next;

    *indirect = entry->next;
}
```

La fonction originale de 10 lignes de code a été réduite à 4 lignes et bien que le nombre de lignes compte moins que la lisibilité du code, cette nouvelle implémentation élimine le traitement des cas d'exception en utilisant un adressage indirect beaucoup plus élégant.

Un autre exemple similaire et plus simple à comprendre est présenté par Brian Barto sur un article publié sur Medium. Il donne l'exemple de l'initialisation à zéro de la bordure d'un tableau bidimensionnel :

```
for (size_t row = 0; row < GRID_SIZE; ++row)
{
    for (size_t col = 0; col < GRID_SIZE; ++col)
    {
        if (row == 0)
            grid[row][col] = 0; // Top Edge

        if (col == 0)
            grid[row][col] = 0; // Left Edge

        if (col == GRID_SIZE - 1)
            grid[row][col] = 0; // Right Edge

        if (row == GRID_SIZE - 1)
            grid[row][col] = 0; // Bottom Edge
    }
}
```

On constate plusieurs fautes de goût :

- `GRID_SIZE` pourrait être différent de la réelle taille de `grid`
- Les valeurs d'initialisation sont dupliquées
- La complexité de l'algorithme est de $O(n^2)$ alors que l'on ne s'intéresse qu'à la bordure du tableau.

Voici une solution plus élégante :

```
const size_t length = sizeof(grid[0]) / sizeof(grid[0][0]);
const int init = 0;

// Edges initialisation
for (size_t i = 0; i < length; i++)
{
    grid[i][0] = grid[0][i] = init; // Top and Left
    grid[length - 1][i] = grid[i][length - 1] = init; // Bottom and␣
 ↪Right
}
```

24.5 L'odeur du code

Un code *sent* si certains indicateurs sont au rouge. On appelle ces indicateurs des anti-patterns. Voici quelques indicateurs les plus courants :

- **Mastodonte** Une fonction est plus longue qu'un écran de haut (~50 lignes)
- Un fichier est plus long que **1000 lignes**.
- **Ligne Dieu**, une ligne beaucoup trop longue et *de facto* illisible.
- **Une fonction à plus de trois paramètres**

```
void make_coffee(int size, int mode, int mouture, int cup_
↪size,
    bool with_milk, bool cow_milk, int number_of_sugars);
```

— **Copier coller**, du code est dupliqué
— **Les commentaires expliquent le comment du code et non le pourquoi**

```
// Additionne une constante avec une autre pour ensuite l
↪'utiliser
double u = (a + cst);
u /= 1.11123445143; // division par une constante␣
↪inférieure à 2
```

— **Arbre de Noël, plus de deux structures de contrôles sont impliquées**

```
if (a > 2) {
    if (b < 8) {
        if (c ==12) {
            if (d == 0) {
                exception(a, b, c, d);
            }
        }
    }
}
```

— Usage de **goto**

```
loop:
    i +=1;
    if (i > 100)
        goto end;
happy:
    happy();
    if (j > 10):
        goto sad;
sad:
    sad();
    if (k < 50):
        goto happy;
end:
```

— **Plusieurs variables avec des noms très similaires**

```
int advice = 11;
int advise = 12;
```

— **Action à distance** par l'emploi immodéré de variables globales
— **Ancre de bateau**, un composant inutilisé, mais gardé dans le logiciel pour des raisons politiques (YAGNI)
— **Cyclomatisme aigu**, quand trop de structures de contrôles sont nécessaires pour traiter un problème apparemment simple
— **Attente active, une boucle qui ne contient qu'une instruction de test, attendant la**

```
while (true) {
    if (finished) break;
}
```

— **Objet divin** quand un composant logiciel assure trop de fonctions essentielles (KISS)
— **Coulée de lave** lorsqu'un code immature est mis en production
— **Chirurgie au fusil de chasse** quand l'ajout d'une fonctionnalité logicielle demande des changements multiples et disparates dans le code (Shotgun surgery).

Annexe A

Visual Studio Code

Visual Studio Code est un éditeur gratuit open-source développé par Microsoft. Il gagne en popularité et réuni les avantages d'être léger, multi-platforme et multi-langage.

Pour pouvoir l'utilser avec votre compilateur et écrire du C, il faut une configuration minimale.

Visual Studio Code n'a pas la notion de **projet** mais d'espace de travail **workspace**. Un espace de travail est simplement un répertoire. A l'intérieur de ce répertoire on y trouvera :

```
.
├── .vscode
│   └── launch.json
└── main.c
```

Visual Studio Code peut en général générer automatiquement le fichier `.vscode/launch.json` qui contient tout ce qu'il faut pour compiler et exécuter le programme :

```
{
    "version": "0.2.0",
    "configurations": [
        {
            "name": "gcc",
            "type": "cppdbg",
            "request": "launch",
            "program":␣
↪"${fileDirname}\\${fileBasenameNoExtension}.exe",
            "args": [],
            "stopAtEntry": false,
            "cwd": "${workspaceFolder}",
            "environment": [],
            "externalConsole": false,
            "MIMode": "gdb",
            "miDebuggerPath":␣
↪"C:\\ProgramData\\chocolatey\\lib\\mingw\\tools\\install\\mingw64\\bin\\gdb.ex
↪
```

(suite sur la page suivante)

(suite de la page précédente)

```
            "setupCommands": [
                {
                    "description": "Enable pretty-printing for gdb",
                    "text": "-enable-pretty-printing",
                    "ignoreFailures": true
                }
            ],
            "preLaunchTask": "gcc.exe build active file"
        }
    ]
}
```

```
{
    "version": "2.0.0",
    "tasks": [
        {
            "type": "shell",
            "label": "gcc.exe build active file",
            "command":␣
↪"C:\\ProgramData\\chocolatey\\lib\\mingw\\tools\\install\\mingw64\\bin\\gcc.ex
↪
            "args": [
                "-g",
                "${file}",
                "-o",
                "${fileDirname}\\${fileBasenameNoExtension}.exe"
            ],
            "options": {
                "cwd":␣
↪"C:\\ProgramData\\chocolatey\\lib\\mingw\\tools\\install\\mingw64\\bin"
            },
            "problemMatcher": [
                "$gcc"
            ]
        }
    ]
}
```

Annexe B

Grammaire C

Yacc (*Yet Another COmpiler-Compiler*) est un logiciel utilisé pour écrire des analyseur syntaxique de code. Il prend en entrée une grammaire.

Parce que les informaticiens ont de l'humour, Yacc à son pendant GNU Bison plus récent (1985) mais toujours activement développé.

Voici à titre d'information la définition formelle du langage C99 :

```
%token IDENTIFIER CONSTANT STRING_LITERAL SIZEOF
%token PTR_OP INC_OP DEC_OP LEFT_OP RIGHT_OP LE_OP GE_OP EQ_OP NE_OP
%token AND_OP OR_OP MUL_ASSIGN DIV_ASSIGN MOD_ASSIGN ADD_ASSIGN
%token SUB_ASSIGN LEFT_ASSIGN RIGHT_ASSIGN AND_ASSIGN
%token XOR_ASSIGN OR_ASSIGN TYPE_NAME

%token TYPEDEF EXTERN STATIC AUTO REGISTER INLINE RESTRICT
%token CHAR SHORT INT LONG SIGNED UNSIGNED FLOAT DOUBLE CONST␣
↪VOLATILE VOID
%token BOOL COMPLEX IMAGINARY
%token STRUCT UNION ENUM ELLIPSIS

%token CASE DEFAULT IF ELSE SWITCH WHILE DO FOR GOTO CONTINUE BREAK␣
↪RETURN

%start translation_unit
%%

primary_expression
        : IDENTIFIER
        | CONSTANT
        | STRING_LITERAL
        | '(' expression ')'
        ;

postfix_expression
        : primary_expression
        | postfix_expression '[' expression ']'
```

(suite sur la page suivante)

(suite de la page précédente)

```
        | postfix_expression '(' ')'
        | postfix_expression '(' argument_expression_list ')'
        | postfix_expression '.' IDENTIFIER
        | postfix_expression PTR_OP IDENTIFIER
        | postfix_expression INC_OP
        | postfix_expression DEC_OP
        | '(' type_name ')' '{' initializer_list '}'
        | '(' type_name ')' '{' initializer_list ',' '}'
        ;

argument_expression_list
        : assignment_expression
        | argument_expression_list ',' assignment_expression
        ;

unary_expression
        : postfix_expression
        | INC_OP unary_expression
        | DEC_OP unary_expression
        | unary_operator cast_expression
        | SIZEOF unary_expression
        | SIZEOF '(' type_name ')'
        ;

unary_operator
        : '&'
        | '*'
        | '+'
        | '-'
        | '~'
        | '!'
        ;

cast_expression
        : unary_expression
        | '(' type_name ')' cast_expression
        ;

multiplicative_expression
        : cast_expression
        | multiplicative_expression '*' cast_expression
        | multiplicative_expression '/' cast_expression
        | multiplicative_expression '%' cast_expression
        ;

additive_expression
        : multiplicative_expression
        | additive_expression '+' multiplicative_expression
        | additive_expression '-' multiplicative_expression
```

(suite sur la page suivante)

(suite de la page précédente)

```
        ;

shift_expression
        : additive_expression
        | shift_expression LEFT_OP additive_expression
        | shift_expression RIGHT_OP additive_expression
        ;

relational_expression
        : shift_expression
        | relational_expression '<' shift_expression
        | relational_expression '>' shift_expression
        | relational_expression LE_OP shift_expression
        | relational_expression GE_OP shift_expression
        ;

equality_expression
        : relational_expression
        | equality_expression EQ_OP relational_expression
        | equality_expression NE_OP relational_expression
        ;

and_expression
        : equality_expression
        | and_expression '&' equality_expression
        ;

exclusive_or_expression
        : and_expression
        | exclusive_or_expression '^' and_expression
        ;

inclusive_or_expression
        : exclusive_or_expression
        | inclusive_or_expression '|' exclusive_or_expression
        ;

logical_and_expression
        : inclusive_or_expression
        | logical_and_expression AND_OP inclusive_or_expression
        ;

logical_or_expression
        : logical_and_expression
        | logical_or_expression OR_OP logical_and_expression
        ;

conditional_expression
        : logical_or_expression
```

(suite sur la page suivante)

(suite de la page précédente)

```
        | logical_or_expression '?' expression ':' conditional_
↪expression
        ;

assignment_expression
        : conditional_expression
        | unary_expression assignment_operator assignment_expression
        ;

assignment_operator
        : '='
        | MUL_ASSIGN
        | DIV_ASSIGN
        | MOD_ASSIGN
        | ADD_ASSIGN
        | SUB_ASSIGN
        | LEFT_ASSIGN
        | RIGHT_ASSIGN
        | AND_ASSIGN
        | XOR_ASSIGN
        | OR_ASSIGN
        ;

expression
        : assignment_expression
        | expression ',' assignment_expression
        ;

constant_expression
        : conditional_expression
        ;

declaration
        : declaration_specifiers ';'
        | declaration_specifiers init_declarator_list ';'
        ;

declaration_specifiers
        : storage_class_specifier
        | storage_class_specifier declaration_specifiers
        | type_specifier
        | type_specifier declaration_specifiers
        | type_qualifier
        | type_qualifier declaration_specifiers
        | function_specifier
        | function_specifier declaration_specifiers
        ;

init_declarator_list
```

(suite sur la page suivante)

(suite de la page précédente)

```
        : init_declarator
        | init_declarator_list ',' init_declarator
        ;

init_declarator
        : declarator
        | declarator '=' initializer
        ;

storage_class_specifier
        : TYPEDEF
        | EXTERN
        | STATIC
        | AUTO
        | REGISTER
        ;

type_specifier
        : VOID
        | CHAR
        | SHORT
        | INT
        | LONG
        | FLOAT
        | DOUBLE
        | SIGNED
        | UNSIGNED
        | BOOL
        | COMPLEX
        | IMAGINARY
        | struct_or_union_specifier
        | enum_specifier
        | TYPE_NAME
        ;

struct_or_union_specifier
        : struct_or_union IDENTIFIER '{' struct_declaration_list '}'
        | struct_or_union '{' struct_declaration_list '}'
        | struct_or_union IDENTIFIER
        ;

struct_or_union
        : STRUCT
        | UNION
        ;

struct_declaration_list
        : struct_declaration
        | struct_declaration_list struct_declaration
```

(suite sur la page suivante)

(suite de la page précédente)

```
        ;

struct_declaration
        : specifier_qualifier_list struct_declarator_list ';'
        ;

specifier_qualifier_list
        : type_specifier specifier_qualifier_list
        | type_specifier
        | type_qualifier specifier_qualifier_list
        | type_qualifier
        ;

struct_declarator_list
        : struct_declarator
        | struct_declarator_list ',' struct_declarator
        ;

struct_declarator
        : declarator
        | ':' constant_expression
        | declarator ':' constant_expression
        ;

enum_specifier
        : ENUM '{' enumerator_list '}'
        | ENUM IDENTIFIER '{' enumerator_list '}'
        | ENUM '{' enumerator_list ',' '}'
        | ENUM IDENTIFIER '{' enumerator_list ',' '}'
        | ENUM IDENTIFIER
        ;

enumerator_list
        : enumerator
        | enumerator_list ',' enumerator
        ;

enumerator
        : IDENTIFIER
        | IDENTIFIER '=' constant_expression
        ;

type_qualifier
        : CONST
        | RESTRICT
        | VOLATILE
        ;

function_specifier
```

(suite sur la page suivante)

(suite de la page précédente)

```
        : INLINE
        ;

declarator
        : pointer direct_declarator
        | direct_declarator
        ;


direct_declarator
        : IDENTIFIER
        | '(' declarator ')'
        | direct_declarator '[' type_qualifier_list assignment_
 ↪expression ']'
        | direct_declarator '[' type_qualifier_list ']'
        | direct_declarator '[' assignment_expression ']'
        | direct_declarator '[' STATIC type_qualifier_list␣
 ↪assignment_expression ']'
        | direct_declarator '[' type_qualifier_list STATIC␣
 ↪assignment_expression ']'
        | direct_declarator '[' type_qualifier_list '*' ']'
        | direct_declarator '[' '*' ']'
        | direct_declarator '[' ']'
        | direct_declarator '(' parameter_type_list ')'
        | direct_declarator '(' identifier_list ')'
        | direct_declarator '(' ')'
        ;

pointer
        : '*'
        | '*' type_qualifier_list
        | '*' pointer
        | '*' type_qualifier_list pointer
        ;

type_qualifier_list
        : type_qualifier
        | type_qualifier_list type_qualifier
        ;


parameter_type_list
        : parameter_list
        | parameter_list ',' ELLIPSIS
        ;

parameter_list
        : parameter_declaration
        | parameter_list ',' parameter_declaration
```

(suite sur la page suivante)

(suite de la page précédente)

```
        ;

parameter_declaration
        : declaration_specifiers declarator
        | declaration_specifiers abstract_declarator
        | declaration_specifiers
        ;

identifier_list
        : IDENTIFIER
        | identifier_list ',' IDENTIFIER
        ;

type_name
        : specifier_qualifier_list
        | specifier_qualifier_list abstract_declarator
        ;

abstract_declarator
        : pointer
        | direct_abstract_declarator
        | pointer direct_abstract_declarator
        ;

direct_abstract_declarator
        : '(' abstract_declarator ')'
        | '[' ']'
        | '[' assignment_expression ']'
        | direct_abstract_declarator '[' ']'
        | direct_abstract_declarator '[' assignment_expression ']'
        | '[' '*' ']'
        | direct_abstract_declarator '[' '*' ']'
        | '(' ')'
        | '(' parameter_type_list ')'
        | direct_abstract_declarator '(' ')'
        | direct_abstract_declarator '(' parameter_type_list ')'
        ;

initializer
        : assignment_expression
        | '{' initializer_list '}'
        | '{' initializer_list ',' '}'
        ;

initializer_list
        : initializer
        | designation initializer
        | initializer_list ',' initializer
        | initializer_list ',' designation initializer
```

(suite sur la page suivante)

(suite de la page précédente)

```
        ;

designation
        : designator_list '='
        ;

designator_list
        : designator
        | designator_list designator
        ;

designator
        : '[' constant_expression ']'
        | '.' IDENTIFIER
        ;

statement
        : labeled_statement
        | compound_statement
        | expression_statement
        | selection_statement
        | iteration_statement
        | jump_statement
        ;

labeled_statement
        : IDENTIFIER ':' statement
        | CASE constant_expression ':' statement
        | DEFAULT ':' statement
        ;

compound_statement
        : '{' '}'
        | '{' block_item_list '}'
        ;

block_item_list
        : block_item
        | block_item_list block_item
        ;

block_item
        : declaration
        | statement
        ;

expression_statement
        : ';'
        | expression ';'
```

(suite sur la page suivante)

(suite de la page précédente)

```
        ;

selection_statement
        : IF '(' expression ')' statement
        | IF '(' expression ')' statement ELSE statement
        | SWITCH '(' expression ')' statement
        ;

iteration_statement
        : WHILE '(' expression ')' statement
        | DO statement WHILE '(' expression ')' ';'
        | FOR '(' expression_statement expression_statement ')'␣
↪statement
        | FOR '(' expression_statement expression_statement␣
↪expression ')' statement
        | FOR '(' declaration expression_statement ')' statement
        | FOR '(' declaration expression_statement expression ')'␣
↪statement
        ;

jump_statement
        : GOTO IDENTIFIER ';'
        | CONTINUE ';'
        | BREAK ';'
        | RETURN ';'
        | RETURN expression ';'
        ;

translation_unit
        : external_declaration
        | translation_unit external_declaration
        ;

external_declaration
        : function_definition
        | declaration
        ;

function_definition
        : declaration_specifiers declarator declaration_list␣
↪compound_statement
        | declaration_specifiers declarator compound_statement
        ;

declaration_list
        : declaration
        | declaration_list declaration
        ;
```

(suite sur la page suivante)

(suite de la page précédente)

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

extern char yytext[];
extern int column;

void yyerror(char const *s)
{
        fflush(stdout);
        printf("\n%*s\n%*s\n", column, "^", column, s);
}
```

Annexe C

Ligne de commande

La maîtrise de la ligne de commande n'est pas indispensable pour ce cours mais la compréhension de quelques commandes est utile pour bien comprendre les exemples données.

Dans un environnement **POSIX** l'interaction avec le système s'effectue pour la plupart du temps via un terminal. Le programme utilisé pour intéragir avec le système d'exploitation est appelé un interpréteur de commande. Sous Windows vous utilisez le programme `cmd.exe` ou `PowerShell.exe`. Sous Linux vous utilisez très souvant un dérivé du Bourne shell nom éponyme de Stephen Bourne et apparu en 1979. La compatibilité est toujours maintenu aujourd'hui via son successeur Bash dont le nom est un acronyme de *Bourne-again shell*.

Bash est écrit en C et les sources sont naturellement disponible sur internet. Lorsque vous lancez Bash vous aurez un simple prompt :

```
$
```

Ce dernier vous invite à tapper une commande laquelle est le plus souvent le nom d'un programme.

C.1 Pipe

```
$ cat foo.txt | hexdump -C -n100
```

Annexe D

Environnement de développement

Si ce n'est pas déjà le cas, vous devriez disposer sur votre ordinateur d'un environnement de développement. Pour ce cours, vous êtes libre d'utiliser l'outil qui vous semble le plus pertinent mais il proposé d'utiliser un environnement POSIX qui est aujourd'hui le standard le plus établi et le plus largement utiilsé, des téléphones portables aux serveurs informatiques, aux Macbooks et aux fusées spatiales. L'autre consensus, également largement utilisé est Windows.

Pour disposer d'un environnement POSIX sous Windows il existe 3 solutions :

MinGW Un environnement POSIX GNU pour Windows. La solution la plus facile.

WSL Windows Subsystem for Linux Une distribution Ubuntu pour Windows 10. La meilleure solution si vous voulez vous confronter à la ligne de commande et mieux comprendre le fonctionnement de Linux.

Cygwin Un émulateur POSIX pour Windows. Une excellente alternative aux autres solutions qui se fait peu à peu remplacer par WSL.

Pour installer le plus facilement votre environnement de développement, je vous recommande Chocolatey, un gestionnaire de paquets pour Windows. Une fois l'outil installé, vous pouvez installer les paquets suivants :

```
C:\> choco install gcc
C:\> choco install vscode
C:\> choco install git
```

Ensuite vous pouvez exécuter `Git Bash` depuis le menu Démarrer et diposer d'une console comme si vous étiez sous Linux ou MacOSx.

Annexe E

Fiches d'unités

E.1 Informatique 1

```yaml
# Version formelle et étendue de la fiche d'unité de cours␣
↪disponible sur GAPS
# http://gaps.heig-vd.ch/public/fiches/uv/uv.php?id=5637
---
title:
    name: Informatique 1
    tag: info1
    id: 5637
domain: Ingénierie et Architecture
filière: Génie électrique
orientation:
    name: Électronique et Automatisation industrielle
    id: EAI
formation: Plein temps
validityDate:
    - 2018-2019
    - 2019-2020
author: François Birling
charge:
    academicHours: 150
    inClassAcademicHours: 96
planning:
    s1:
        class: # 48
            - hours: 4
              chapters:
                - Introduction.
                - Aperçu du fonctionnement de l'ordinateur.
                - Codage de l'information
            - hours: 2
              chapters:
                - Présentation du langage C
```

(suite sur la page suivante)

(suite de la page précédente)

```
            - hours: 12
              chapters:
                - Types de données de base
                - Variables
                - Constantes
                - Opérateurs
                - Entrées et sorties console
            - hours: 8
              chapters:
                - Structures de contrôle
                - Branchements
                - Boucles
            - hours: 6
              chapters:
                - Fonctions
            - hours: 8
              chapters:
                - Types avanceés
                    - Tableaux
                    - Structures
                    - Chaînes de caractères
            - hours: 4
              chapters:
                - Introduction à l'analyse et à la conception
            - hours: 4
              chapters:
                - Controle continu et corrections
        laboratory:
            - hours: 2
              chapters:
                - Environnement de travail (réseau informatique)
                - Outils bureautique
            - hours: 2
              chapters:
                - Environnement de développement intégré
                    - Installation
                    - Configuration
                    - Édition
                    - Compilation
            - hours: 8
              chapters:
                - Dialogues utilisateurs
            - hours: 10
              chapters:
                - Utilisation des structures de contrôle.
            - hours: 8
              chapters:
                - Type de données composés
            - hours: 8
```

(suite sur la page suivante)

(suite de la page précédente)

```
            chapters:
              - Mini projet.
prerequisites: |
    L'étudiant-e doit connaître et savoir utiliser les notions␣
↪suivantes
        - utilisation d'un système d'exploitation, notamment la␣
↪gestion de fichiers
        - bases des outils de bureautique.
    L'unité préparatoire d'informatique UPI permet d'acquérir ces␣
↪connaissances.
    Conditions pour la programmation automatique de cette unité :
    L'étudiant-e doit avoir obtenu une note supérieure ou égale à␣
↪la limite de compensation dans les unités :
    néant. L'étudiant-e doit avoir suivi ou suivre en parallèle les␣
↪unités : néant.
goals:
    classroom:
        - Expliquer les principes généraux de représentation de l
↪'information dans les ordinateurs.
        - Décrire la marche à suivre et les outils nécessaires pour␣
↪créer un programme.
        - Citer les éléments du langage C utilisés couramment pour␣
↪écrire des programmes.
        - Choisir le type de données simple approprié pour␣
↪représenter les informations du monde réel.
        - Concevoir et programmer un dialogue opérateur en mode␣
↪console, formater un affichage pour le rendre lisible.
        - Calculer la valeur d'une expression construite avec␣
↪différents opérateurs du langage C et en déterminer le type.
        - Choisir la structure de contrôle appropriée pour résoudre␣
↪un problème algorithmique simple.
        - Concevoir et implanter un algorithme imbriquant jusqu'à 3␣
↪niveaux de structure de contrôle.
        - Créer une fonction utilisant le passage de paramètres par␣
↪valeur, par adresse et/ou le retour d'un résultat.
        - Construire et utiliser un type de données composé␣
↪(tableau, structure).
        - Utiliser les principales fonctions des bibliothèques␣
↪`math.h` et `string.h`
        - Corriger la présentation, les erreurs de syntaxe et de␣
↪sémantique dans un programme fourni.
    laboratory:
        - Installer et configurer un environnement de développement␣
↪intégré (EDI) pour le langage C.
        - Créer des programmes avec un EDI (éditer un code source,␣
↪le compiler, le tester, le déverminer).
        - Créer un programme gérant un menu en mode console et␣
↪affichant des résultats sous forme tabulaire.
        - Mettre au point itérativement un programme pour atteindre␣
↪un fonctionnement fiable et ergonomique.
```

(suite sur la page suivante)

(suite de la page précédente)

```
        - Comprendre un cahier des charges, identifier et clarifier␣
↪les exigences importantes, et s'y conformer.
        - Analyser de manière autonome les problèmes rencontrés et␣
↪formuler une question précise.
        - Livrer un logiciel de façon professionnelle (organisation␣
↪des livrables, gestion des exigences et du délai).
        - Citer des applications pratiques de la programmation en␣
↪relation avec ses futurs débouchés professionnels.
plan:
  - Numération
    - Bases (système décimal, hexadécimal, octal, binaire)
    - Conversion de bases
    - Complément à un
    - Complément à deux
    - Arithmétique binaire (et, ou, ou exclusif, négation)
  - Processus de développement
    - Outils
      - Environnement intégré (IDE)
      - Compilateur (*compiler*)
      - Chaîne de développement (*toolchain*)
    - Cycle de développement
    - Cycle de compilation
    - Installation d'un environnement de développement
    - Programmes et processus
  - Généralités du langage C
    - Séquences
    - Embranchements (if, switch)
    - Boucles (while, do..while, for)
    - Sauts (break, continue, return, goto)
  - Types de données
    - Typage
    - Stockage des données en mémoire
    - Entiers naturels
    - Entiers relatifs
    - Nombres réels (virgule flottante)
    - Caractères
      - Table ASCII
    - Chaînes de caractères
    - Booléens
  - Interaction utilisateur en mode console
    - Entrée standard
    - Sortie standard
    - Sortie d'erreur standard
    - Questions/Réponses avec `printf` et `scanf`
    - Formater un résultat sous forme tabulée et lisible
    - Menu (choix multiples)
  - Opérateurs
    - Opérateurs du langage C
    - Priorité des opérateurs
```

(suite sur la page suivante)

(suite de la page précédente)

```
  - Expressions
  - Promotion et promotion implicite
- Conception
  - Choix des structures de controles adaptées à des problèmes
  - Algorithmes simple (min, max, moyenne, ...)
    - Manipulation de chaînes
    - Manipulation de tableaux
    - Manipulation de bits
- Algorithmie
  - Complexité d'un algorithme
  - Exemples d'algorithmes
  - Algorithmes de tri (tri à bulle)
- Fonctions
  - Passage par valeur et par adresse
  - Utilisation de la valeur de retour
  - Prototypes de fonctions
- Types de données composées
  - Structures
  - Unions
  - Tableaux
  - Énumérations
- Bibliothèques standard
  - <math.h>
    - Fonctions trigonométriques
    - Exponentielle
    - Logarithme
  - <string.h>
    - Comparaison de chaînes de caractères
    - Concaténation de chaînes de caractères
    - Copie de chaînes de caractères
    - Longueur d'une chaîne de caractères
    - Recherche d'une sous-chaîne dans une chaîne de caractères
  - <stdio.h>
    - printf
    - scanf
    - putchar
    - getchar
    - puts
    - gets
- Structure du code
  - Corriger les erreurs de syntaxes
  - Corriger les erreurs sémantiques
  - Indentation du code
  - Commentaires
```

E.1.1 Pannification du semestre

Semaine	Académique	Cours	Labo
38	1	Introduction	00 Familiarisation
39	2	Numération	01 Premier pas en C
40	3	Fondements du C	02 Équation Quadratique
41	4	Variables, opérateurs	03 Fléchettes
42	5	Types, entrées Sorties | 04 Nombre d'Armstrong	
43	Vacances d'automne		
44	6	Entrées sorties	05 Pneus
45	7	TE1	06 Tables Multiplications
46	8	Structure de contrôles	07 Chaînes
47	9	Fonctions	08 Monte-Carlo
48	10	Tableaux et structures	09 Sudoku
49	11	Programmes et processus	
50	12	Algorithmique	
51	13	Pointeurs	10 Galton
52	Vacances de Noël		
1			
2	14	Ergonomie et dialogues	12 Tableau des scores
3	15	TE2	
4	16	Exercices de révision	
5	Préparation Examens		
6	Examens		
7	Relâches		

E.2 Informatique 2

```
# Version formelle et étendue de la fiche d'unité de cours␣
↪disponible sur GAPS
# http://gaps.heig-vd.ch/public/fiches/uv/uv.php?id=5638
---
title:
    name: Informatique 2
    tag: info2
    id: 5638
domain: Ingénierie et Architecture
filière: Génie électrique
orientation:
    name: Électronique et Automatisation industrielle
    id: EAI
formation: Plein temps
validityDate:
    - 2018-2019
    - 2019-2020
```

(suite sur la page suivante)

(suite de la page précédente)

```
author: François Birling
charge:
    academicHours: 120
    inClassAcademicHours: 80
planning:
    s1:
        class: # 48
            - hours: 4
              chapters:
                - Préprocesseur
            - hours: 4
              chapters:
                - Classes de stockage
            - hours: 8
              chapters:
                - Conception de type de données abstraits simples
                - Création de bibliothèques
            - hours: 10
              chapters:
                - Pointeurs
                - Allocation dynamique
            - hours: 6
              chapters:
                - Implémentation des listes
                - Queues et piles basée sur les tableaux
            - hours: 8
              chapters:
                - Type de données récursifs, queues et piles
            - hours: 4
              chapters:
                - Fichiers binaires et textes
            - hours: 4
              chapters:
                - Contrôles continus
        laboratory:
            - hours: 6
              chapters:
                - Mise en oeuvre de type de données composées␣
↪(structures, tableaux multidimensionnels)
            - hours: 4
              chapters:
                - Lecture et écriture de fichiers texte et binaire␣
↪en mode séquentiel
            - hours: 4
              chapters:
                - Mise en oeuvre de l'allocation dynamique de␣
↪mémoire
            - hours: 2
              chapters:
```

(suite sur la page suivante)

(suite de la page précédente)

```
                - Compilation séparée et implémentation de␣
↪bibliothèques
            - hours: 4
              chapters:
                - Implémentation de types de données abstraits,␣
↪type simple, liste tableau
            - hours: 6
              chapters:
                - Implémentation de types de données abstraits,␣
↪file, pile
            - hours: 6
              chapters:
                - Mini-projet
prerequisites: |
    L'étudiant-e doit connaître et savoir utiliser les notions␣
↪suivantes
        - base de la programmation, types de base, structures de␣
↪contrôle et sous-programmes
        - bases des outils de bureautique.
    L'unité d'enseignement APR1 (analyse et programmation) permet d
↪'acquérir ces connaissances

goals:
    class:
        - Décomposer un algorithme selon l'approche descendante␣
↪(raffinage successif) et ascendante.
        - Décomposer une application de complexité moyenne en␣
↪algorithmes élémentaires.
        - Concevoir un type de données abstrait simple et les␣
↪fonctions pour le manipuler.
        - Concevoir une bibliothèque de fonctions en utilisant la␣
↪compilation séparée.
        - Ecrire un programme qui lit ou écrit un fichier au format␣
↪binaire, texte.
        - Mettre en oeuvre l'allocation dynamique de mémoire pour␣
↪créer des tableaux dimensionnés à
l'exécution.
        - Définir et manipuler un type de données récursif (liste␣
↪chaînée).
        - Mettre en oeuvre des champs de bits et des unions pour␣
↪manipuler des bits dans un masque binaire.
    laboratory:
        - Programmer une bibliothèque de fonctions de qualité␣
↪professionnelle (et l'utiliser).
        - Programmer et mettre au point des algorithmes de␣
↪complexité moyenne.
        - Réaliser une application de taille et de complexité␣
↪moyennes, mêlant différents aspects de la
programmation.
```

(suite sur la page suivante)

(suite de la page précédente)

```
plan:
  - Algorithmie
    - Raffinage successif
    - Décomposition en éléments fonctionnels simples
    - Conception d'algorithmes de complexité moyenne
  - Types composés
    - Manipulation d'une structure
    - Passage par copie et référence
  - Bibliothèque
    - Concevoir une bibliothèque statique
    - Utilisation d'une bibliothèque statique dans un programme
  - Fichiers
    - Types de fichiers
      - Binaire
      - Textes
      - Données tabulées
      - Données indexées
    - Système de fichier
      - Arborescence
      - Dossiers
      - Chemins relatifs et absolus
    - Manipulation de fichiers
      - Pointeur de fichier (ftell, fseek)
      - Lecture (fread, fscanf)
      - Écriture (fwrite, fprintf)
  - Gestion de la mémoire
    - Pointeurs
      - Règle gauche droite
    - Allocation dynamique
      - malloc
      - calloc
      - free
      - Création de tableaux dynamiques
  - Types de données récursifs
    - Liste simplement chaînée
    - Liste doublement chaînée
  - Alignement mémoire
    - Unions
    - Champs de bits
  - Livraison
    - Préparer le code à la livraison
    - Construire une bibliothèque documentée
```

E.2.1 Pannification du semestre

Semaine	Académique	Cours	Labo
8	1	Introduction	GitHub - WSL
9	2	Fichiers	Proust (partie 1)
10	3	Allocation dynamique	Proust (partie 2)
11	4	Allocation dynamique	Météo (partie 1)
12	5	Compilation séparée	Météo (partie 2)
13	6	Préprocesseur	Tableau Dynamique (1/2)
14	7	Unions, champs de bits	Tableau Dynamique (2/2)
15	8	Usage bibliothèques	Révision TE1
16	Vacances de Pâques		
17	9	TE1	GMP
18	10	Algorithmique Big-O	
19	11	Tris	Quick-Sort / Merge-Sort
20	12	Queues et piles	Révision TE2
21	13	Sockets	Labo Test
22	14	TE2	
23	15	Arbres binaires	BST
24	16	Exercices de révision	
25	Préparation Examens		
26	Examens		

E.3 Modalités d'évalutation et de validation

Le cours se composent de :

- Travaux écrits notés (coefficient 100%)
- Quiz notés ou non (coefficient 10% ou 0%)
- Séries d'exercices
- Travaux pratiques, 2 à 3 labos notés (laboratoires)
- Labo test noté comptabilisé comme un labo

La note finale est donnée par l'expression :

$$\text{final} = \frac{\sum_{t=1}^{\text{T}} \text{TE}_t + 10\% \cdot \sum_{q=1}^{\text{Q}} \text{Quiz}_q}{4 \cdot (\text{T} + 10\% \cdot \text{Q})} + \frac{1}{4L} \sum_{l=1}^{L} \text{Labo}_l + \frac{1}{2}\text{Exam}$$

L'equivalent en C :

```c
#define QUIZ_WEIGHT (.1) // Percent
#define EXAM_WEIGHT (.5) // Percent

typedef struct notes {
    size_t size;
    float values[];
```

(suite sur la page suivante)

(suite de la page précédente)

```
} Notes;

float sum(Notes *notes) {
    float s = 0;
    for (int i = 0; i < notes->size; i++)
        s += notes->values[i];
    return s;
}

float mark(Notes tes, Notes quizzes, Notes labs, float exam) {
    return (
    sum(tes) + QUIZ_WEIGHT * sum(quizzes)
    ) / (
        (EXAM_WEIGHT / 2.) * (tes.size + QUIZ_WEIGHT * quizzes.size)
    ) +
            (EXAM_WEIGHT / 2.) * sum(labs) / labs.size +
        EXAM_WEIGHT * exam;
}
```

Les règles sont :

— En cas d'absence à un quiz, la note de 1.0 est donnée.
— En cas de plagiat, le dilemme du prisionnier s'applique.
— Le quiz le plus mauvais ne compte pas.

Annexe F

Laboratoires

Les laboratoires sont des travaux pratiques permettant à l'étudiant d'attaquer des problèmes de programmation plus difficiles que les exercices faits en classe.

F.1 Protocole

1. Récupérer le référentiel du laboratoire en utilisant GitHub Classroom.
2. Prendre connaissance du cahier des charges.
3. Rédiger le code.
4. Le tester.
5. Rédiger votre rapport de test si demandé.
6. Le soumettre avant la date butoire.

F.2 Evaluation

Une grille d'évaluation est intégrée à tous les laboratoires. Elle prends la forme d'un fichier `criteria.yml` que l'étudiant peut consulter en tout temps.

F.3 Directives

— La recherche sur internet est autorisée et conseillée.
— Le plagia n'est pas autorisé, et sanctionné si découvert par la note de 1.0.
— Le rendu passé la date butoire est sanctionné à raison de 1 point puis 1/24 de point par heure de retard.

F.4 Format de rendu

- Fin de lignes : `LF` (`'\n'`).
- Encodage : UTF-8 sans BOM.
- Code source respectueux de ISO/IEC 9899 :1999.
- Le code doit comporter un exemple d'utilisation et une documentation mise à jour dans `README.md`.
- Lorsqu'un rapport est demandé vous le placerez dans `REPORT.md`.

F.5 Anatomie d'un travail pratique

Un certain nombre de fichiers vous sont donnés, il est utile de les connaîtres. Un référentiel sera généralement composé des éléments suivants :

```
$ tree
.
├── .clang-format
├── .devcontainer
│   ├── Dockerfile
│   └── devcontainer.json
├── .editorconfig
├── .gitattributes
├── .gitignore
├── .vscode
│   ├── launch.json
│   └── tasks.json
├── Makefile
├── README.md
├── assets
│   └── test.txt
├── foo.c
├── foo.h
├── main.c
├── criteria.yml
└── tests
    ├── Makefile
    └── test_foo.c
```

F.5.1 .clang-format

Ce fichier est au format YAML et contient des directives pour formater votre code automatiquement soit à partir de VsCode si vous avez installé l'extension Clang-Format et l'exécutable `clang-format` (`sudo apt install -y clang-format`). Clang-format est un utilitaire de la suite LLVM, proposant Clang un compilateur alternatif à GCC.

On voit que le texte passé sur `stdin` (jusqu'à EOF) est ensuite formatté proprement :

```
$ clang-format --style=mozilla <<EOF
#include <stdio.h>
int
main
()
{printf("hello, world\n");}
EOF
#include <stdio.h>
int
main()
{
printf("hello, world\n");
}
```

Par défaut `clang-format` utilise le fichier de configuration nommé `.clang-format` qu'il trouve.

Vous pouvez générer votre propre configuration facilement depuis un configurateur tel que clang-format configurator.

F.5.2 .editor_config

Ce fichier au format YAML permet de spécifier des recommendations pour l'édition de fichiers sources. Vous pouvez y spécifier le type de fin de lignes **CR** ou **CRLF**, le type d'indentation (espaces ou tabulations) et le type d'encodage (ASCII ou UTF-8) pour chaque type de fichiers. EditorConfig est aujourd'hui supporté par la plupart des éditeurs de textes qui cherchent automatiquement un fichier de configuration nommé `.editor_config`.

Dans Visual Studio Code, il faut installer l'extension EditorConfig for VS Code pour bénéficier de ce fichier.

Pour les travaux pratique on se contente de spécifier les directives suivantes :

```
root = true

[*]
end_of_line = lf
insert_final_newline = true
indent_style = space
```

(suite sur la page suivante)

(suite de la page précédente)

```
indent_size = 4
charset = utf-8

[*.{json,yaml}]
indent_style = space
indent_size = 2

[Makefile]
indent_style = tab

[*.{cmd,bat}]
end_of_line = crlf
```

F.5.3 .gitattributes

Ce fichier permet à Git de résoudre certains problèmes dans l'édition de fichiers sous Windows ou POSIX lorsque le type de fichiers n'a pas le bon format. On se contente de définir quel sera la fin de ligne standard pour certain type de fichiers :

F.5.4 .gitignore

Ce fichier de configuration permet à Git d'ignorer par défaut certains fichiers et ainsi éviter qu'ils ne soient ajoutés par erreur au référentiel. Ici, on souhaite éviter d'ajouter les fichiers objets `.o` et les exécutables `*.out` :

F.5.5 .vscode/launch.json

Ce fichier permet à Visual Studio Code de savoir comment exécuter le programme en mode débug. Il est au format JSON. Les lignes importantes sont `program` qui contient le nom de l'exécutable à lancer `args` qui spécifie les arguments passés à ce programme et `MiMode` qui est le nom du débogueur que vous utiliserez. Par défaut nous utilisons GDB.

```
{
    "version": "0.2.0",
    "configurations": [
        {
            "name": "Launch Main",
            "type": "cppdbg",
            "request": "launch",
            "program": "${workspaceFolder}/a.out",
            "args": ["--foobar", "filename", "<<<", "hello, world"],
            "stopAtEntry": true,
            "cwd": "${workspaceFolder}",
```

(suite sur la page suivante)

(suite de la page précédente)

```
            "environment": [],
            "externalConsole": false,
            "MIMode": "gdb",
            "setupCommands": [
                {
                    "description": "Enable pretty-printing for gdb",
                    "text": "-enable-pretty-printing",
                    "ignoreFailures": true
                }
            ],
            "preLaunchTask": "Build Main"
        }
    ]
}
```

F.5.6 .vscode/tasks.json

Ce fichier contient les directives de compilation utilisées par Visual Studio Code lors de l'exécution de la tâche *build* accessible par la touche <F5>. On y voit que la commande exécutée est `make`. Donc la manière dont l'exécutable est généré dépend d'un `Makefile`.

F.5.7 Makefile

Ce fichier contient les directives nécessaires au programme `make` pour générer votre exécutable. Vous pouvez vous inspirer de ce `Makefile` générique mais n'oubliez pas que la tabulation dans un Makefile doit être le caractère tabulation (pas des espaces). Si vous avez l'extension EditorConfig installée pour votre éditeur vous pouvez reformater le fichier avant de l'enregistrer.

```
CSRCS=$(wildcard *.c)
COBJS=$(patsubst %.c,%.o,$(CSRCS))
EXEC?=a.out

CFLAGS=-std=c99 -g -Wall -pedantic
LDFLAGS=-lm

all: $(EXEC)

-include $(COBJS:.o=.d)

$(EXEC): $(COBJS)
    $(CC) -o $@ $^ $(LDFLAGS)

%.o: %.c
    $(CC) -c $(CFLAGS) -o $@ $< -MMD -MF $(@:.o=.d)
```

(suite sur la page suivante)

(suite de la page précédente)

```
clean:
    $(RM) $(EXEC) *.o a.out $(COBJS:.o=.d)

.PHONY: all prof clean
```

En substance, ce fichier contient des règles, des dépendances et des recettes de fabrication. Les règles de base sont `all` et `clean`. La règle `all` dépend de la règle `$(EXEC)` qui est une variable qui contient le nom de l'exécutable, ici `a.out`. Vous pouvez spécifier le nom de l'exécutable souhaité à la ligne `EXEC=mon_executable`. La règle `$(EXEC)` dépend de `$(COBJS)` qui sont la liste des objets C, à savoir tous les fichiers `.c` dont l'extension est remplacée par `.o`. Une règle générique permet ensuite de générer tous les fichiers objets nécessaires à partir du fichier C correspondant : `%.o: %.c`. Enfin, en compilation séparée, l'exécutable est créé en assemblant tous les fichiers objets.

Pas de panique, il vous suffit de savoir exécuter `make all` ou `make clean` pour vous en sortir.

F.5.8 README.md

Il s'agit de la documentation principale de votre référentiel. Elle contient la donnée du travail pratique en format Markdown. Ce fichier est également utilisé par défaut dans GitHub. Il contient notament le titre du laboratoire, la durée, le délai de rendu et le format individuel ou de groupe :

F.5.9 criteria.yml

Ce fichier contient les directives d'évaluation du travail pratique. Il est au format YAML. Pour chaque point évalué une description est donnée avec la clé `description` et un nombre de point est spécifié. Une exigence peut avoir soit un nombre de point positif soit négatif. Les points négatifs agissent comme une pénalité. Ce choix d'avoir des points et des pénalités permet de ne pas dilluer les exigences au travers d'autre critères importants mais normalment respectés des étudiants.

Des points bonus sont donnés si le programme dispose d'une aide et d'une version et si la fonctionnalité du programme est étendue.

Ce fichier est utilisé par des tests automatique pour faciliter la correction du travail pratique.

Annexe G

Résumé

G.1 Introduction

Le langage C a créé en **1972** par Brian Kernighan et Dennis Ritchie est s'est dès lors imposé comme le standard industriel pour la programmation embarquée et pour tout développement nécessitant de hautes performances.

Le language est standardisé par l'ISO (standardisation internationale) et le standard le plus couramment utilisé en 2019 est encore C99.

Il faut retenir que le C est un langage dit :
- Impératif : programmation en séquences de commandes
- Structuré : programmation impérative avec des structures de contrôle imbriquées
- Procédural : programmation impérative avec appels de procédures

Ce sont ses paradigmes de programmation

G.2 Cycle de développement

Le cycle de dévelopment se compose toujours des phases : étude, écriture du cahier des charges, de l'écriture des tests, de la conception du logiciel, du codage à proprement parler et des validations finales. Le modèle en cascade est un bon résumé beaucoup utilisé dans l'industrie :

G.3 Cycle de compilation

Faire évoluer un logiciel est aussi un processus itératif :
- Editer le code avec un éditeur comme *vi* ou *vscode*
- **Compilation et prétraitement**

```
$ gcc -std=c99 -O2 -Wall -c foo.c -o foo.o
$ gcc -std=c99 -O2 -Wall -c bar.c -o bar.o
```

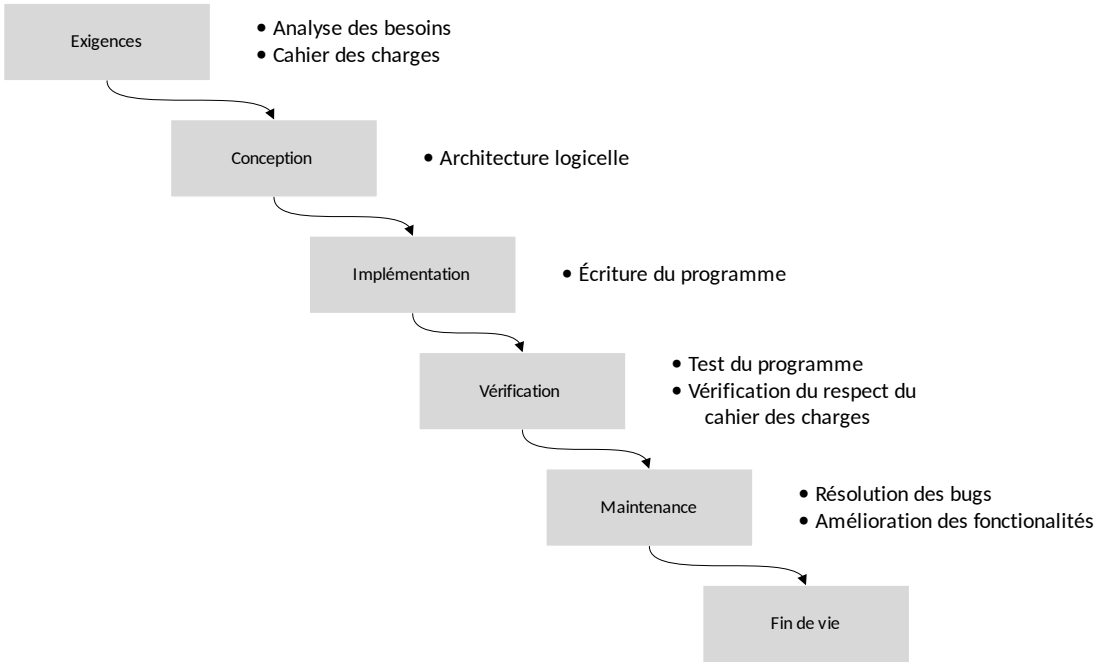


Fig. 7.1 – Modèle en cascade

— **Edition des liens**

```
$ gcc -o foobar foo.o bar.o -lm
```

— Tests

G.4 Make

Souvent, pour s'éviter de répéter les mêmes commandes les développeurs utilisent un outil comme *make* qui tire des règles de compilations d'un fichier nommé *Makefile*. Cet outil permet d'automatiquement recompiler les fichiers qui ne sont plus à jour et regénérer automatiquement l'exécutable. Certaines recettes de *make* sont souvent utilisées comme :

— `make all` Pour compiler tout le projet
— `make clean` Pour supprimer tous les fichiers intermédiaires générés
— `make mrproper` Pour supprimer tous les fihciers intermédiaires ainsi que les exécutables produits.
— `make test` Pour exécuter les tests de validation

D'autres recettes peuvent être écrites dans le fichier *Makefile* mais la courbe d'apprentissage du langage de *make* est raide.

G.5 Linux/POSIX

Un certain nombre de commandes sont utilisées durant ce cours et voici un résumé de ces dernières

Commande	Description
`cat`	Affiche sur `stdout` le contenu d'un fichier
`ls`	Liste le contenu du répertoire courant
`ls -al`	Liste le contenu du répertoire courant avec plus de détails
`echo`	Affiche sur `stdout` les éléments passés par argument au programme
`make`	Outil d'aide à la compilation utilisant le fichier `Makefile`
`gcc`	Compilateur open-source largement utilisé dans l'industrie
`vi`	Éditeur de texte ultra puissant mais difficile à apprendre

G.5.1 Programmation

G.6 Diagramme de flux

Le diagramme de flux est beaucoup utilisé pour exprimer un algorithme comme celui d'Euclide pour chercher le plus grand diviseur commun.

G.6.1 Langage C

G.7 Caractères non imprimables

Expression	Nom	Nom anglais	Description
`\n`	`LF`	*Line feed*	Retour à la ligne
`\v`	`VT`	*Vertical tab*	Tabulation verticale (entre les paragraphes)
`\f`	`FF`	*New page*	Saut de page
`\t`	`TAB`	*Horizontal tab*	Tabulation horizontale
`\r`	`CR`	*Carriage return*	Retour charriot
`\b`	`BS`	*Backspace*	Retour en arrière, effacement d'un caractère

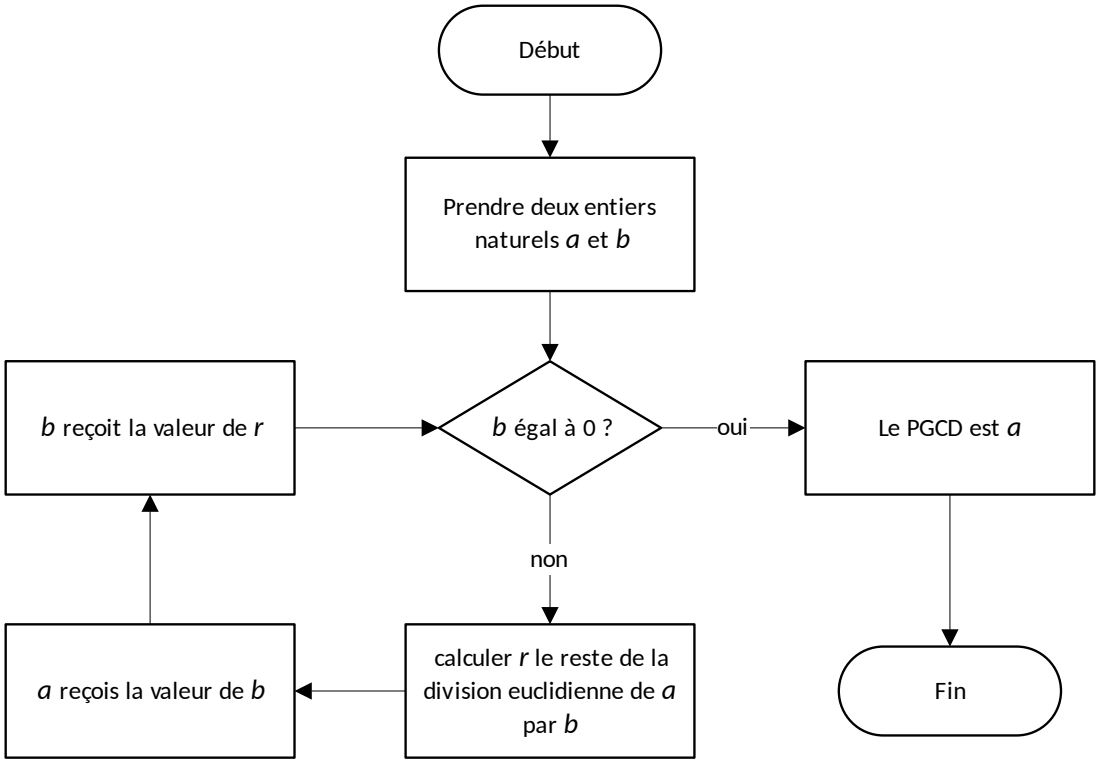


Fig. 7.2 – Algorithme de calcul du PGCD d'Euclide.

G.8 Fin de lignes

Les caractères de fin de ligne dépendent du système d'exploitation et sont appelé **EOL** : *End Of Line*.

Expression	Nom	Système d'exploitation
`\r\n`	CRLF	Windows
`\r`	CR	Anciens Macintoshs (< 2000)
`\n`	LF	Linux/Unix/POSIX

G.9 Identificateurs

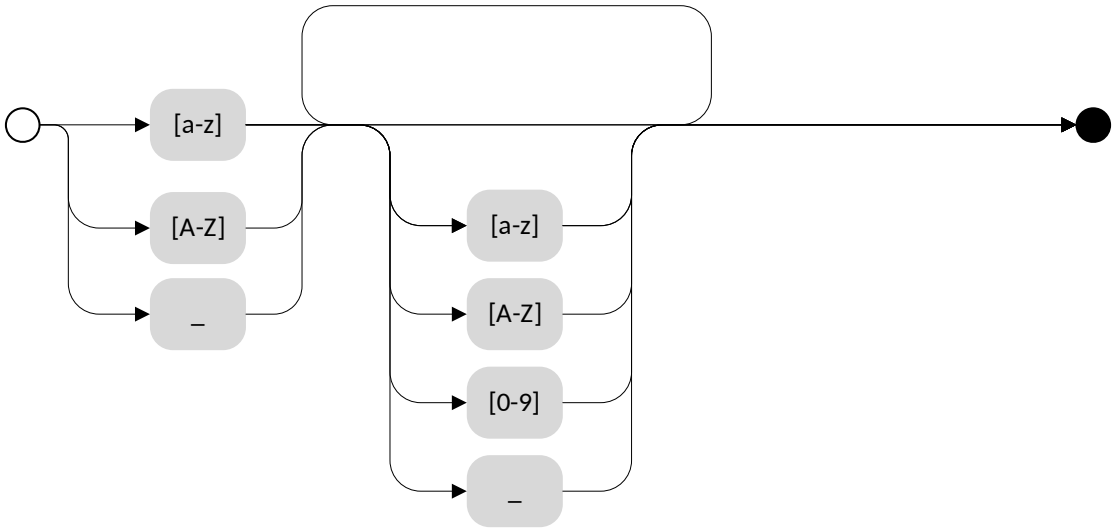


Fig. 7.3 – Grammaire d'un identificateur C

Le format des identificateurs peut également être exprimé par une expression régulière :

```
^[a-zA-Z_][a-zA-Z0-9_]*$
```

G.10 Variable

Une variable possède 6 paramètres : **nom**, **type**, **valeur**, **adresse**, **portée**, **visibilité**.

Elle peut être : **globale** et dans ce cas elle est automatiquement initialisée à 0 :

```c
int foo;

int main(void) {
    return foo;
}
```

Ou elle peut être locale et dasn ce cas il est nécessaire de l'initialiser à une valeur :

```c
int main(void) {
    int foo = 0;
    return foo;
}
```

Il est possible de déclarer plusieurs variable d'un même type sur la même ligne :

```c
int i, j, k;
int m = 32, n = 22;
```

Les conventions de nommage pour une variable sont : `camelCase` et `snake_case`, certains utilisent la notation `PascalCase`.

Les termes `toto`, `tata`, `foo`, `bar` sont souvent utilisés comme noms génériques et sont appelés termes *métasyntaxiques*.

G.11 Constantes littérales

Une constante littérale est une grandeur exprimant une valeur donnée qui n'est pas calculée à l'exécution :

Expression	Type	Description
`6`	`int`	Valeur décimale
`12u`	`unsigned int`	Valeur non signée en notation décimale
`6l`	`long`	Valeur longue en notation décimale
`010`	`int`	Valeur octale
`0xa`	`int`	Valeur hexadécimale
`0b111`	`int`	Valeur binaire (uniquement `gcc`, pas standard **C99**)
`12.`	`double`	Nombre réel
`'a'`	`char`	Caractère
`"salut"`	`char*`	Chaîne de caractère

G.12 Commentaires

Il existe deux types de commentaires :

— Les commentaires de lignes (depuis C99)

```
// This is a single line comment.
```

— Les commentaires de block

```
/* This is a
   Multi-line comment */
```

G.13 Fonction main

La fonction main peut s'érire sous deux formes :

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

```
int main(int argc, char *argv[]) {
    return 0;
}
```

G.13.1 Numération

Les données dans l'ordinateur sont stockées sous forme binaire et le *type* d'une variable permet de définir son interprétation.

— **Une valeur entière et non signée est exprimée sous la forme binaire pure :**

```
┌─┬─┬─┬─┬─┬─┬─┬─┐
│0│1│0│1│0│0│1│1│ = 0b1010011 = 83
└─┴─┴─┴─┴─┴─┴─┴─┘
```

— **Une valeur entière et signée est exprimée en complément à deux :**

```
┌─┬─┬─┬─┬─┬─┬─┬─┐   ┌─┬─┬─┬─┬─┬─┬─┬─┐
│1│1│0│1│0│0│1│1│ = ! │0│0│1│0│1│1│0│0│ = (-1) * (0b101100 + 1) = -45
└─┴─┴─┴─┴─┴─┴─┴─┘   └─┴─┴─┴─┴─┴─┴─┴─┘
```

— **Une valeur réelle ou flottante est exprimée selon le standard IEEE-754 et comporte**

```
┌ Signe 1 bit
│        ┌ Exposant 8 bits
│        │                             ┌ Mantisse 23 bits
┴ ───────┴────────  ──────────────────┴───────────────────────
┬ ┬
┌─┬─┬─┬─┬─┬─┬─┬─┐ ┌─┬─┬─┬─┬─┬─┬─┬─┐ ┌─┬─┬─┬─┬─┬─┬─┬─┐ ┌─┬─┬─┬─┬─┬─┬─┬─┐
│0│0│0│1│0│0│0│0│ │0│1│0│0│1│0│0│0│ │1│1│0│1│1│1│1│1│ │0│1│0│0│0│0│0│1│
└─┴─┴─┴─┴─┴─┴─┴─┘ └─┴─┴─┴─┴─┴─┴─┴─┘ └─┴─┴─┴─┴─┴─┴─┴─┘ └─┴─┴─┴─┴─┴─┴─┴─┘
```

G.13.2 Opérateurs

Les opérateurs appliquent une opération entre une ou plusieurs valeurs :

— Les opérateurs **unaire** s'appliquent à un seul opérande (`!12`, `~23`)

— Les opérateurs standards s'appliquent à deux opérandes (`12 ^ 32`)

Les opérateurs ont une priorité et une direction d'associativité :

```
u = ++a + b * c++ >> 3 ^ 2

Rang  Opérateur  Associativité
----  ---------  -------------
 1    ()++       -->
 2    ++()       <--
 2    +          <--
 2    *          <--
 5    >>         -->
 9    ^          -->
14    =          -->
```

Donc la priorité de ces opération sera :

```
(u = ((((++a) + (b * (c++))) >> 3) ^ 2))
```

Dans le cas des opérateurs de pré et post incrémentation, il sont en effet les plus prioritaires mais leur action est décalée dans le temps au précédant/suivant point de séquence. C'est à dire :

```
a += 1;
(u = (((a + (b * c)) >> 3) ^ 2));
c += 1;
```

G.14 Valeur gauche

Une valeur gauche *lvalue* défini ce qui peut se trouver à gauche d'une affectation. C'est un terme qui apparaît souvent dans les erreurs de compilation. L'exemple suivant retourne l'erreur : *lvalue required as increment operand* car le résultat de $a + b$ n'a pas d'emplacement mémoire et il n'est pas possible de l'assigner à quelque chose pour effectuer l'opération de pré-incrémentation.

```
c = ++(a + b);
```

Dans cet exemple c est une valeur gauche

G.14.1 Types de données

Dans 90% des cas voici les types qu'un dévelopeur utilisera en C et sur le modèle de donnée **LP64**

Type	Profondeur	Description
`char`	8-bit	Caractère ou valeur décimale
`int`	32-bit	Entier signé
`unsigned int`	32-bit	Entier non signé
`long long`	64-bit	Entier signé
`float`	32-bit	Nombre réel (23 bit de mantisse)
`double`	64-bit	Nombre réel (54 bit de mantisse)

Pour s'assurer d'une taille donnée on peut utiliser les types standard **C99** en incluant la bibliothèque `<stdint.h>`

```
#include <stdint.h>

int main(void) {
    int8_t foo = 0;  // Valeur signée sur 8-bit
    uint32_t bar = 0;  // Valeur non-signée sur 32-bit
```

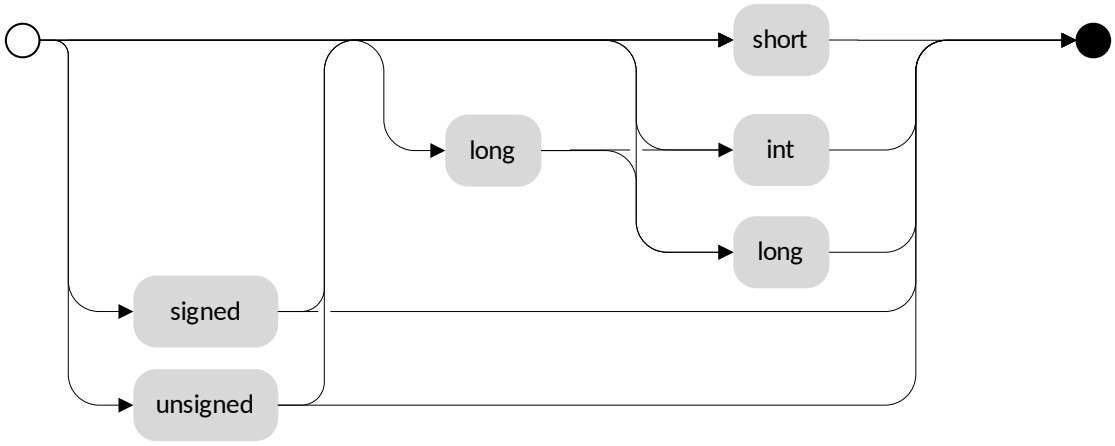
(suite sur la page suivante)

(suite de la page précédente)

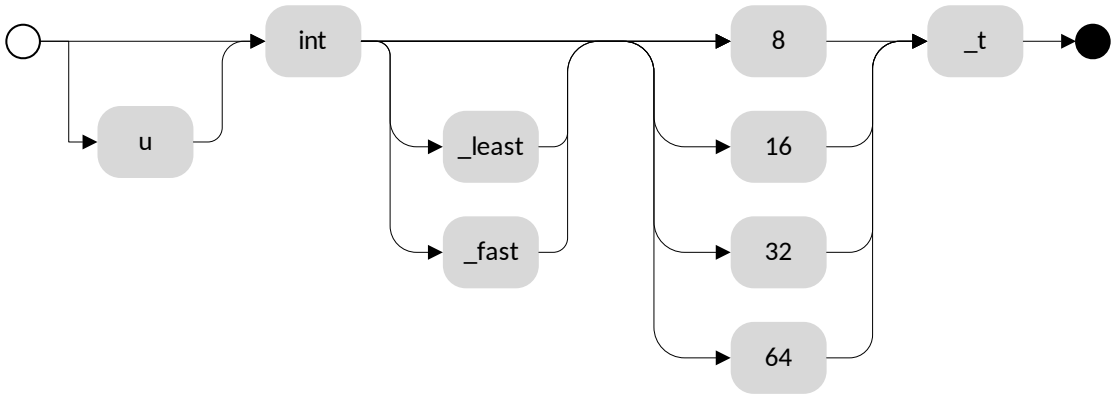
```
    uint_least16_t = 0;  // Valeur non-signée d'au moins 16-bit
}
```

Les valeurs signées sont exprimées en **complément à deux** c'est à dire que les valeurs maximales et minimales sont pour un entier 8-bit de `-128` à `+128`.

La construction des types standards :



La construction des types portables :



G.15 Caractères

Un caractère est une valeur binaire codée sur 8-bit et dont l'interprétation est confiée à une table de correspondance nommée ASCII :

	000 0x00	001 0x01	002 0x02	003 0x03	004 0x04	005 0x05	006 0x06	007 0x07	010 0x08	011 0x09	012 0x0A	013 0x0B	014 0x0C	015 0x0D	016 0x0E	017 0x0F
000 0x00	NUL \0	SOH	STX	ETX	EOT	ENQ	ACK	BEL \a	BS \b	TAB \t	LF \n	VT \v	FF \f	CR \r	SO	SI
020 0x10	DLE	DC1	DC2	DC3	DC4	NAK	SYN	ETB	CAN	EM	SUB	ESC	FS	GS	RS	US
040 0x20		!	"	#	$	%	&	'	(	)	*	+	,	-	.	/
060 0x30	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
0100 0x40	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
0120 0x50	P	Q	R	S	T	U	V	W	X	Y	Z	[	\ \\	]	^	_
0140 0x60	`	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
0160 0x70	p	q	r	s	t	u	v	w	x	y	z	{	|	}	~	DEL

Fig. 7.4 – Table ANSI INCITS 4-1986 (standard actuel)

Seul ces valeurs sont garanties d'être stockées sur 8-bit. Pour les caractères accentués ou les émoticônes, la manière dont ils sont codé en mémoire dépend de l'encodage des caractères. Souvent on utilise le type d'encodage **utf-8**.

Les écritures suivantes sont donc strictement identiques :

```c
char a;

a = 'a';
a = 0x61;
a = 97;
a = 0141;
```

G.16 Chaîne de caractère

Une chaîne de caractère est exprimée avec des guillemets double. Une chaîne de caractère comporte toujours un caractère terminal `\0`.

```c
char str[] = "Hello";
```

La taille en mémoire de cette chaîne de caractère est de 6 *bytes*, 5 caractères et un caractère de terminaison.

G.17 Booléens

En C la valeur `0` est considérée comme fausse (*false*) et une valeur différente de `0` est considérée comme vraie (*true*). Toutes les assertions suivantes sont vraies :

```c
if (42) { /* ... */ }
if (!0) { /* ... */ }
if (true && true || false) { /* ... */ }
```

Pour utiliser les mots clés `true` et `false` il faut utiliser la bibliothèque `<stdbool.h>`

G.18 Promotion implicite

Un type est automatiquement et tacitement promu dans le type le plus général :

```c
char a;
int b;
long long c;
unsigned int d;

a + b // Résultat promu en `int`
a + c // Résultat promu en `long long`
b + d // Résultat promu en `int`
```

Attention aux valeurs en virgule flottante :

```c
int a = 9, b = 2;
double b;

a / b;  // Résultat de type entier, donc 4 et non 4.5
(float)a / b;  // Résultat de type float donc 4.5
b / a;  // Résultat en type double (promotion)
```

G.19 Transtypage

Préfixer une variable ou une valeur avec `(int)` comme dans : `(int)a` permet de convertir explicitement cette variable dans le type donné.

Le transtypage peut être implcitie par exemple dans `int a = 4.5`

Ou plus spécifiquement dans :

```c
float u = 0.0;
printf("%f", b); // Promotion implicite de `float` en `double`
```

G.19.1 Structure de contrôle

G.20 Séquence

Une séquence est déterminée par un bloc de code entre accolades :

```
{
    int a = 12;
    b += a;
}
```

G.21 Si, sinon

```
if (condition)
{
    // Si vrai
}
else
{
    // Sinon
}
```

G.22 Si, sinon si, sinon

```
if (condition)
{
    // Si vrai
}
else if (autre_condition)
{
    // Sinon si autre condition valide
}
else
{
    // Sinon
}
```

G.23 Boucle For

```
for (int i = 0; i < 10; i++)
{
    // Block exécuté 10 fois
}

k = i; // Erreur car `i` n'est plus accessible ici...
```

G.24 Boucle While

```
int i = 10;

while (i > 0) {
    i--;
}
```

G.24.1 Programmes et Processus

Élement	Description
`stdin`	Entrée standard
`stdout`	Sortie standard
`stderr`	Sortie d'erreur standard
`argc`	Nombre d'arguments
`argv`	Valeurs des arguments
`exit-status`	Status de sortie d'un programme `$?`
`signaux`	Intéraction avec le système d'exploitation

G.24.2 Entrées Sorties

G.25 `printf`

Les sorties formattées utilisent *printf* dont le format est :

```
%[parameter][flags][width][.precision][length]type
```

`parameter` (optionnel) Numéro de paramètre à utiliser

`flags` (optionnel) Modificateurs : préfixe, signe plus, alignement à gauche ...

`width` (optionnel) Nombre **minimum** de caractères à utiliser pour l'affichage de la sortie.

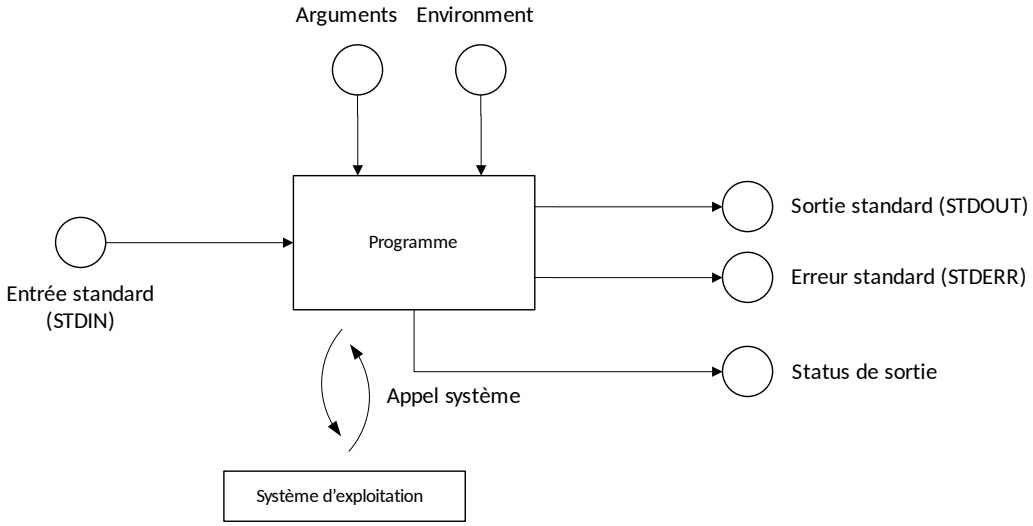


Fig. 7.5 – Résumé des interactions avec un programme

`.precision` (optionnel) Nombre **minimum** de caractères affichés à droite de la virgule. Essentiellement, valide pour les nombres à virgule flottante.

`length` (optionnel) Longueur en mémoire. Indique la longueur de la représentation binaire.

`type` Type de formatage souhaité

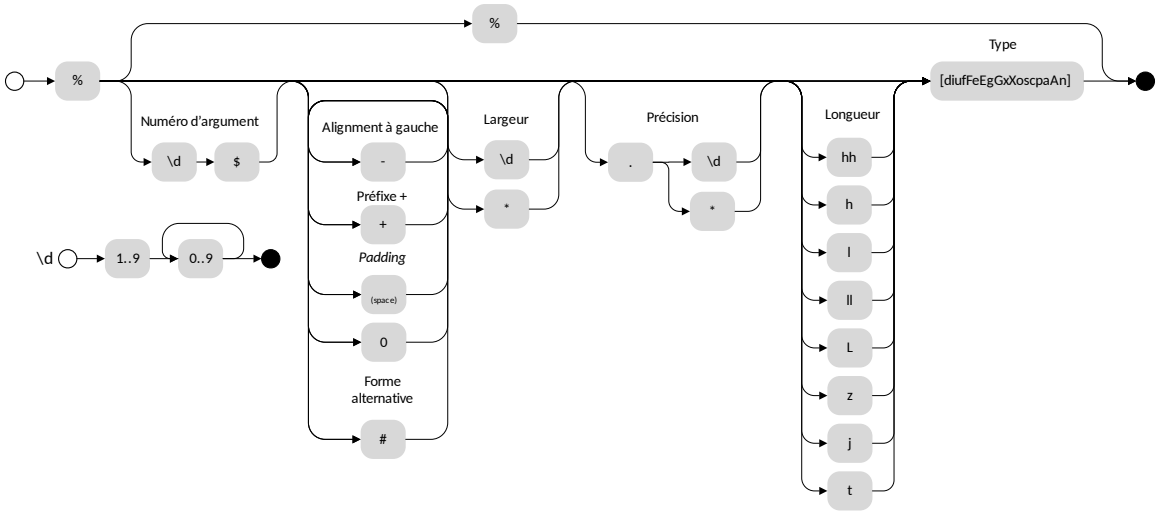


Fig. 7.6 – Formatage d'un marqueur

G.25.1 Techniques de programmation

G.26 Masque binaire

Pour tester si un bit est à un :

```
if (c & 0x040)
```

Pour forcer un bit à zéro :

```
c &= ~0x02;
```

Pour forcer un bit à un :

```
c |= 0x02;
```

G.27 Permuter deux variables sans valeur intermédiaire

```
a = b ^ c;
b = a ^ c;
a = b ^ c;
```

Table des figures

1.1 Les pères fondateurs du C 2
1.2 Représentation graphique des notions de compilateur, IDE, toolchain, ... 3
1.3 Modèle en cascade 8
1.4 Cycle de compilation illustré 9
1.5 Processus de pré-processing 10

2.1 L'un des premiers ordinateurs : l'Eniac 17
2.2 Algorithme de calcul du PGCD d'Euclide. 18
2.3 Mécanisme Jacquard au Musée des arts et métiers de Paris. 20
2.4 Armoiries des ducs de Mayenne 21

3.1 Distinction de différents caractères non-imprimables 27
3.2 Grammaire d'un identificateur C 28
3.3 Les carafes dans la Vivonne 35

4.1 1506 en écriture hiéroglyphique 40

5.1 Unité de calcul arithmétique (ALU) composées de deux entrées `A` et `B`, d'une sortie `C` et d'un mode opératoire `0`. 55

6.1 Table ASCII ASA X3.4 établie en 1963 82
6.2 Table ANSI INCITS 4-1986 (standard actuel) 83
6.3 Table d'extension ISO-8859-1 (haut) et ISO-8859-15 (bas) 83
6.4 Tendances sur l'encodage des pages web en faveur de UTF-8 dès 2008 . . 84

7.1 Exemples d'embranchements dans les diagrammes de flux BPMN (Business Process Modeleing Notation) et NSD (Nassi-Shneiderman) 100
7.2 Aperçu des trois structure de boucles 105

8.1 Programmeuse en tenue décontractée à côté de 62'500 cartes perforées . . 116
8.2 Résumé des interactions avec un programme 120

9.1 Formatage d'un marqueur 133

10.1 Margaret Hamilton la directrice du projet Apollo Guidance Computer (AGC) à côté du code du projet. 150
10.2 Le vieux Kamaji et ses bras extensibles. 151
10.3 Sauvegarde des registres du processeur et convention d'appel de fonction. 154

13.1 Allocation et libération mémoire 205
13.2 Organisation de mémoire d'un programme 206

14.1 Pointeur sur une chaîne de caractère 217

15.1 Bibliothèque du Trinity College de Dublin 229
15.2 Exemple d'interface graphique écrite avec *ncurses*. Ici la configuration du noyau Linux. 233

17.1 Différentes complexités d'algorithmes 255

19.1 Brouillard matinal sur le Golden Gate Bridge, San Francisco. 270

21.1 Arbre binaire équilibré 297

7.1 Modèle en cascade 370
7.2 Algorithme de calcul du PGCD d'Euclide. 372
7.3 Grammaire d'un identificateur C 373
7.4 Table ANSI INCITS 4-1986 (standard actuel) 379
7.5 Résumé des interactions avec un programme 383
7.6 Formatage d'un marqueur 383

Liste des tableaux

6.1 Types entiers standards 75
6.2 Entiers standard définis par `stdint` 76
6.3 Modèle de données 78

Index

-O2, 65
_Bool, 27
_Complex, 27
_Imaginary, 27

1801, 19
1834, 22
1937, 22
1950, 22
1965, 23
1972, 1
1985, 1
1989, 1
2018, 23
2019, 2

a.out, 14
abaque, 21
albatros, 42
algorithmique, 17
alphabet, 6, 25
ALU, 55
anglais, 4
ANSI, 82
apprendre, 5
arrondi, 52
associativité, 55
auto, 27

base, 39
Behold Summit, 23
big endian, 71
binaire, 40
Bit de signe, 46
booléens, 87
boutisme, 71
break, 27
Brian Kernighan, 1
build, 10

C11, 2
C18, 2
C90, 1
C99, 2
calculateur, 21
canuts, 19
caractère, 82
Carte perforée, 19
case, 27
casse, 28
CDC6600, 47
char, 27
chaîne de caractères, 85
Chiffres arabes, 25
CLANG, 4
Code ::Blocks, 4
compilateur, 3
complément à deux, 48
Complément à un, 47
const, 27
continue, 27
CR, 25
cycle de développement, 7

de gueules, 21
De Morgan, 51
default, 27
digraphes, 26
do, 27
double, 27
déclaration, 30

Ed, 4
else, 27
endianess, 71
entier relatif, 73
entiers naturels, 72
Entiers relatifs, 46
enum, 27
EOL, 26
Euclide, 18
exposant, 79
extern, 27

feu d'artifice, 5
FF, 25
float, 27, 81
Floyd, 293
for, 27

GCC, 4
goto, 27
grammaire, 6

hello, 12
hexadécimal, 42
héraldique, 19

identificateur, 28
if, 27
inline, 27
int, 27
ISO 80000-2, 69
ISO/IEC 8859, 82

Joseph Marie Jacquard, 19

Ken Thompson, 1
Kernel, 1
Kernighan, 1

latin1, 82
LF, 25
ligature, 12
link, 10
linked-list, 291
liste chaînée, 291
little endian, 71
long, 27
ls, 14

mantisse, 79
Maslow, 12
Matrix, 12

nombres entiers, 72
noyau, 1
numération, 39
Néo, 12

octal, 41
opérateur, 55
opérations bit-à-bit, 49
ordinateur, 21

paradigme, 6
pgcd, 18
point de séquence, 55
portée, 30
priorité, 55
programmation, 17
programmation impérative, 7
programmation procédurale, 7
programmation structurée, 7
précision simple, 81
préprocesseur, 8
pêcher, 5

quadruplets, 42

register, 27
restrict, 27
return, 27
rounding, 52
Révolte des canuts, 19

short, 27
signed, 27
sizeof, 27
standardisation, 1
static, 27
struct, 27, 165
switch, 27
système décimal, 40

TAB, 25
ternaire, 55
Thompson, 1
toolchain, 3
transcription, 7
trigraphes, 26
truncate, 52
typage, 69
type, 30
type incomplet, 89
typedef, 27
téléscripteurs, 26

union, 27
unsigned, 27
UTF-8, 84

valeur, 30
valeur gauche, 65
variable, 29
Vim, 4

virgule fixe, 78
virgule flottante, 79
visibilité, 30
Visual Studio, 3
void, 27, 89
volatile, 27
VsCode, 4
VT, 25

while, 27
world !, 12

Bibliographie

Les références utilisées dans cet ouvrage sont :

- Programming languages C – ISO/IEC 9899
- Le guide complet du langage C – Claude Delanoy, 844 pages (ISBN-13 978-2212140125)
- Le Langage C 2e edition – K&R, 304 pages (ISBN-13 978-2100715770)
- Cracking the coding interview – Gayle Laakmann, 687 pages (ISBN-13 978-0984782857)
- C : The Complete Reference, 4th Ed. – Osborne, 2.5 pounds (ISBN-13 978-0070411838)
- Die.net – https://www.die.net
- Wikipedia – https://www.wikipedia.org
- StackOverflow – https://stackoverflow.com

Colophon

L'écriture du présent ouvrage a été réalisée en utilisant reStructuredText, un langage couramment utilisé pour l'écriture de documentation et largement utilisé dans la communauté Python. Ce langage s'intègre dans le projet Docutils qui s'inspire d'outils tels que Javadoc. La compilation des sources est réalisée sous Sphinx, un paquet Python permettant de générer une documentation HTML ainsi qu'un ouvrage PDF via LaTeX.