MP2I

Laurent Vercouter

PRÉPAS SCIENCES

COLLECTION DIRIGÉE PAR **BERTRAND HAUCHECORNE**

INFORMATIQUE

NOUVEAUX PROGRAMMES !

- Cours complet et détaillé
- Nombreux exemples
- Vrai/Faux
- Exercices
- Corrections et commentaires

ellipses

PRÉPAS SCIENCES
collection dirigée par Bertrand Hauchecorne

Informatique
MP2I



Laurent VERCOUTER
Professeur des universités en informatique
à l'INSA Rouen Normandie

ellipses

COLLECTION
PRÉPAS SCIENCES

Retrouvez tous les titres de la collection et des extraits sur www.editions-ellipses.fr

Les macros de cet ouvrage ont été réalisées par Nicolas Nguyen en LaTeX.

Les notices culturelles « un scientifique » et « un peu d'histoire »
des pages de titre des chapitres ont été rédigées par Bertrand Hauchecorne.

ISBN 9782340-048669
© Ellipses Édition Marketing S.A., 2021
8/10 rue la Quintinie 75015 Paris

DANGER
LE
PHOTOCOPILLAGE
TUE LE LIVRE

Le Code de la propriété intellectuelle n'autorisant, aux termes de l'article L. 122-5.2° et 3°a), d'une part, que les « copies ou reproductions strictement réservées à l'usage privé du copiste et non destinées à une utilisation collective », et d'autre part, que les analyses et les courtes citations dans un but d'exemple et d'illustration, « toute représentation ou reproduction intégrale ou partielle faite sans le consentement de l'auteur ou de ses ayants droit ou ayants cause est illicite » (art. L. 122-4).
Cette représentation ou reproduction, par quelque procédé que ce soit constituerait une contrefaçon sanctionnée par les articles L. 335-2 et suivants du Code de la propriété intellectuelle.

www.editions-ellipses.fr

Avant-propos

La nouvelle classe préparatoire MP2I est avant tout destinée à former des spécialistes en informatique théorique. L'ouvrage de la collection *Prépas sciences* que vous avez entre les mains est spécifique à cet enseignement et vous permettra d'être à l'aise dans cette formation.

En effet, réussir en classes préparatoires nécessite d'assimiler rapidement un grand nombre de connaissances, mais surtout de savoir les utiliser, à bon escient, et les rendre opérationnelles au moment opportun. Bien sûr, l'apprentissage du cours de votre professeur jour après jour est indispensable. Cependant, on constate que pour beaucoup, c'est loin d'être suffisant. Combien d'entre vous ont bien appris leur cours et pourtant se trouvent démunis lors d'un devoir, et plus grave, le jour du concours. Cette collection a été conçue pour répondre à cette difficulté.

Les chapitres correspondent en gros, à une semaine de cours ; leur contenu suit strictement le programme et leur structure est identique.

- **Le cours complet** rassemble tous les résultats à savoir. Il vous permet d'accéder à une connaissance approfondie des notions. Sa relecture est indispensable avant un devoir en classe et a fortiori avant l'examen. Les nombreuses situations données en exemple, en particulier les programmes, vous initient aux techniques utiles pour résoudre les exercices classiques. Elles éclairent et illustrent le cours.
- **La partie vrai/faux** vous permet de tester votre recul par rapport au programme, vous révèle les mauvais réflexes à corriger et vous met en garde contre les erreurs classiques.
- **Les exercices** sont incontournables pour assimiler le programme et pour répondre aux exigences du concours.
- **Les corrigés**, rédigés avec soin et de manière exhaustive, sont souvent des programmes commentés et présentent une réponse parmi une multitude de possibles.

Ainsi l'ouvrage d'informatique, comme ceux de maths, de physique-chimie et de SII, vous accompagneront tout au long de l'année et vous guideront dans votre cheminement **vers la réussite aux concours**.

Bertrand Hauchecorne

Sommaire

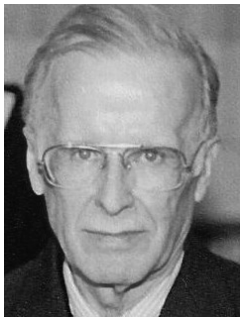
1. Méthodes de programmation 1
2. Concepts fondamentaux de la programmation impérative 25
3. Structures de données 63
4. Concepts fondamentaux de la programmation fonctionnelle 105
5. Récursivité et induction 127
6. Logique propositionnelle 141
7. Algorithmique 163
8. Gestion des ressources machine 199
9. Bases de données 223

Index 255

Chapitre 1

Méthodes de programmation

UN SCIENTIFIQUE



Après des études de chimie, puis de médecine, **John Backus** (1924-2007) devient technicien radio. Ce n'est que tardivement qu'il s'intéresse aux mathématiques puis entre à IBM en 1950.
En novembre 1954, il présente le langage Fortran puis participe à l'élaboration du langage de programmation ALGOL. Par la suite, il participe à l'élaboration de langage de programmation purement fonctionnels. Ses travaux sont récompensés par le prix Turing obtenu en 1977.

Un peu d'histoire

Lors du tout début de l'informatique, dans l'immédiat après-guerre, on travaillait directement en langage machine ce qui demandait une connaissance technique très spécifique.
Arrivé à IBM en 1950, John Backus est confronté à la difficulté de noter les nombres. Il élabore la syntaxe et le fonctionnement d'un langage intermédiaire entre la machine et le programmeur dont il allège la tâche en créant le Fortran.
De très nombreux langages suivirent très rapidement comme le COBOL en 1959. Dans les années 1970 apparaît la programmation structurée et les langages de programmation s'adaptent aux différents besoins qui apparaissent. Ils acquièrent alors de nouvelles potentialités grâce à la vitesse toujours plus grande des processeurs. Le développement d'Internet nécessite la création de nouveaux langages spécifiques.

Objectifs

- Comprendre les problématiques du développement logiciel :
  - algorithme, programme ;
  - paradigme de programmation ;
  - méthodologie de développement ;
- Comprendre la compilation et l'exécution d'un programme :
  - langage compilé ou interprété ;
  - messages d'erreur ;
  - tests ;
- Évaluer la performance d'un algorithme :
  - terminaison, correction ;
  - complexité algorithmique ;
- Comprendre la représentation numérique de l'information.

Cours

Développement logiciel

Un logiciel est un programme ou un ensemble de programmes qui peut être exécuté par un ordinateur pour réaliser des tâches automatiquement. Pour cela, un programme déroule une suite d'instructions qui considèrent en entrée des données et produisent en sortie d'autres données. Dans un premier temps, il est nécessaire de bien distinguer les termes *programme*, *algorithme* et *langage de programmation*.

Un **algorithme** est une méthode de résolution d'un problème. Il définit une suite d'instructions permettant d'aboutir à une solution au problème. Chaque instruction y est décrite de manière précise, non ambigüe, avec des données d'entrées et de sortie bien identifiées mais l'algorithme reste à un niveau abstrait, indépendant de son implémentation dans un langage de programmation. La notion d'algorithme pour résoudre des problèmes est d'ailleurs largement antérieure à l'apparition des premiers ordinateurs, le terme "algorithme" venant d'une latinisation du nom d'un mathématicien perse du IX^e siècle, Al-Khwârizmî, qui a recensé et classé des algorithmes de résolution de problèmes mathématiques.

Un **programme** est un ensemble d'instructions exprimées dans un langage interprétable par un ordinateur pour qu'il puisse les réaliser. Il peut contenir l'implémentation d'algorithmes dont la suite d'instructions abstraites est alors traduite dans le langage utilisé par le programme. L'**exécution** du programme consiste à demander à un ordinateur de réaliser la suite d'instructions qui le compose.

Un **langage de programmation** est une notation définissant un ensemble d'éléments et de règles syntaxiques pour écrire le code **source** d'un programme. Un langage de programmation peut être vu comme une étape intermédiaire entre le langage machine, très difficile à comprendre pour un programmeur humain, et une description abstraite d'algorithmes qu'un ordinateur ne peut pas interpréter. Le code source d'un programme va ensuite être traduit en langage machine lors d'une phase de compilation ou d'interprétation (voir plus loin) pour pouvoir être exécuté.

Deux langages de programmation sont utilisés dans cet ouvrage pour les codes sources des programmes présentés et pour les exercices. Il s'agit des langages C et OCaml. Pour la manipulation de bases de données abordée dans le chapitre 9, le langage SQL est utilisé. Les codes sources des programmes du livre sont téléchargeables à l'adresse :

`https://github.com/lvercouter/INFO_MP2I`

Paradigmes de programmation

Il existe de nombreux langages de programmation qui peuvent être classés par la manière dont ils abordent l'écriture du code source d'un programme et par les concepts qu'ils manipulent. Ces familles de langages s'appellent des **paradigmes de programmation**.

Le paradigme de la programmation **impérative** est le plus répandu. Il consiste en l'écriture d'instructions qui s'exécutent en séquence, c'est-à-dire les unes après les autres. L'exécution d'un programme se déroule en exécutant sa première instruction, puis quand son exécution est terminée

exécute la suivante et ainsi de suite jusqu'à la fin du programme. En plus de l'exécution séquentielle, la programmation impérative repose sur trois concepts principaux :

- l'opérateur d'**affectation** utiliser pour modifier le contenu d'une variable ;
- les **instructions conditionnelles** qui évaluent la valeur de vérité d'une condition et en fonction de cette évaluation exécute ou non un bloc d'instructions ;
- les **instructions itératives** qui répètent l'exécution d'un même bloc d'instructions jusqu'à ce qu'une condition soit atteinte.

Une sous-famille de la programmation impérative est le paradigme de la programmation **structurée**. Il reprend les concepts de la programmation impérative en y ajoutant une décomposition du code source en plusieurs sous-programmes. Cette décomposition permet l'écriture de blocs d'instructions plus petits, plus ciblés sur une tâche précise et améliore la lisibilité et la modularité du code. De nombreux langages de programmation appartiennent au paradigme de la programmation structurée, comme le C utilisé pour les exemples de cet ouvrage, le C++, Pascal, Ada, BASIC, Fortran...

Le paradigme de la programmation **fonctionnelle** préconise une approche déclarative dans l'écriture d'un programme. Elle repose sur l'écriture de fonctions mathématiques et aborde la résolution d'un problème comme une composition de fonctions. La programmation fonctionnelle n'admet pas l'opération d'affectation. Le passage de valeurs d'une fonction à une autre n'utilise donc pas de variables mais une imbrication de fonctions pour que le résultat produit par l'évaluation de certaines fonctions soit pris en entrée d'autres fonctions. Parmi les langages de programmation fonctionnelle, on retrouve, entre autres, le Lisp (premier langage de ce paradigme), Scheme, Haskell, Scala ou OCaml (utilisé dans les exemples de cet ouvrage).

Le paradigme de la programmation **orientée-objet** repose sur la décomposition d'un programme en plusieurs entités appelées *objets*. Chaque objet regroupe à la fois des variables qui lui sont locales et des méthodes de traitement. La définition d'objets favorise la modularité dans l'écriture d'un programme par des objets dont on va encourager la cohésion dans leur contenu pour qu'un objet soit focalisé sur un concept ou une entité et définisse les données qui le caractérise ainsi que les méthodes qui s'y applique. Parmi les langages de programmation orientée-objet on retrouve Smalltalk, Ada, C++, Java, Python...

Remarque : Un langage de programmation ne suit pas forcément exclusivement un paradigme de programmation. Certains langages fournissent des instructions appartenant à plusieurs paradigmes de programmation de façon à ce que le programmeur puisse choisir l'approche qui convient le mieux au problème traité.

Méthodologie de développement logiciel

Pour répondre à un besoin donné, il est possible d'écrire un programme ou un logiciel de nombreuses manières différentes mais toutes ne sont pas de même qualité. Ils peuvent différer par leur temps d'exécution, par le fait qu'ils marchent tout le temps ou provoquent parfois des erreurs, par la possibilité de ré-utiliser certaines parties pour d'autres tâches et ne pas avoir à ré-écrire complètement un autre programme... Bertrand Meyer [1] identifie un certain nombre de facteurs de qualité pour évaluer un logiciel. Parmi ceux-ci, les facteurs externes représentent les qualités d'*utilisation* du logiciel. Il s'agit principalement de :

- la **correction** qui est la faculté du logiciel à bien réaliser la tâche pour laquelle il est conçu ;

1. "Conception et programmation orientées objet", B. Meyer, eds. Eyrolles, 5ème tirage, 2011.

- la **robustesse** qui est la faculté du logiciel à réagir de manière appropriée en présence de conditions anormales ;
- l'**extensibilité** qui est la facilité de modification du logiciel pour de nouveaux besoins ;
- la **réutilisabilité** qui est la capacité de certains éléments logiciels à servir à la construction de logiciels différents ;
- la **compatibilité** qui est la facilité de combiner le logiciel avec d'autres ;
- l'**efficacité** qui est la capacité du logiciel à minimiser les ressources (temps, calcul, ...) nécessaire à son exécution ;
- la **portabilité** qui est la capacité du logiciel à être utilisé dans différents environnements matériels ou logiciels ;
- la **facilité d'utilisation** qui est la faculté du logiciel à être pris en main par ses utilisateurs.

Les facteurs *internes* représentent les qualités attendues pour les programmeurs. Certains facteurs comme la réutilisabilité ou la portabilité sont à la fois externes et internes car il est souhaitable pour un programmeur que l'effort produit pour écrire un programme puisse aussi contribuer à d'autres logiciels ou dans d'autres environnements. Deux autres principaux facteurs internes peuvent être évoqués ici :

- la **lisibilité** qui est la facilité de compréhension du code source ;
- la **modularité** qui est la pertinence de la décomposition du code source en entités relativement petites, ré-utilisables et à forte cohésion sémantique.

La prise en compte de tous ses facteurs nécessite une grande rigueur lors de l'écriture des programmes d'un logiciel. Le suivi d'une *méthodologie* de développement logiciel permet de traiter les problèmes séparément, par étapes, et de se focaliser sur la rigueur nécessaire à chacune d'entre elles. Il existe différentes méthodes de développement proposées dans le domaine du *génie logiciel*. L'étude de ces méthodes n'entre pas dans le programme de cet ouvrage mais nous pouvons lister ici les principales étapes que l'on retrouve fréquemment :

- la phase d'**analyse** lors de laquelle les besoins fonctionnels du logiciel sont recensés, avec le client ou l'utilisateur final du logiciel, pour identifier de manière précise et non ambigüe ce que doit réaliser le logiciel ;
- la phase de **conception** lors de laquelle l'architecture globale du logiciel est définie pour identifier différents modules, leurs relations, les structures de données à manipuler, ... Des langages de modélisation, tels que UML ou OMT, sont souvent utilisés lors de la conception globale du logiciel. Une conception détaillée peut aussi intervenir par la définition d'algorithmes ;
- la phase d'**implémentation** lors de laquelle le ou les programmeurs écrivent dans un langage de programmation le code source des programmes ;
- la phase de **tests** lors de laquelle le bon fonctionnement des programmes est testé, notamment sur leurs facteurs de correction et de robustesse ;
- la phase de **recette** lors de laquelle le logiciel est livré pour être utilisé.

Même dans la réalisation de programmes de taille assez modeste, il est important de procéder par ses étapes. Si les besoins sont mal exprimés ou mal compris, la réalisation ne correspondra pas aux vraies attentes. Si la phase de conception est omise, le programmeur se retrouve à définir l'architecture de son programme en même temps qu'il écrit avec de très fortes chances pour que celle-ci ne soit pas adaptée. Ces deux exemples illustrent le risque d'avoir besoin de revenir en arrière, d'avoir écrit des parties inutiles, abandonnées résultant en une perte de temps.

Lisibilité, utilisabilité

Parmi les facteurs internes de qualité d'un logiciel, la lisibilité a une place importante car elle va influer sur d'autres facteurs tels que l'extensibilité ou la réutilisabilité. La lisibilité correspond à la facilité à comprendre un programme. On peut distinguer deux niveaux : la compréhension d'un code source pour des programmeurs sur ce code et la compréhension de ce que fait un code exécutable et comment l'utiliser.

Dans le premier cas, il convient d'adopter de bonnes pratiques, dès l'écriture du code source des premiers programmes, qui facilite leur lisibilité. Une première pratique consiste à insérer des **commentaires** dans le code source. Des symboles, dépendant du langage de programmation, servent à définir des zones de commentaires qui ne seront pas pris en compte lors de la compilation ou de l'interprétation du code source. De cette manière, il est possible d'écrire un texte libre pour expliquer une partie un peu compliquée du code source. Il faut trouver un bon équilibre dans la quantité de commentaires et les consacrer aux parties de code non triviales. Avoir trop de commentaires alourdit inutilement le code source mais trop peu de commentaires peut rendre des parties de code difficiles à comprendre. Un piège à éviter est que les commentaires ne doivent pas paraphraser le code. Ils doivent aider à comprendre la nature des tâches réalisées. L'exemple ci-dessous est un commentaire qui paraphrase l'instruction qui suit (le code est en langage C où une ligne de commentaires est indiquée en commençant par `//`) :

```
// c prend comme valeur 2 * 3.14159 * r
c = 2 * 3.14159 * r;
```

Un meilleur exemple de commentaire pour cette ligne est le suivant qui explique pourquoi cette opération est réalisée :

```
// c est la circonférence du cercle de rayon r
c = 2 * 3.14159 * r;
```

Une deuxième bonne pratique est l'**indentation** du code. L'indentation consiste à insérer des tabulations ou des espaces en début de ligne pour que tout le code ne soit pas aligné à gauche mais que des blocs d'instructions soient décalés. Cela permet de visualiser très facilement la structure du code et la manière dont les blocs d'instructions vont être emboîtés dans d'autres instructions. L'exemple ci-dessous montre une fonction en C calculant le factoriel d'un nombre sans indentation.

```
1 int factoriel(int n) {
2 int r = 1;
3 for (int i = n; i > 0; i--)
4 r = r * i;
5 return r;
6 }
```

Une version indentée du même code est donnée ci-dessous.

```
1 int factoriel(int n) {
2   int r = 1;
3   for (int i = n; i > 0; i--)
```

```
4       r = r * i;
5     return r;
6   }
```

Dans ce second exemple, on voit l'intérêt des tabulations ajoutées en début de ligne pour comprendre en un coup d'oeil comment sont structurées les instructions. Les lignes 2 à 5 sont décalées par rapport aux première et dernière ligne pour montrer qu'elles sont dans la fonction `factoriel`. La ligne 4 est de nouveau décalée pour montrer que cette instruction est dans la boucle `for` de la ligne précédente. Par contre la ligne suivante est au même niveau que la boucle `for` pour montrer qu'elle est en dehors des instructions de cette boucle et s'exécutera après. Vous pourrez revenir sur ces codes après lecture du chapitre 2 pour comprendre leur fonctionnement. Dans la plupart des langages (à l'exception notable de Python), l'indentation n'a aucune influence sur la compilation ou l'interprétation du code source. Son rôle est uniquement de rendre le code source plus lisible. Cela veut dire aussi que rien ne vous empêche de faire une mauvaise indentation qui rendrait le code plus illisible !

Une dernière bonne pratique pour la lisibilité du code source est d'utiliser des **noms explicites** pour les identifiants de variables et de fonctions. Parfois le simple choix d'un nom précis permet tout de suite de comprendre ce que fait une fonction ou la nature d'une valeur stockée dans une variable. Dans l'exemple de code donné précédemment, le fait de nommer la fonction `factoriel` permet de comprendre intuitivement ce qu'elle fait. Si cette fonction avait été simplement appelée `f` cela aurait été plus obscur.

La lisibilité d'un programme s'évalue aussi du point de vue de son utilisation. Dans ce cas, on parle de la **documentation** d'un programme. La documentation ne va pas expliquer les instructions du code source mais plutôt comment utiliser ce qu'il fournit d'un point de vue extérieur. Si le programme est un module destiné à être utilisé par d'autres programmes, la documentation explique le contenu du module et, pour les fonctions, ce qu'elles font, quelles sont leurs entrées et quelles sont leurs sorties. Une documentation est aussi nécessaire pour des programmes exécutables pour expliquer comment ils doivent être installés, configurés puis utilisés.

■ Compilation vs interprétation

Le code source d'un programme ne peut pas être directement exécuté par un ordinateur. Il doit être traduit dans le langage de la machine. Selon le langage de programmation utilisé, cette traduction se fait de deux manières différentes, par compilation ou par interprétation du code source.

Langage compilé

Les langages compilés utilisent un programme appelé **compilateur**. Il prend en entrée un fichier contenant un code source et produit en résultat un fichier appelé *objet* contenant les instructions du programme en langage machine, c'est-à-dire qu'elles peuvent être exécutées par l'ordinateur et le système d'exploitation pour lesquels elles ont été compilées. La création d'un programme exécutable est souvent réalisée dans une deuxième étape appelée **édition de liens** par l'assemblage de plusieurs fichiers objets. La figure 1.1 illustre ce processus de compilation.

Un compilateur est propre à un langage de programmation et à un environnement informatique, matériel et logiciel. Si j'écris un code source en C, je dois utiliser un compilateur C pour

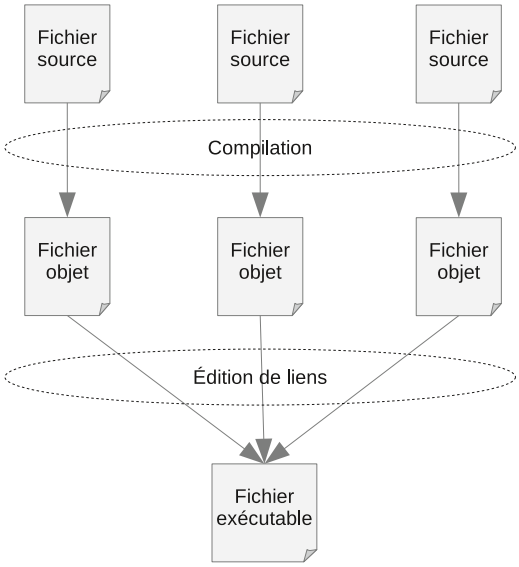


FIGURE 1.1 – Processus de compilation

mon système d'exploitation et pour le type de processeur de ma machine. Le programme exécutable résultant de la compilation pourra être exécuté dans ce même environnement informatique mais il ne sera pas portable sur d'autres environnements. Par exemple, si vous compilez un programme en utilisant le système d'exploitation Linux, il ne pourra pas être exécuté sous Windows (et inversement).

La phase de compilation analyse le code source selon les règles d'écriture du langage de programmation. Si les règles syntaxiques ne sont pas respectées, la compilation va échouer et des messages d'erreurs indiquent les raisons de cet échec. Dans ce cas les fichiers objets et exécutables ne sont naturellement pas créés.

S'il n'y a pas d'erreur à la compilation, les fichiers objets et exécutables sont créés. L'exécution du programme se fait en lançant directement le programme exécutable. La figure 1.2 montre une copie d'écran d'un terminal, sous Linux, dans lequel le fichier source `factoriel.c` est compilé, puis le programme exécutable obtenu exécuté.

Le contenu du répertoire est d'abord listé par la commande `ls` montrant qu'il n'y a que le fichier `factoriel.c`. La compilation est ensuite réalisée avec le compilateur `gcc` (l'utilisation de `gcc` est expliquée dans le chapitre 2 page 27). Elle produit le fichier exécutable `factoriel` qui apparaît maintenant après une nouvelle exécution de la command `ls`. `factoriel` est ensuite exécuté et l'affichage du texte demandant un nombre puis calculant son factoriel sont le résultat de l'exécution des instructions du programme.

Les deux langages utilisés dans cet ouvrage, `C` et `OCaml` sont des langages compilés.

```
INFO_MP2I:chap1 $ ls
factoriel.c
INFO_MP2I:chap1 $ gcc -o factoriel factoriel.c
INFO_MP2I:chap1 $ ls
factoriel       factoriel.c
INFO_MP2I:chap1 $ ./factoriel
De quel nombre voulez-vous le factoriel ? 5
5! = 120
INFO_MP2I:chap1 $
```

FIGURE 1.2 – Exemple de compilation et d'exécution

Langage interprété

Les langages interprétés ne reposent pas sur une phase de compilation préalable à l'exécution d'un programme. Ils interprètent les instructions du code source pendant son exécution. Pour cela, ils utilisent un programme appelé **interpréteur** qui lit un code source, analyse chacune des instructions les unes après les autres et, si elles sont écrites sans erreur syntaxique, les traduit en langage machine et les exécute. C'est ici l'interpréteur qui est propre à un langage de programmation et à un environnement matériel et logiciel. Le programme n'existe que dans son code source et peut être exécuté par des interpréteurs pour son langage quelque soit le système d'exploitation et type de processeur. Par exemple le langage Python est un langage interprété et la figure 1.3 montre une copie d'écran de l'interprétation d'un code source Python qui propose un calcul de factoriel comme précédemment.

```
INFO_MP2I:chap1 $ python factoriel.py
De quel nombre voulez-vous le factoriel ?
5
(5, '! = ', 120)
INFO_MP2I:chap1 $
```

FIGURE 1.3 – Exemple d'interprétation d'un code Python

L'interpréteur dans cet exemple est `python3` et le code source du programme est dans le fichier `factoriel.py`.

Chacune des approches présente des avantages et des inconvénients. Les langages interprétés ont notamment l'avantage d'être portables. Le programme n'existant que dans son code source, il est utilisable sous cette même forme dans tout système d'exploitation, sous réserve qu'il y existe un interpréteur pour son langage de programmation. À l'inverse les programmes écrits dans un langage compilé ne sont pas portables mais ils ont pour avantage d'offrir des temps d'exécution plus rapides. En effet la phase de compilation a créé un fichier exécutable en langage machine et ses instructions peuvent être directement exécutées, alors que dans le cas d'un langage interprété la traduction se fait pendant l'exécution.

Erreurs et bugs

Pendant l'écriture d'un programme, il est très fréquent d'obtenir des erreurs pendant sa compilation ou pendant son exécution. Un compilateur analyse un code source pour le traduire en langage machine mais il faut pour cela que les instructions respectent scrupuleusement les règles

syntaxiques du langage. Si ce n'est pas le cas, le compilateur échoue dans la création d'un programme exécutable et affiche des **messages d'erreur**. Ces messages d'erreur sont très importants pour que le programmeur puisse comprendre ce qu'il doit corriger. Dans la plupart des compilateurs, un message d'erreur indique le fichier source, la ligne et la colonne où l'erreur s'est produite et un texte expliquant le type d'erreur. Selon le type d'erreur, ce n'est pas forcément à l'endroit indiqué qu'une correction est à faire mais cela reste dans son voisinage. La figure 1.4 montre un exemple d'erreur à l'exécution lors d'une compilation avec `gcc`.

```
INFO_MP2I:chap1 $ gcc -o factorielErr factorielErr.c
factorielErr.c:5:55: error: expected ';' after expression
  printf("De quel nombre voulez-vous le factoriel ? ")
                                                      ^
                                                      ;
1 error generated.
INFO_MP2I:chap1 $
```

FIGURE 1.4 – Exemple d'erreur à la compilation

Le message d'erreur indique ici que l'erreur s'est produite dans le fichier `factorielErr.c` à la ligne 5 et la colonne 55. Le texte qui suit indique qu'un point-virgule était attendu après l'expression recopiée ci-dessous. Le point-virgule étant le symbole terminant une instruction en C, son omission dans ce fichier est une erreur syntaxique.

Lors de la correction des erreurs, il est préférable de traiter les messages d'erreur dans leur ordre d'apparition. Parfois une première erreur fait "perdre le fil" au compilateur qui n'arrive plus à comprendre le code source et affiche d'autres messages d'erreur. Quand cela se produit la correction de premières erreurs peut faire disparaître plusieurs messages d'erreur.

La compilation peut aussi afficher des **avertissements**. Les avertissements n'empêchent seuls pas la création d'un programme principal. Ce sont des messages du compilateur sur des instructions du code source qui sont inhabituelles ou dangereuses pour la bonne exécution du programme. Ces avertissements sont aussi des informations importantes à considérer car ils peuvent être annonciateurs de problèmes à l'exécution.

Pendant l'exécution, des dysfonctionnements peuvent également intervenir. Le terme **bug** (ou bogue en version francisée) est utilisé pour désigner les erreurs à l'exécution. Un bug se traduit par un comportement anormal du programme. Dans certains cas, cela peut provoquer un arrêt brutal du programme. Dans d'autres cas, le programme continue de fonctionner mais ne réalise pas ou mal les tâches pour lesquelles il a été programmé. Les bugs sont plus difficiles à corriger que les erreurs à la compilation car il n'y a pas de message d'erreur pour les expliquer. La nature du comportement anormal et le moment où il se produit sont des indices pour diriger le programmeur vers la partie du code qui pourrait produire le bug. La cause est souvent dans la définition des algorithmes qui ne résolvent pas bien ou pas dans tous les cas les problèmes posés, ou dans le passage d'un algorithme à sa version implémentée dans le code source. La figure 1.5 montre une copie d'écran de l'exécution du programme calculant un factoriel mais qui dysfonctionne. Le code en C correspondant à ce programme mal écrit suit.

```
INFO_MP2I:chap1 $ gcc -o factorielBug factorielBug.c
INFO_MP2I:chap1 $ ./factorielBug
De quel nombre voulez-vous le factoriel ? 5
5! = 0
INFO_MP2I:chap1 $
```

FIGURE 1.5 – Exemple de bug à l'exécution

```
1 int main(int argc, char **argv) {
2   int n, factoriel;
3   printf("De quel nombre voulez-vous le factoriel ? ");
4   scanf("%d",&n);
5   factoriel = 0;
6   for (int i = 2; i <= n; i ++)
7     factoriel = factoriel * i;
8   printf("%d! = %d\n",n,factoriel);
9 }
```

On voit ici que le calcul du factoriel n'est pas correct car il indique que le factoriel de 5 est 0. L'erreur vient d'une mauvaise écriture de l'algorithme qui multiple tous les nombres entiers jusqu'à la valeur saisie au clavier (la variable `n`). Elle se situe ligne 10 par la variable `factoriel` qui est initialisée à 0 et non à 1, ce qui lui fait absorber toutes les valeurs entières suivantes.

Exceptions, tests

Un bug dans l'exécution d'un programme peut avoir des origines diverses. Cela peut être lié à une erreur dans l'écriture du programme mais cela peut aussi venir de conditions d'exécution inhabituelles, rares ou de données prises en entrée qui seraient invalides. Le principe de la **programmation défensive** consiste à considérer lors de l'écriture d'un programme toutes les situations auxquelles il peut être confronté, et surtout les plus défavorables. Lorsque des données sont fournies en entrée au programme, un type de données peut garantir la nature de l'information stockée mais le programmeur ne doit pas estimer que d'autres contraintes sont garanties, par exemple le fait que la valeur reste dans un intervalle de valeurs autorisées. Ceci est encore plus vrai quand une valeur est saisie par un utilisateur du programme. Cet utilisateur est tout à fait libre de saisir ce qu'il veut ou de faire des erreurs de frappe. Il est alors nécessaire dans le code du programme de détecter l'apparition de ces cas indésirables et d'écrire comment le programme doit se comporter le cas échéant.

L'exemple précédent sur le calcul d'un factoriel n'applique pas ce principe de programmation défensive (c'est volontaire pour conserver un exemple de code relativement simple). En effet, une valeur est demandée à l'utilisateur pour calculer son factoriel mais rien ne nous garantit qu'il va saisir un entier naturel positif, ni même qu'il ne va pas saisir un texte ne comportant pas que des chiffres. S'il donne une valeur négative, ce programme affichera quand même une valeur pour son factoriel qui sera 0 (ou 1 si le bug est corrigé), ce qui est mathématiquement faux.

Exceptions

Certains langages de programmation introduisent le concept d'**exceptions** pour détecter et réagir à des situations inhabituelles. Pour cela le programmeur doit évaluer les données prises en entrée ou d'autres conditions d'exécution et si un problème est détecté, crée et émet une exception.

L'effet est de sortir de l'exécution courante du code pour remonter à un bloc d'instructions dédié au traitement de l'exception.

OCaml fait partie des langages disposant d'un mécanisme de gestion d'exceptions. Une exception est une valeur avec un type de donnée spécifique `exn`. Dans une fonction, une exception est levée par l'instruction `raise` suivie de la valeur de l'exception. Il existe en OCaml un ensemble d'exceptions prédéfinies pour des problèmes courants. La fonction `division` suivante illustre un exemple d'utilisation de l'exception `Division_by_zero`.

```
1 let division x y = if y = 0 then raise Division_by_zero else x/y
```

L'appel à cette fonction avec un diviseur de valeur 0 émettra l'exception `Division_by_zero` et ne renverra pas d'autre valeur.

```
# division 5 0;;
Exception: Division_by_zero.
```

Les exceptions peuvent aussi être interceptées pour y associer des instructions à exécuter sans que le programme s'interrompe. Par exemple, quand une fonction en appelle une autre et que cette dernière lève une exception, la fonction appelante peut intercepter l'exception, la traiter et poursuivre son exécution. En OCaml, l'interception des exceptions utilise les mots-clés `try ... with ...`. Les exceptions qui sont levées entre les termes `try` et `with` sont interceptées si elles correspondent à un motif défini après le `with`. La gestion d'exceptions suit le mécanisme de filtrage de motifs d'OCaml expliqué dans le chapitre 4 page 115. L'exemple suivant est une fonction `affiche_division` qui affiche dans le terminal le résultat d'une division. Si aucune exception n'est levée, elle affiche le résultat de la division par l'instruction `print_int`. Si une exception est levée et correspond au type `Division_by_zero` indiqué dans le motif, c'est l'instruction associée `print_string` qui sera exécutée avant de poursuivre l'exécution du programme.

```
1 let affiche_division x y =
2         try print_int  (division x y) with
3                 Division_by_zero -> print_string "Division impossible"
```

Un appel à `affiche_division` dans les deux cas est donné ci-dessous.

```
# let affiche_division x y =
        try print_int (division x y) with
            Division_by_zero -> print_string "Division impossible";;
val affiche_division : int -> int -> unit = <fun>
# affiche_division 5 2;;
2- : unit = ()
# affiche_division 5 0;;
Division impossible- : unit = ()
```

Si une fonction appelante n'intercepte pas une exception levée, celle-ci est propagée de la même manière que si elle avait été levée à l'endroit où elle apparaît. Cette fonction appelante interrompt donc son exécution et lève l'exception au moment où elle-même a été appelée. La propagation se poursuit jusqu'à ce que l'exception soit interceptée ou, si elle n'est jamais interceptée, interrompt le programme quand elle est remontée au programme principal.

Assertions

De manière générale, pour éviter l'apparition de bugs pendant l'exécution d'un programme que ce soit lié à la validité des données ou à des erreurs de programmation, il faut réaliser des **tests** dans différentes conditions. Bien entendu, il n'est souvent pas réaliste de tester toutes les situations ou toutes les valeurs possibles des données prises en entrée. Il est cependant possible d'identifier les types de scénarios qui peuvent se produire et de tester sur un cas représentatif pour chacun des scénarios. Pour cela, un programme de tests est écrit à côté du programme à tester, importe les fonctions à tester et applique une série de tests pour couvrir l'ensemble des types de scénarios. Les instructions d'**assertion** sont utilisés dans le programme de test pour vérifier qu'une condition est valide. Si la condition est valide l'exécution du programme de test continue et on considère que le test réalisé par l'assertion est réussi. Si la condition est fausse, le programme de test s'arrête en indiquant que ce test n'est pas concluant. En C l'écriture d'assertion utilise le module `assert.h` qui propose une fonction `assert` pour réaliser un test. L'exemple ci-dessous montre un programme de test en C pour la fonction `puissance` (voir chapitre 2 page 48).

```
1  #include<assert.h>
2  #include"puissance.h"
3
4  int main(int argc, char**argv) {
5    int p = puissance(3,2);
6    assert(p == 9);
7    p = puissance(5,0);
8    assert(p == 1);
9    p = puissance(-4,2);
10   assert(p == 16);
11   p = puissance(-4,3);
12   assert(p == -64);
13 }
```

Cette fonction, qui calcule la puissance d'un nombre pour un exposant positif, est testée dans différents cas. Le programme de test vérifie d'abord que la valeur renvoyée pour 3^2 est bien 9. Ensuite il vérifie qu'un nombre à la puissance 0 vaut 1. Puis, il teste deux cas de puissances d'un nombre négatif, avec un exposant pair puis impair. Les conditions dans les fonctions `assert` effectue le test entre la valeur renvoyée et la valeur attendue.

Il est utile d'écrire des programmes de tests pour s'assurer du bon fonctionnement des principales fonctions d'un programme. Cela permet de les tester après leur écriture mais aussi plus tard dans l'écriture du programme lorsque certaines modifications pourraient impacter des fonctions précédemment écrites. Exécuter les programmes de tests assure une vérification instantanée que le fonctionnement reste correct. Certaines méthodologies, tel que l'*Extreme Programming* ou le *Test Driven Development* préconise un recours systématique à des programmes de tests et même de les écrire avant d'écrire les fonctions à tester. De cette manière ils représentent une sorte de cahier des charges du comportement attendu des fonctions à tester.

■ Performances d'un algorithme

Terminaison

La **terminaison** d'un algorithme est une propriété signifiant qu'il s'arrêtera après exécution d'un nombre fini d'opérations. La terminaison d'un algorithme est garantie si :

- il définit un nombre fixe d'opérations à exécuter ;
- il est prouvé que les boucles et appels récursifs ne sont pas infinis.

Le premier cas est trivial. Il correspond à un algorithme dans lequel chaque instruction n'est exécutée au plus qu'une fois. Ce nombre d'instructions étant fini, l'algorithme terminera toujours.

Dans le second cas, une manière de prouver la terminaison est d'utiliser un **variant de boucle**. Un variant de boucle est une expression, exprimée dans la condition d'une boucle ou à partir de variables issues de cette condition, qui est un entier positif et qui décroît strictement à chaque itération. Par exemple, si un algorithme comporte une boucle avec un indice i, initialisé à une valeur strictement positive, être décrémenté à chaque itération et qui provoque la sortie de la boucle quand il n'est plus strictement positif, cet indice i peut servir de variant de boucle. Dans le cas où l'on aurait une boucle avec un indice i qui est incrémenté jusqu'à ce qu'il dépasse une valeur n (comme dans l'exemple du programme vu précédemment calculant le factoriel d'un nombre), un variant de boucle peut être l'expression $n - i$.

Correction

La **correction** d'un algorithme est une propriété qui est vraie si :

- il termine ;
- pour toute entrée, il produit le résultat escompté.

La preuve de la correction d'un algorithme utilise un **invariant de boucle**. Un invariant de boucle est une expression qui :

- est vraie avant d'entrer la boucle ;
- est vraie à chaque itération de la boucle ;
- correspond à ce qui est attendu en fin de boucle.

Cette démarche s'apparente à une démonstration par récurrence en désignant chaque itération par un rang k, en prouvant que l'invariant est vraie quand $k = 0$ et qu'il reste vrai de manière générale pour $k+1$. Un point difficile pour prouver la correction d'un algorithme est de définir cet invariant.

Prenons un exemple sur le programme de calcul d'un factoriel de la page 7. Désignons les valeurs des variables n, r et i à l'itération k par n_k, r_k et i_k. L'algorithme implémenté par cette fonction construit son résultat en partant du nombre donné en entrée pour le multiplier avec tous ses prédécesseurs strictement positifs à chaque itération. On peut donc considérer que si l'on multiplie un résultat intermédiaire avec le factoriel du prochain nombre qui sera considéré, on obtient la valeur attendue comme résultat final. Ceci s'exprime par l'invariant de boucle :

$$r_k \times i_k! = n!$$

L'exécution d'une itération de la boucle fait évoluer les variables de la manière suivante :

$$r_{k+1} = r_k \times i_k$$

$$i_{k+1} = i_k - 1$$

L'invariant est vrai à l'entrée de la boucle car $r_0 = 1$, $i_0 = n$ et donc

$$r_0 * i_0! = 1 \times n! = n!$$

L'invariant prouve la correction du programme en étant vrai à chaque itération car

$$r_{k+1} \times (i_{k+1}!) = r_k \times i_k \times (i_k - 1)! = r_k \times i_k! = n!$$

Complexité algorithmique

L'efficacité d'un algorithme est évaluée en fonction de son coût pour la résolution du problème posé. Ce coût est représenté par la **complexité algorithmique**. Selon la nature de la ressource prise en compte dans le coût, on distingue différentes complexités, notamment :

- la complexité en mémoire qui représente la quantité de mémoire allouée lors de l'exécution de l'algorithme ;
- la complexité en temps qui estime la quantité d'instructions à exécuter pour l'exécution complète de l'algorithme.

Nous nous intéressons ici plus précisément à la complexité en temps qui est la plus utilisée pour évaluer la performance des algorithmes. Le temps réel d'exécution d'un programme dépend de nombreux facteurs et notamment des supports matériels et logiciels de son exécution. Plus le processeur de la machine est puissant, plus l'exécution du programme sera rapide. Le langage de programmation choisi pour implémenter l'algorithme a aussi une influence, par exemple un langage compilé sera probablement plus rapide qu'un langage interprété.

La mesure du temps réel d'exécution d'un programme n'est donc pas un indicateur pertinent pour évaluer la complexité en temps de l'algorithme abstrait qu'il implémente. Cette complexité est évaluée *relativement à la taille du problème traité*. N désigne la taille du problème, par exemple le nombre de données prises en entrée de l'algorithme. La complexité en temps est exprimée avec une fonction sur N. L'analyse de la complexité en temps d'un algorithme consiste alors en l'étude du déroulement de ses instructions, notamment lors de boucles ou d'appels récursifs dont le nombre d'instructions exécutées peut dépendre de N. Il existe trois cas de complexité en temps :

- la complexité dans le meilleur des cas qui estime le temps d'exécution dans la situation la plus favorable ;
- la complexité en moyenne qui estime le temps d'exécution dans la moyenne de toutes les situations possibles ;
- la complexité dans le pire des cas qui estime le temps d'exécution dans la situation la plus défavorable.

Prenons un exemple avec un algorithme qui recherche si une valeur est présente dans une liste de N valeurs non triées. L'algorithme lit chaque valeur de la liste les unes après les autres et s'arrête quand il trouve la valeur recherchée ou quand il atteint la fin de la liste. Le meilleur des cas est la situation où la valeur recherchée est présente à la première place consultée dans la liste. Dans ce cas, le temps d'exécution est constant car indépendant de la taille de la liste. Le pire des cas est celui où la valeur n'est pas présente dans la liste car il faut consulter toutes les valeurs. Le temps d'exécution dans le pire des cas est *linéaire* en fonction de N. Enfin, on peut considérer qu'en moyenne $\frac{N}{2}$ valeurs seront consultées, ce qui reste une complexité d'ordre linéaire.

C'est la complexité dans le pire des cas qui est la plus utilisée pour évaluer la performance d'un algorithme. Elle donne des garanties sur la terminaison de l'algorithme dans un temps raisonnable en fonction de la taille du problème. Cette complexité est exprimée par le symbole $\mathcal{O}$ décrivant le comportement asymptotique d'une fonction pour borner le temps d'exécution de l'algorithme. Pour cette fonction, les facteurs constants ou de proportionnalité sont ignorés car l'objectif est essentiellement de montrer comment le temps d'exécution augmente avec la taille du problème. Dans l'exemple précédent la complexité linéaire dans le pire des cas est notée $\mathcal{O}(N)$. La complexité dans le meilleur des cas n'a pas beaucoup d'intérêt. La complexité en moyenne s'approche le plus du comportement réel de l'algorithme mais elle n'offre pas de garantie quand la situation rencontrée est défavorable.

Reprenons l'exemple de recherche d'une valeur mais en considérant cette fois que la liste est triée. Un algorithme de recherche dichotomique améliore le nombre d'instructions à exécuter en comparant la valeur recherchée à celle du milieu de la liste pour orienter la recherche sur une des deux moitiés de la liste. La recherche se poursuit en considérant la valeur au milieu de la moitié de liste restante et ainsi de suite de manière à diviser de moitié l'espace de recherche à chaque itération. La complexité algorithmique (dans le pire des cas) de cet algorithme est *logarithmique*, notée $\mathcal{O}(\log(N))$.

Le tableau 1.1 répertorie les complexités algorithmiques classiquement utilisées, avec une estimation du temps réel d'exécution si l'on suppose (de manière totalement arbitraire) que le traitement d'une donnée d'entrée s'exécute en une microseconde.

Complexité	*Type*	*Estimation à $N = 10$*	*Estimation à $N = 100$*	*Estimation à $N = 1000$*	*Estimation à $N = 10^6$*
$\mathcal{O}(1)$	constante	1 μs	1 μs	1 μs	1 μs
$\mathcal{O}(\log(N))$	logarithmique	3 μs	6 μs	10 μs	20 μs
$\mathcal{O}(N)$	linéaire	10 μs	100 μs	1 ms	1 s
$\mathcal{O}(N\log(N))$	linéarithmique	3 μs	6 μs	10 ms	20 s
$\mathcal{O}(N^2)$	quadratique	100 μs	10 ms	1 s	10 jours
$\mathcal{O}(2^N)$	exponentielle	1 ms	10^{14} siècles	...	...

TABLE 1.1 – Types de complexité algorithmique

■ Représentation numérique de l'information

Le mode de représentation des informations a longtemps été spécifique à leur nature et aux supports et outils pour les manipuler et les stocker. Pour les textes écrits, il s'agissait essentiellement de l'imprimerie ou de l'écriture manuscrite sur papiers ou autres supports inscriptibles, les sons utilisaient des chaînes de traitement analogiques, les photographies s'appuyaient sur la technique argentique, ... La révolution numérique provoquée par l'essor des technologies numériques depuis la deuxième moitié du XX^e siècle a permis, entre autres, d'unifier la manière de représenter des informations. Chaque type d'information peut être représenté sous un format numérique, permettant ainsi d'utiliser les mêmes outils pour les stocker, les traiter, les transmettre, ... Des informations de différentes natures (texte, image, son, vidéo, etc.) peuvent ainsi être stockées sur un même disque dur ou utiliser les mêmes protocoles réseaux pour être transmises.

Une information numérisée est représentée dans un format binaire. L'unité élémentaire de la représentation numérique est le **bit** qui a pour valeur 0 ou 1. Un ensemble de 8 bits est appelé un **octet**. Il y a ainsi 2^8 valeurs différentes que l'on peut représenter dans un octet. Une valeur stockée dans un ou plusieurs octets est considérée comme une donnée mais, prise seule, il n'est pas possible de lui donner un sens. Le passage d'une donnée à une information (qui elle est porteuse de sens) nécessite un modèle d'interprétation. Un modèle d'interprétation sert à analyser les valeurs binaires et à les comprendre comme une information textuelle, numérique, etc. Dans la suite de ce chapitre, nous expliquons comment sont interprétés deux types basiques d'information : les nombres et les textes.

Nombres

Valeurs entières

L'interprétation d'une donnée numérique comme une valeur entière consiste en un changement de base. Il s'agit de passer d'une représentation en base 2 à une autre base, la plupart du temps décimale. Chaque bit correspond à une valeur exprimée par une puissance de 2. Le bit de poids le plus faible (le plus à droite sur une représentation) correspond à 2^0, le suivant à 2^1, puis 2^2, et ainsi de suite jusqu'au bit de poids le plus fort qui correspond à 2^7 sur un seul octet. L'interprétation d'un octet comme un nombre entier revient à conserver les valeurs des bits qui sont à 1 et à les additionner. La figure 1.6 montre un exemple d'octet à interpréter comme un entier.

0	0	1	1	0	1	0	1

FIGURE 1.6 – Exemple d'octet

L'interprétation de cette donnée comme un entier donne la valeur 53 par le calcul suivant :

$$0 \times 2^7 + 0 \times 2^6 + 1 \times 2^5 + 1 \times 2^4 + 0 \times 2^3 + 1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0 = 53$$

La représentation numérique des informations a une conséquence importante qui limite le nombre de valeurs différentes représentables. En effet, on voit bien dans cet exemple qu'un octet ne peut accepter qu'un nombre fini de combinaisons. Avec un seul octet, seules 256 combinaisons distinctes existent et on ne pourra représenter que des nombres allant de 0 à 255 avec ce mode d'interprétation.

Pour réduire cette contrainte, le nombre d'octets utilisé pour représenter un nombre entier est classiquement de 2 ou 4 octets (cela dépend du langage de programmation). Cela étend le nombre de combinaisons possibles respectivement à 2^{16} et 2^{32} valeurs distinctes mais il y aura toujours une limite aux valeurs maximale et minimale qui sont représentables.

Le modèle d'interprétation utilisé ci-dessus correspond à un entier *non signé*, c'est-à-dire un entier naturel positif. Le modèle d'interprétation d'entiers signés utilise le bit de poids le plus fort comme **bit de signe** qui indique si l'entier est positif ou négatif. Si la valeur de ce bit est de 1, l'entier est considéré comme négatif et la valeur calculée par addition des puissances de 2 doit être multipliée par -1. La figure 1.7 donne un autre exemple d'un entier signé encodé ici sur 2 octets.

La valeur de cet entier signé est $-1 \times (2^{11} + 2^8 + 2^6 + 2^1) = -2370$.

Réels à virgule fixe

Une manière de représenter un nombre réel est d'utiliser une **virgule fixe**, c'est-à-dire que le modèle d'interprétation associe certains bits à la partie entière de la valeur réelle et d'autres bits

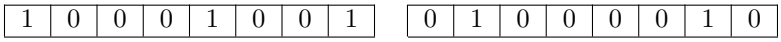


FIGURE 1.7 – Entier signé codé sur 2 octets

à la partie décimale. La virgule est considérée fixée à l'endroit où l'on passe de la partie entière à la partie décimale. Un bit de signe est aussi utilisé pour représenter les valeurs négatives. L'interprétation de la partie entière se fait de la même manière que pour les entiers, en associant à chaque bit une puissance de 2. La partie décimale suit la même logique en associant des puissances de 2 négatives à chaque bit. Le bit de poids le plus fort de la partie décimale correspond à 2^{-1}, le suivant à 2^{-2} et ainsi de suite. La valeur du nombre réel correspond à la somme des puissances de 2 pour tous les bits qui valent 1, multiplié par -1 si le bit de signe est à 1.

Le nombre fini de combinaisons possible à la même conséquence que pour les nombres entiers de limiter les valeurs maximale et minimale représentables. Il y a aussi une seconde conséquence sur la précision de la valeur réelle. Il n'est pas possible de représenter l'infinité des parties décimales possibles. Ainsi lorsqu'une valeur réelle est stockée numériquement, c'est son approximation la plus proche existant dans les valeurs représentables qui est conservée.

Flottants

Le second mode de représentation des nombres réels est l'utilisation d'une **virgule flottante**. Une représentation en virgule flottante est composée de trois parties :

- un bit de signe ;
- une **mantisse** qui est une valeur entière contenant l'ensemble des chiffres significatifs des parties entière et décimale ;
- un **exposant** qui sert à placer la virgule.

L'interprétation d'un flottant ayant un bit de signe s, une mantisse m et un exposant e consiste à calculer la valeur $s \times m \times 2^e$. L'exposant peut être positif ou négatif mais pour faciliter la comparaison entre flottants, il est stocké avec un biais pour n'avoir que des valeurs positives. Si l'exposant est stocké sur 8 bits, sa réelle valeur sera entre -126 et $+127$. Un biais de 127 est ajouté pour avoir une valeur positive mais il faudra retirer ce biais lors de l'interprétation.

La norme IEEE 754 fixe deux formats pour les flottants encodés sur 4 ou 8 octets comme indiqué dans le tableau 1.2.

Précision	*Encodage*	*Signe*	*Exposant*	*Mantisse*	*Interprétation*
Simple	32 bits	1 bit	8 bits	23 bits	$(-1)^S \times M \times 2^{(E-127)}$
Double	64 bits	1 bit	11 bits	52 bits	$(-1)^S \times M \times 2^{(E-1023)}$

TABLE 1.2 – Flottants simple et double précision

Texte

La représentation numérique d'informations textuelles s'appuie sur une table d'encodage qui fait correspondre un caractère à une valeur binaire. Il existe plusieurs tables d'encodage, il est donc nécessaire de préciser laquelle doit être utilisée lors de l'interprétation de données numériques.

La première table d'encodage largement utilisée et encore très répandue aujourd'hui est la table **ASCII** (*American Standard Code for Information Interchange*) définie dans les années 1960 pour représenter des textes anglais. Elle utilise 7 bits pour représenter chaque caractère d'un texte en incluant des codes de contrôle par exemple pour des touches du clavier qui ne correspondent pas à un caractère visuel. La table ASCII est présentée dans le tableau 1.3 en indiquant en colonne la valeur des 3 bits de poids le plus fort et en ligne celle des 4 autres bits.

	000	001	010	011	100	101	110	111
0000	NULL	DLE		0	@	P	`	p
0001	SOH	DC1	!	1	A	Q	a	q
0010	STX	DC2	"	2	B	R	b	r
0011	ETX	DC3	#	3	C	S	c	s
0100	EDT	DC4	$	4	D	T	d	t
0101	ENQ	NAK	%	5	E	U	e	u
0110	ACK	SYN	&	6	F	V	f	v
0111	BEL	ETB	′	7	G	W	g	w
1000	BS	CAN	(	8	H	X	h	x
1001	HT	EM	)	9	I	Y	i	y
1010	LF	SUB	*	:	J	Z	j	z
1011	VT	ESC	+	;	K	[	k	{
1100	FF	FS	,	<	L	\	l	|
1101	CR	GS	-	=	M	]	m	}
1110	SO	RS	.	>	N	^	n	~
1111	SI	US	/	?	O	_	o	DEL

TABLE 1.3 – Table ASCII

Les 32 premières valeurs correspondent aux codes de contrôle que l'on ne va pas détailler ici. Le premier caractère encodé a pour valeur 0100000 (32 en décimal) et correspond à un espace. On trouve ensuite un ensemble de symboles encodés entre les valeurs 33 et 47, puis une représentation textuelle des chiffres de '0' à '9'. L'ensemble des valeurs restantes encode d'autres symboles et les caractères alphabétiques de 'A' à 'Z' en majuscule et en minuscule. Le maintien de l'ordre alphabétique est intéressant pour permettre de comparer des caractères ou chaînes de caractère.

La table ASCII ayant constitué un premier standard d'encodage des caractères, elle a été et est encore très utilisée. Elle présente néanmoins l'inconvénient d'avoir été conçue pour la langue anglaise et ne comporte pas de caractères utilisés dans d'autres langues, notamment les caractères accentués. De nombreux autres tables d'encodage ont été définies pour pouvoir, entre autres, stocker numériquement des textes écrits dans des langues qui ne sont pas l'Anglais. L'espace mémoire requis pour stocker un caractère varie d'un caractère à un autre. Selon le nombre de caractères à encoder, il se peut qu'un seul octet ne permette pas de couvrir toutes les combinaisons possibles.

L'encodage UTF-8 est maintenant largement utilisé car il couvre l'ensemble des caractères du répertoire universel de caractères codés développé par l'organisation internationale de normalisation ISO. Les caractères sont codés sur un espace mémoire de 1 à 4 octets pour pouvoir couvrir le grand nombre de caractères de ce répertoire.