Introduction à l'Informatique

Bernard Boigelot
Université de Liège

2021

Table des matières

Avant-propos

Les ordinateurs sont devenus des objets indispensables de la vie quotidienne. Nous en dépendons pour nous informer, nous divertir, communiquer et travailler. En plus de cela, il existe des ordinateurs moins directement visibles employés comme composants essentiels de dispositifs plus complexes ; par exemple, une voiture moderne en comprend plusieurs dizaines. De nos jours, la plupart des ingénieurs, quelle que soit leur discipline, sont amenés à devoir réaliser ou gérer des développements informatiques.

Ce cours a pour objectif d'apprendre à utiliser un ordinateur pour résoudre des problèmes. Il comprend une introduction à l'*algorithmique*, qui étudie comment passer d'un problème à une procédure effective permettant de le résoudre, et à la *programmation*, qui vise à exprimer une telle procédure sous la forme d'un *programme* pouvant être exécuté par un ordinateur. Dans ce cours, la programmation est introduite à l'aide du langage de programmation C, qui a l'avantage d'être simple, très utilisé, et de permettre d'introduire aisément des concepts et des mécanismes possédant une portée plus générale.

Ces notes de cours sont structurées de la façon suivante :

— Le chapitre 1 introduit la matière du cours, en commençant par une brève histoire de l'informatique et en définissant les notions d'algorithme et de programme.

— Le chapitre 2 décrit les bases du langage C, en fournissant suffisamment d'éléments pour pouvoir commencer à rédiger des programmes simples.

— Le chapitre 3 est consacré à l'algorithmique. Il introduit un problème élémentaire (la recherche de nombres parfaits), et discute de la façon de le résoudre en développant des solutions de plus en plus élaborées. Des mécanismes permettant de comparer les performances des solutions obtenues et de prouver rigoureusement que ces dernières sont correctes sont également introduits.

— Le chapitre 4 définit le concept de fonction. Celui-ci permet de structurer un programme en une combinaison de fragments plus simples, et de réutiliser facilement des solutions développées pour des sous-problèmes. Ce chapitre présente ensuite le mécanisme d'exécution récursive d'une fonction, qui permet de résoudre élégamment certains problèmes

algorithmiques.

— Le chapitre 5 décrit les éléments du langage C liés à la manipulation de tableaux et de chaînes de caractères, et illustre leur utilisation dans le cadre de deux problèmes algorithmiques : le calcul de nombres premiers et le tri d'un ensemble de valeurs.

— Le chapitre 6 aborde la représentation de données structurées, et introduit la notion fondamentale de pointeur. Il présente également les mécanismes permettant d'allouer dynamiquement de la mémoire pour accommoder des données de taille variable.

— Le chapitre 7 étudie deux structures de données importantes : les piles et les files, et discute de leurs implémentations possibles. Le fait qu'une même structure puisse être implémentée de différentes façons est un mécanisme essentiel au développement de programmes possédant de bonnes propriétés de modularité.

Chacun de ces chapitres se termine par une série d'exercices, servant en partie de base aux séances de travaux pratiques.

Ce cours ne constitue qu'une première étape dans l'apprentissage de l'algorithmique et de la programmation ; sa matière est destinée à être approfondie et complétée par d'autres enseignements. En particulier, les cours de *Structures de données et algorithmes* et de *Compléments de programmation* continuent l'apprentissage de l'algorithmique et de la programmation. *Object-oriented programming* et *Programmation fonctionnelle* explorent d'autres paradigmes de programmation qui complètent celui introduit dans ce cours. *Organisation des ordinateurs* s'intéresse quant à lui aux principes de fonctionnement des machines capables d'exécuter les programmes tels que ceux développés dans ce cours ; il s'agit d'un sujet qui est ensuite approfondi dans *Computation structures*. Cette liste de cours n'est pas exhaustive.

Chapitre 1

Les ordinateurs, les algorithmes et les programmes

1.1 Introduction

Dans ce cours, nous allons apprendre à programmer un ordinateur. Avant de pouvoir expliquer ce que cela signifie exactement, la première étape est de définir précisément ce que l'on entend par "ordinateur".

1.1.1 Qu'est-ce qu'un ordinateur ?

Si l'on demande à plusieurs personnes prises au hasard d'expliquer ce qu'est un ordinateur, il est probable que la plupart d'entre elles vont évoquer le PC (*Personal Computer*, ordinateur personnel) qu'elles utilisent pour naviguer sur Internet, jouer à des jeux ou consulter leur courrier électronique. Une autre forme d'ordinateur que nous connaissons tous est bien sûr le *smartphone* dont nous dépendons pour nous divertir et communiquer. Certaines personnes vont sans doute aussi mentionner les grands systèmes informatiques que l'on trouve par exemple dans le siège central des banques, chargés de gérer les données financières relatives à un ensemble de comptes. Un autre exemple d'ordinateurs est donné par ceux qui sont chargés de résoudre des problèmes scientifiques, par exemple simuler le comportement de l'atmosphère en vue de prévoir la météo.

À côté de ces types d'ordinateurs qui sont connus du grand public, il existe d'autres formes de systèmes informatiques qui sont en fait beaucoup plus répandus, mais moins directement visibles : il s'agit des systèmes *enfouis (embedded systems)* ou *embarqués*, qui sont des ordinateurs utilisés comme composants de dispositifs plus complexes. On trouve de tels systèmes enfouis dans de nombreux objets et appareils de la vie quotidienne : télévisions, électroménager, thermo-

stats, centrales d'alarme, véhicules, dispositifs médicaux, ascenseurs, … Une voiture moderne comprend ainsi plusieurs dizaines d'ordinateurs dont la fonction va du contrôle en temps-réel des paramètres du moteur à la gestion des phares et des essuie-glaces.

Une question naturelle consiste donc à déterminer quel est le point commun entre tous les types d'"ordinateurs" que nous venons de citer. Ce point commun est qu'ils sont tous amenés à *traiter des données*. Par exemple, un ordinateur employé pour prévoir la météo est alimenté par un ensemble de données fournissant des informations de température, de pression atmosphérique et d'humidité en un grand nombre d'endroits, et sera chargé de traiter ces données en leur faisant subir un certain nombre de calculs afin de déterminer la meilleure estimation possible de l'évolution de ces paramètres.

Cette notion de traitement de données possède une portée plus générale. Le fait de résoudre un problème peut être vu comme un cas particulier de traitement de données. Par exemple, l'équation du second degré

$$2x^2 + 3x - 2 = 0$$

est entièrement caractérisée par ses trois coefficients $[2, 3, -2]$. On peut donc considérer que l'opération consistant à résoudre cette équation est un traitement des données d'entrée $[2, 3, -2]$, dont le résultat est formé par les données de sortie $\{-2, \frac{1}{2}\}$.

Un ordinateur est donc une machine capable de traiter des données, c'est-à-dire recevoir des données en entrée, leur faire subir un certain traitement, et produire des données de sortie. Comme le montrent les exemples d'ordinateurs discutés précédemment, ces données peuvent prendre différentes formes ; il peut s'agir de nombres, de textes, d'images, de sons, …

Cette définition de l'ordinateur n'est cependant pas complète. Une caractéristique fondamentale d'un ordinateur est que tous les traitements de données qu'il effectue sont basés sur des *opérations préétablies*. Cela signifie que l'ordinateur n'est pas capable de déterminer lui-même quelles opérations de traitement il doit effectuer : l'ensemble de ces opérations et l'ordre dans lequel elles doivent être réalisées doivent lui être explicitement fournis.

Dans le cas d'une équation du second degré comme celle que nous avons déjà évoquée, nous avons appris à l'école secondaire quelles opérations effectuer afin de la résoudre dans le cas général : pour résoudre

$$ax^2 + bx + c = 0,$$

on commence par déterminer si $a \neq 0$. Si c'est le cas, on calcule $\Delta = b^2 - 4ac$. On teste ensuite si Δ est négatif, nul ou positif, et ainsi de suite. La solution générale de ce problème prend la forme d'une procédure qui décrit précisément quelles opérations effectuer et dans quel ordre afin d'obtenir l'ensemble des solutions de l'équation. Une telle procédure ne laisse aucune liberté d'interprétation, et peut être suivie par quiconque comprend les opérations à réaliser et est capable de les effectuer sans erreur.

C'est ainsi que fonctionne un ordinateur : si on lui fournit un *programme* qui décrit rigoureusement un ensemble d'opérations de traitement de données à effectuer dans un certain ordre,

l'ordinateur est capable d'*exécuter* ce programme rapidement et correctement.

Nous avons donc la définition suivante.

Un ordinateur est une machine capable de résoudre des problèmes et de traiter des données en effectuant des opérations préétablies.

Au cinéma, on voit souvent des ordinateurs qui font preuve d'intuition ou d'intelligence. Cela n'existe cependant pas ; dans la réalité, les ordinateurs sont uniquement capables d'exécuter mécaniquement les programmes qu'on leur fournit, sans faire preuve de la moindre initiative ou intelligence.

Cette constatation peut sembler entrer en contradiction avec l'émergence de l'*intelligence artificielle*, qui permet aux systèmes informatiques modernes de battre le meilleur joueur humain aux échecs ou au jeu de go, traduire un texte ou conduire une voiture de façon autonome. Cependant, si des ordinateurs sont capables de réaliser de telles prouesses, c'est uniquement parce qu'on leur a fourni des programmes qui précisent quelles opérations doivent être précisément effectuées pour y arriver. Un ordinateur jouant aux échecs fonctionne de la même façon qu'un autre résolvant une équation du second degré : tous deux se contentent d'exécuter sans réfléchir les instructions de leur programme. À la différence d'un être humain qui réaliserait les mêmes tâches, aucun ne comprend ce qu'est le jeu d'échecs ou une équation du second degré. On peut donc dire que l'"intelligence" dont font preuve de tels systèmes informatiques ne réside pas dans leur ordinateur, mais plutôt dans les programmes qui sont fournis à cet ordinateur.

Le but de ce cours est d'appendre à programmer un ordinateur, c'est-à-dire de rédiger un programme qui permet à un ordinateur d'effectuer un traitement de données particulier ou de résoudre un problème donné. Pour réaliser cela, deux étapes sont nécessaires. Premièrement, étant donné un problème à résoudre, il s'agit de trouver une procédure spécifiant quelles opérations de traitement doivent être effectuées pour arriver à une solution. Une telle procédure s'appelle un *algorithme*, et la discipline qui étudie le développement d'algorithmes est l'*algorithmique*.

Quand un algorithme a été obtenu pour un problème donné, l'étape suivante consiste à le *programmer*, c'est-à-dire l'exprimer sous une forme, le *programme*, qui peut être fournie à un ordinateur en vue d'être exécutée. L'objectif dans ce cours est d'explorer à la fois l'aspect algorithmique et l'aspect programmation de l'informatique, afin d'être capable de passer de l'énoncé d'un problème à une solution effectivement exécutable par un ordinateur. Suffisamment d'éléments seront fournis dans le cours pour que les solutions aux problèmes que nous allons aborder puissent être testées à l'aide d'un ordinateur personnel.

1.1.2 Une petite histoire de l'informatique

L'idée d'essayer d'automatiser le traitement des données, en particulier les opérations de calcul, est très ancienne. Il y a plus de 4000 ans, les Sumériens et puis les Babyloniens représentaient déjà les nombres en *notation positionnelle*, qui est un système d'écriture permettant des procédés de calcul simples pour des opérations arithmétiques telles que l'addition ou la multiplication. Ce système est proche de celui qui est toujours utilisé de nos jours, si ce n'est que la notation babylonienne reposait sur la base 60 plutôt que la base 10 que nous employons maintenant [1].

Bien plus tard, au $9^{\text{ème}}$ siècle, le mathématicien persan Muhammad al-Khwārizmī a publié un traité établissant les bases de l'algèbre. En particulier, cet ouvrage fournit une procédure effective permettant de résoudre n'importe quelle équation du second degré, ce que l'on peut considérer comme étant une des premières publications d'un algorithme. Les termes d'*algorithme* et d'*algorithmique* sont d'ailleurs dérivés du nom latinisé de ce mathématicien.

Au $17^{\text{ème}}$ siècle, le mathématicien français Blaise Pascal a construit une machine capable de manipuler des nombres, dans le but de faciliter le travail de son père qui était comptable. Cette machine permettait de calculer automatiquement des additions et des soustractions, en implémentant les règles du calcul écrit grâce à un procédé mécanique reposant sur un système d'engrenages. On peut y voir un des premiers dispositifs spécifiquement construits pour traiter automatiquement des données.

Au $19^{\text{ème}}$ siècle, un ingénieur anglais, Charles Babbage, a conçu, construit et fait fonctionner une machine arithmétique mécanique, la *machine à différences (difference engine)*, basée sur les mêmes principes que celle de Pascal, mais capable d'effectuer des opérations plus complexes. L'objectif consistait à générer automatiquement des tables de valeurs pour une fonction polynomiale donnée. Étant donné que n'importe quelle fonction réelle peut être approximée avec un degré de précision arbitraire par un polynôme, une telle machine permet de produire automatiquement et sans erreur des tables de valeurs pour les fonctions élémentaires telles que *exp*, *log*, *sin*, *cos*, *tan*, ... utilisées en mathématique et en ingénierie.

Tout comme la machine de Pascal, on ne peut pas considérer que la machine à différences de Babbage constitue un ordinateur au sens général du terme, car il n'est pas possible de programmer cette machine afin de lui faire effectuer n'importe quel traitement de données : la seule opération qu'elle est capable de réaliser est celle qui consiste à générer des tables de valeurs pour des polynômes. Par exemple, on ne peut pas l'utiliser pour résoudre une équation du second degré ou jouer aux échecs. Babbage a cependant réalisé qu'il serait possible de concevoir une machine qui pourrait être programmée de façon à effectuer n'importe quelle opération de traitement. Il a alors consacré le reste de sa vie à mettre au point une telle machine, appelée la *machine analytique (analytical engine)*. Cette machine, beaucoup plus complexe que la machine

1. L'influence de la notation babylonienne est toujours présente de nos jours dans la représentation des angles et des heures, qui utilisent la base 60.

à différences, n'a jamais pu être entièrement construite. Conceptuellement, il s'agit cependant du premier modèle d'ordinateur ayant été imaginé.

Les programmes destinés à la machine analytique devaient être représentés sous la forme de *cartes perforées*, c'est-à-dire de cartes en carton percées de trous dont la disposition encode les instructions à effectuer. Ce mécanisme, développé à l'origine pour spécifier les motifs devant être générés par des métiers à tisser, a été largement utilisé en informatique jusqu'aux années 1980.

En parallèle à ces travaux visant à construire une machine capable de traiter les données de la façon la plus générale possible, et avant même qu'une telle machine ne soit disponible, des mathématiciens se sont intéressés aux propriétés théoriques des ordinateurs. En particulier, un chercheur anglais, Alan Turing, a établi en 1936 en collaboration avec son promoteur de doctorat Alonzo Church qu'un très petit ensemble d'opérations élémentaires suffit pour implémenter n'importe quel algorithme. Alan Turing a également mis en évidence certaines limitations de l'informatique, en démontrant qu'il existe des problèmes qui ne peuvent pas être résolus par un ordinateur, quels que soient les mécanismes de celui-ci. En d'autres termes, ce résultat prouve l'existence de problèmes qui n'admettent pas de solution algorithmique [2].

Il faut attendre 1941 pour voir arriver la première machine entièrement fonctionnelle possédant toutes les caractéristiques d'un ordinateur : le *Z3*, construit par Konrad Zuse, un ingénieur allemand. À la différence de la machine analytique dont les mécanismes étaient basés sur des jeux d'engrenages, le Z3 était un dispositif électromécanique essentiellement basé sur quelques milliers de relais. L'avantage de cette approche est qu'une telle machine est plus facile à construire et à mettre au point qu'un appareil purement mécanique employant une combinaison complexe d'engrenages. Bien sûr, il n'est pas immédiat de comprendre comment il est possible d'assembler des composants tels que des relais pour obtenir une machine capable d'exécuter des programmes. Cette question est explorée plus en détail dans les cours d'*Organisation des ordinateurs* et de *Computation structures*.

La puissance de calcul du Z3 était très modeste : une opération d'addition de deux nombres nécessitait de l'ordre d'une seconde, et une multiplication trois fois plus de temps. Les programmes étaient fournis sous la forme de films perforés. Ce mécanisme a été amélioré une dizaine d'années plus tard, quand sont apparus des ordinateurs qui considéraient leurs programmes comme étant des données particulières. Ces ordinateurs étaient donc capable d'effectuer des opérations de traitement dont le résultat prenait la forme de programmes, pouvant ensuite être exécutés. Un exemple de tel ordinateur est l'*EDVAC (Electronic Discrete Variable Automatic Computer)* mis au point à l'Université de Pennsylvanie (USA) en 1951. Cet ordinateur était construit à l'aide d'environ 18000 tubes à vide, qui sont des composants purement électroniques pouvant remplacer les relais, et atteignait une puissance de calcul de l'ordre de 1200 opérations arithmétiques par seconde. Il pesait 7,8 tonnes, consommait 56 kW de puissance électrique, et

2. Un exemple de tel problème est le *problème de l'arrêt*, qui vise à déterminer si l'exécution d'un programme informatique donné en entrée finit par s'arrêter, ou bien continue ses opérations indéfiniment.

coûtait un demi-million de dollars de l'époque.

Une avancée technologique majeure qui a révolutionné l'informatique est l'invention du *transistor* en 1947. Il s'agit d'un composant électronique pouvant être utilisé comme les relais et les tubes à vide pour construire des circuits capables de traiter les données. L'avantage des transistors est cependant qu'ils sont faciles à miniaturiser et beaucoup plus économes en énergie. Il devient alors possible d'équiper les ordinateurs de circuits considérablement plus complexes, ainsi que plus compacts. Le premier ordinateur construit à l'aide de transistors voit le jour en 1953. En 1958, l'invention du *circuit intégré* permet ensuite de miniaturiser davantage les circuits des ordinateurs. Le premier *microprocesseur*, qui intègre au sein d'un seul composant tous les circuits d'un ordinateur chargés d'exécuter les programmes est produit en 1971. Il comprend environ 2300 transistors et est capable d'effectuer environ 92000 opérations par seconde, pour un prix de l'ordre de 200 dollars de l'époque.

À côté de cette évolution technologique dans le domaine de l'électronique, des progrès sont également réalisés quant à la façon d'utiliser les ordinateurs. Des *langages de programmation* tels que *Fortran* (1954), *LISP* (1958) et *COBOL* (1959) sont créés afin de faciliter le développement de logiciels de plus en plus complexes. Ces langages sont toujours en partie utilisés de nos jours. Le système d'exploitation *Unix* commence à être développé en 1969, et est toujours largement utilisé [3].

La première version du langage de programmation *C*, qui est celui étudié dans ce cours, apparaît en 1972. Il s'agit toujours actuellement du langage de programmation le plus utilisé. À la même époque voient également le jour les premiers ordinateurs destinés au grand public, sous la forme de micro-ordinateurs (1978) et d'ordinateurs personnels (*Personal Computer, PC*, 1981). Le système d'exploitation *Windows* fait son apparition en 1985, le *World-Wide Web* en 1989, et le langage de programmation *Java* en 1995.

Ce processus d'évolution technologique se poursuit. De nos jours, les microprocesseurs qui équipent les ordinateurs modernes comprennent des milliards de transistors, dont les dimensions ont été réduites jusqu'à être de l'ordre de celles de quelques dizaines d'atomes de silicium. Ces microprocesseurs sont capables d'effectuer plusieurs centaines de milliards d'opérations par seconde, pour un prix de quelques centaines de dollars. De nouveaux langages et environnements de programmation sont également régulièrement créés afin de simplifier le processus de développement de logiciels.

3. Ce système est notamment à la base de *Linux*, sur lequel repose en particulier *Android*, et de *macOS*.

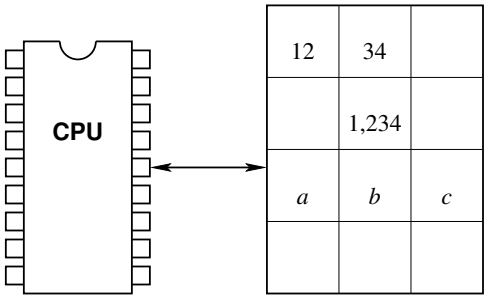

FIGURE 1.1 – Modèle conceptuel d'un ordinateur

1.2 L'algorithmique

L'algorithmique s'intéresse à la façon de produire, à partir de l'énoncé d'un problème, un algorithme capable de résoudre ce dernier, en d'autres termes une procédure pouvant être exécutée par un ordinateur.

1.2.1 La notion d'algorithme

Pour pouvoir raisonner sur des algorithmes, il est nécessaire de faire des hypothèses sur le type d'ordinateur dont on dispose ; en particulier, il faut spécifier la nature des opérations que cet ordinateur est capable d'effectuer. Notre but est de rester le plus indépendant possible des détails matériels de l'ordinateur utilisé, afin de pouvoir développer des algorithmes pouvant potentiellement être exécutés par n'importe quel ordinateur.

À un niveau d'abstraction élevé, on peut considérer qu'un ordinateur possède deux composants principaux, illustrés à la figure 1.1. Le premier composant est le *processeur (Central Processing Unit, CPU)*, qui est chargé d'exécuter les programmes et d'effectuer les opérations de traitement des données. Le deuxième composant est une *mémoire* qui retient toutes les données manipulées par le processeur. Ces données incluent les données d'entrée qui sont fournies aux programmes, les données de sortie produites par ceux-ci, et les données de travail dont les programmes ont besoin pour gérer les résultats intermédiaires des opérations qu'ils effectuent. Les programmes eux-mêmes peuvent être vus comme une donnée particulière qui est également placée en mémoire.

Les données présentes en mémoire peuvent être de différents types : il peut notamment s'agir de nombres, de textes, ou de données plus complexes. Conceptuellement, on peut considérer que la mémoire est structurée en un ensemble de cases contenant chacune une donnée élémentaire. Au cours de ses opérations, le processeur peut lire le contenu d'une case donnée de la mémoire,

ou modifier le contenu d'une case en y écrivant une nouvelle valeur qui remplace alors la précédente.

Par rapport à ce modèle conceptuel d'ordinateur, on peut maintenant définir ce que l'on entend par algorithme : un *algorithme* est une procédure permettant de résoudre un problème ou de traiter des données en décrivant précisément quelles opérations doivent être effectuées. Une opération effectuée par un algorithme peut correspondre à une lecture ou une écriture en mémoire, ou bien à une opération de traitement interne au processeur.

En reprenant l'exemple de la résolution d'une équation du second degré $ax^2 + bx + c = 0$, on pourrait convenir de placer dans les trois premières cases de la mémoire les données d'entrée a, b et c de ce problème. Un algorithme chargé de résoudre l'équation lirait alors le contenu de ces cases, et effectuerait les opérations de traitement correspondantes, par exemple, une de ses premières opérations consisterait à déterminer si le contenu a de la première case est nul ou non. Au cours de l'exécution de l'algorithme, d'autres cases de la mémoire seraient utilisées pour retenir des résultats intermédiaires, par exemple la valeur du discriminant $\Delta = b^2 - 4ac$. À la fin de l'exécution de l'algorithme, celui-ci laisserait dans des cases prédéterminées de la mémoire le nombre de solutions obtenues et leur valeur.

De nombreux problèmes se prêtent à une résolution algorithmique. Informellement, on peut citer ceux qui consistent à calculer la racine carrée d'un nombre positif à une certaine précision près, de déterminer si un nombre naturel est premier ou non, de calculer le plus court chemin entre deux points dans un espace donné, de calculer le meilleur coup à jouer dans une configuration donnée du jeu d'échecs ou de go, de piloter un avion ou une voiture en fonction de consignes particulières et de données transmises par différents capteurs, de détecter des visages dans une photographie, de traduire un texte ou une conversation du français vers l'anglais, de simuler l'évolution d'un phénomène physique à partir d'une situation initiale connue, ... Dans la section suivante, nous allons examiner plus en détail certains problèmes, en essayant de développer des algorithmes permettant de les résoudre.

1.2.2 Exemples de problèmes algorithmiques

La fonction factorielle

L'énoncé du problème est le suivant : étant donné un entier $n \geq 1$, calculer

$$n! = n\,(n-1)\,(n-2)\,\ldots\,1.$$

Il s'agit d'un problème qui ne présente aucune difficulté car son énoncé mène directement à un algorithme permettant de le résoudre : pour obtenir $n!$ pour une valeur donnée de n, il suffit de calculer successivement n, $n-1$, $n-2$, ... et de multiplier ces nombres. (Une astuce utile est de réaliser que l'on peut terminer l'énumération des facteurs à 2 plutôt qu'à 1, pour $n > 1$.)

Le plus grand commun diviseur

Tous les problèmes n'admettent pas une solution aussi immédiate que le précédent. Nous nous intéressons à présent au calcul du *Plus Grand Commun Diviseur (PGCD)* de deux entiers $a, b \geq 1$ donnés, c'est-à-dire le plus grand nombre $\gcd(a, b)$ qui divise à la fois a et b.

On peut comme dans le cas précédent chercher à obtenir une solution algorithmique directement dérivée de l'énoncé du problème : puisque $\gcd(a, b)$ divise a et b, on a $\gcd(a, b) \leq a$ et $\gcd(a, b) \leq b$. Pour calculer $\gcd(a, b)$, il suffit donc d'énumérer tous les entiers appartenant à l'intervalle $[1, \min(a, b)]$, de déterminer pour chacun d'entre eux s'il divise à la fois a et b, et de garder le plus grand entier qui satisfait cette condition. Une façon simple d'implémenter ce dernier mécanisme consiste à énumérer les éléments de l'intervalle par ordre décroissant de valeur, et à s'arrêter dès que la valeur courante divise à la fois a et b.

Cette solution algorithmique naïve est correcte, mais inefficace : l'utiliser pour calculer, par exemple, $\gcd(5229827045084057754, 1570889565818035132)$ conduit à devoir effectuer un nombre d'opérations qui est hors de portée de la capacité des ordinateurs actuels. Nous allons cependant montrer qu'il est possible de développer un meilleur algorithme, capable de produire pratiquement instantanément le résultat de ce calcul [4].

Cette situation est représentative de la plupart des problèmes que l'on rencontre en algorithmique : les solutions que l'on peut directement dériver de leur énoncé sont trop inefficaces pour être utilisables en pratique. Il devient alors nécessaire d'étudier plus en profondeur ces problèmes pour en comprendre la structure et découvrir des propriétés que l'on peut exploiter afin de réduire le nombre d'opérations à effectuer. Il n'existe cependant aucune méthode systématique pour faire cela ; l'algorithmique s'apprend en se construisant petit à petit une expérience de résolution de problèmes variés.

Pour le calcul du PGCD, un algorithme efficace est connu depuis longtemps. Au $3^{\text{ème}}$ siècle avant notre ère, le mathématicien grec Euclide a proposé une solution basée sur les principes suivants.

Premièrement, la définition de la fonction $\gcd(a, b)$ est symétrique en a et b : Pour tous $a, b \geq 1$, on a $\gcd(a, b) = \gcd(b, a)$. Cela permet de simplifier le problème en supposant sans perte de généralité que l'on a $a \leq b$. (Si cette condition n'est pas satisfaite pour des valeurs données de a et de b, il suffit de les permuter.)

Ensuite, on remarque que si un nombre divise à la fois a et b, alors il divise également $b - a$. De façon similaire, si un nombre divise à la fois a et $b-a$, alors il divise également b. On en déduit que l'on a $\gcd(a, b) = \gcd(a, b-a)$. Cette propriété est très intéressante, car elle permet de réduire la taille des instances du problème : remplacer le calcul de $\gcd(a, b)$ par celui de $\gcd(a, b - a)$

4. Ce résultat est égal à 67956946.

conduit à manipuler des nombres plus petits, ce qui est de nature à rendre le problème plus facile.

On peut continuer à appliquer cette propriété de réduction : on a

$$\begin{aligned}\gcd(a,b) &= \gcd(a, b-a)\\ &= \gcd(a, b-2a)\\ &= \gcd(a, b-3a)\\ &\;\;\vdots\\ &= \gcd(a, b-ka),\end{aligned}$$

où k est un entier tel que $ka \leq b$. En choisissant la plus grande valeur possible de k qui satisfait cette condition, on obtient finalement

$$\gcd(a,b) = \gcd(a, b \bmod a),$$

où $x \bmod y$ désigne le reste de la division de x par y. Notons que pour tout $a > 0$, on a $\gcd(a,0) = a$, car 0 est divisible par n'importe quelle valeur non nulle.

Les propriétés que nous avons établies conduisent à un algorithme efficace (l'*algorithme d'Euclide*) pour calculer $\gcd(a,b)$ pour n'importe quels entiers $a, b \geq 1$:

1. Si $a > b$, alors permuter a et b.
2. Tant que $a \neq 0$, répéter l'opération suivante :
   — Remplacer (a,b) par $(b \bmod a, a)$.
3. Retourner la valeur de b.

Le tableau suivant illustre les étapes successives de l'exécution de cet algorithme pour le calcul de $\gcd(24,18)$:

Étape	a	b
1	24	18
2	18	24
3	6	18
4	0	6

L'algorithme d'Euclide est très efficace. Il a été démontré [5] que le nombre d'étapes nécessaires au calcul de $\gcd(a,b)$, pour $a \leq b$, est inférieur ou égal à cinq fois le nombre de chiffres de l'écriture décimale de a.

5. Gabriel Lamé, 1844. *Note sur la limite du nombre des divisions dans la recherche du plus grand commun diviseur entre deux nombres entiers*. Comptes rendus de l'Académie des Sciences, 19 :867–870.

La racine carrée

Nous allons à présent aborder le problème de calculer la racine carrée d'un nombre réel positif ou nul donné. Avant de chercher à le résoudre, il convient d'abord d'en préciser exactement les détails. La difficulté est qu'un ordinateur n'est en toute généralité pas capable de manipuler des nombres réels arbitraires, car pour connaître la valeur d'un tel nombre, il est nécessaire de disposer d'une quantité infinie de données. En effet, si l'on représente les réels en notation positionnelle, chacun d'entre eux est caractérisé par une séquence infinie de chiffres qui sont tous significatifs.

En pratique, la plupart des applications ne nécessitent pas de connaître exactement la valeur des nombres que l'on est amené à manipuler. On opte alors pour une représentation approximative des réels, en fixant par exemple une tolérance affectant les données d'entrée et le résultat des opérations effectuées. On peut ainsi spécifier le problème de calculer la racine carrée d'un nombre réel de la façon suivante : étant donnés un réel $r \geq 0$ et une précision souhaitée $\varepsilon > 0$, le problème consiste à calculer une valeur $\mathrm{sqrt}(r)$ telle que

$$|\mathrm{sqrt}(r) - \sqrt{r}| < \varepsilon.$$

Contrairement aux deux précédents, l'énoncé de ce problème ne donne pas d'indications immédiates sur la façon de le résoudre algorithmiquement. L'étude des principes qui sous-tendent le développement de tels algorithmes sort du cadre de ce cours [6]. Nous nous contentons de donner ici un exemple de solution connue depuis plusieurs milliers d'années (la *méthode de Héron*) :

1. Partir d'une estimation initiale quelconque x de $\sqrt{r}$, par exemple $x = r$.
2. Remplacer x par $x' = \dfrac{x + \frac{r}{x}}{2}$.
3. Tant que les deux dernières estimations x et x' sont telles que $|x - x'| \geq \varepsilon$, répéter l'étape précédente.
4. Sinon, retourner la valeur de la dernière estimation x'.

Par exemple, pour les données d'entrée $r = 2$ et $\varepsilon = 10^{-6}$, les estimations de $\sqrt{2}$ successivement produites par cet algorithme sont les suivantes.

6. Ils font notamment partie de la matière du cours d'*Introduction à l'algorithmique numérique*.

Étape	Estimation x
1	2.00000000
2	1.50000000
3	1.41666667
4	1.41421569
5	1.41421356
6	1.41421356

L'algorithme de Héron est en fait particulièrement efficace ; on peut ainsi démontrer que chacune de ses étapes permet d'environ doubler le nombre de chiffres corrects du résultat. Signalons cependant que l'implémentation pratique de cet algorithme n'est pas immédiate : comme nous l'avons vu, les représentations des nombres réels utilisées en informatique sont approximatives, et il est nécessaire de garantir que les opérations arithmétiques réalisées aux différentes étapes de l'algorithme produisent un résultat suffisamment précis. Il s'agit également d'une problématique qui est abordée dans le cours d'*Introduction à l'algorithmique numérique*.

1.3 La programmation

Dans la section 1.2.2, nous avons présenté trois exemples de problèmes qui peuvent être résolus à l'aide d'algorithmes. Nous nous intéressons maintenant à l'opération qui consiste à traduire ces algorithmes en des *programmes* pouvant être exécutés par un ordinateur.

1.3.1 L'implémentation d'un algorithme

Nous avons vu que le *processeur* est le composant de l'ordinateur chargé d'exécuter les programmes. Techniquement, un processeur est capable d'exécuter des *instructions*, qui sont des opérations élémentaires que ses circuits sont directement à même d'effectuer. Le *jeu d'instructions* d'un processeur, c'est-à-dire l'ensemble des instructions qu'il est capable d'exécuter, dépend de son *architecture*. Il existe des architectures de processeurs très simples, que l'on rencontre notamment dans des applications embarquées de petite envergure, et d'autres beaucoup plus complexes, comme celle des processeurs qui équipent les ordinateurs personnels modernes [7].

Pour traduire un programme sous une forme qui lui permet d'être exécuté par un processeur, il faut donc exprimer ce programme comme une combinaison d'instructions appartenant au jeu d'instructions de ce processeur. Dans la section 1.1.2, nous avons mentionné qu'Alan Turing a établi que le jeu d'instructions d'un processeur peut être très simple : À l'aide d'un très petit

7. Cette architecture est étudiée dans le cours d'*Organisation des ordinateurs*

nombre d'instructions [8], il est possible d'implémenter n'importe quel algorithme. Bien sûr, il s'agit de considérations théoriques ; en pratique, on emploie des architectures complexes dans un but d'efficacité, l'objectif étant que les programmes s'exécutent le plus rapidement possible.

1.3.2 Les langages de programmation

Il n'est pas très commode pour les programmeurs de devoir tenir compte de l'architecture des processeurs qui exécutent leurs programmes. Afin de pouvoir s'abstraire de ce genre de détail, et aussi de disposer de mécanismes de programmation plus confortables que les instructions directement exécutables par les processeurs, on a défini des *langages de programmation*. Ceux-ci permettent de programmer des algorithmes en les exprimant sous une forme qui reste relativement indépendante des détails matériels de l'ordinateur utilisé. Grâce à ces langages, il devient possible d'écrire des programmes pouvant être déployés sur n'importe quelle architecture de processeur.

Il existe plusieurs familles de langages de programmation. Dans ce cours, nous étudions les mécanismes d'un langage *impératif*, c'est-à-dire qui conduit à préciser explicitement les opérations devant être effectuées pendant l'exécution d'un programme. Il existe aussi des langages dits *déclaratifs*, pour lesquels ces opérations ne figurent pas directement dans les programmes ; de tels langages sont notamment utilisés pour interroger des bases de données (e.g., SQL). Certains langages sont de *haut niveau*, c'est-à-dire qu'ils mettent à la disposition du programmeur des mécanismes d'abstraction évolués ; d'autres sont plus proches du jeu d'instructions du processeur. Il existe des langages génériques, comme C, C++, Java, Python, C#, Rust et Go, et d'autres plus spécifiques à des domaines d'application particuliers.

Dans ce cours, nous allons étudier le langage de programmation C, qui a l'avantage d'être simple et très utilisé. Les concepts et mécanismes de programmation que nous allons introduire ont cependant une portée plus générale, et sont facilement transposables à d'autres langages de programmation impératifs.

Les programmes que l'on rédige dans un langage de programmation donné ne peuvent en général pas être directement exécutés par un processeur : le *code source* d'un programme doit d'abord être traduit en *code machine*, composé d'instructions appartenant au jeu d'instructions du processeur. Cette opération de traduction du code source vers le code machine peut être réalisée soit avant l'exécution du programme (on parle alors de *compilation*), soit au fur et à mesure de cette exécution (il s'agit alors d'*interprétation*). Le mécanisme de compilation est en général plus efficace que l'interprétation, car il permet de rendre toute la puissance de calcul du processeur disponible pour l'exécution des opérations utiles d'un programme. Il s'agit de l'approche mise

8. Par exemple, des instructions permettant d'incrémenter et de décrémenter des variables entières, et de tester si leur valeur est égale à zéro, suffisent.

en œuvre par le langage C : avant de pouvoir exécuter un programme C, il faut employer un outil logiciel appelé *compilateur*, pour traduire le code source de ce programme en du code machine exécutable par le processeur. Nous montrerons à la section 1.3.4 comment effectuer concrètement cette opération.

Tous les langages de programmation sont définis grâce à une *syntaxe* et une *sémantique*. La syntaxe précise un certain nombre de règles d'écriture que tous les programmes doivent respecter pour être *syntaxiquement valides*. Par exemple, la syntaxe du langage C impose de faire suivre la plupart des instructions d'un programme par un point-virgule "`;`". Si un programme n'est pas syntaxiquement correct, alors sa compilation échoue en produisant un message d'erreur indiquant la nature du problème.

À côté de la syntaxe, la sémantique d'un langage donne un sens aux programmes qui sont syntaxiquement valides. On peut distinguer la *sémantique statique*, qui impose un certain nombre de règles devant être satisfaites par les programme pour qu'ils puissent être considérés sémantiquement valides, et la *sémantique dynamique*, qui spécifie la nature des opérations qui seront effectuées lors de l'exécution du programme. Par exemple, la sémantique statique du langage C requiert que chaque variable soit déclarée [9] avant d'être utilisée. Les programmes qui ne respectent pas cette contrainte sont incorrects, même s'ils sont syntaxiquement valides, et leur compilation échoue en affichant un message d'erreur.

1.3.3 Un premier programme

La figure 1.2 fournit le code source d'un programme C complet implémentant l'algorithme d'Euclide étudié dans la section 1.2.2. Nous allons d'abord présenter de façon intuitive les éléments de ce programme ; l'ensemble des mécanismes et constructions utilisés seront étudiés plus en détail dans le chapitre 2. Nous verrons à la section 1.3.4 comment compiler ce programme afin de le faire exécuter concrètement par un ordinateur.

Ce programme est essentiellement composé d'*instructions*, comme `int a, b, c`, `scanf("%d %d", &a, &b)` ou `a = c`. Comme cela a déjà été mentionné, la syntaxe du langage C impose de faire suivre d'un point-virgule "`;`" la plupart des instructions d'un programme.

Les instructions d'un programme peuvent être combinées pour former des *séquences*. Par exemple, dans notre programme, les instructions `c = a`, `a = b` et `b = c` sont combinées de façon à former la séquence

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

9. Les mécanismes correspondants seront introduits au chapitre 2.

```
#include <stdio.h>

int main()
{
  int a, b, c;

  printf("Entrez deux entiers strictement positifs: ");
  scanf("%d %d", &a, &b);

  if (a > b)
    {
      c = a;
      a = b;
      b = c;
    }

  while (a != 0)
    {
      c = a;
      a = b % a;
      b = c;
    }

  printf("Le PGCD est égal à %d.\n", b);
}
```

FIGURE 1.2 – Implémentation de l'algorithme d'Euclide

Une séquence peut comprendre n'importe quel nombre d'instructions. Ces instructions sont exécutées une à une, dans l'ordre où elles apparaissent. En entourant une séquence d'instructions d'*accolades* "{" et "}", on obtient un *bloc* :

```
{
  c = a;
  a = b;
  b = c;
}
```

Un bloc joue le même rôle syntaxique qu'une instruction individuelle. Cela signifie qu'il est généralement permis, à quelques rares exceptions près, de remplacer une instruction d'un programme par un bloc. En particulier, une instruction figurant dans un bloc peut aussi être remplacée par un bloc, et ainsi de suite jusqu'à un niveau d'imbrication arbitraire. Les langages de programmation qui possèdent un tel mécanisme sont appelés les *langages structurés*.

Les espaces blancs comme le symbole d'espacement " ", les tabulations et les sauts de ligne n'ont pas de fonction syntaxique en C et sont en général ignorés par le compilateur. Leur usage permet cependant de rendre le code source d'un programme plus lisible ; en particulier, on *indente* les instructions d'un programme de façon à en expliciter la structure, en alignant les instructions de chaque bloc à une position horizontale décalée par rapport à celles du bloc qui les contient [10].

Dans le programme, on voit qu'à la suite de la ligne contenant `int main()` se trouve un bloc qui englobe le reste des instructions du programme. Remarquons que la convention d'indentation permet de déterminer facilement où ce bloc commence et où il se termine. Nous n'allons pas à ce stade expliquer précisément ce que signifie cette ligne de code [11], mais simplement mentionner que le bloc que nous venons de décrire est celui qui sera exécuté dès que le programme commencera son exécution.

La première instruction `int a, b, c` de ce bloc principal est une *déclaration de variables*. Une *variable* est un emplacement de mémoire reservé pendant une partie de l'exécution d'un programme, servant à retenir une donnée d'un certain *type*. Cette instruction déclare trois variables capables de retenir des valeurs entières. Les noms de ces variables sont spécifiés par le biais des *identificateurs* `a`, `b` et `c`. Ceux-ci apparaissent dans les instructions suivantes du programme qui sont amenées à manipuler ces variables.

Par exemple, l'instruction `c = a` est une instruction d'*affectation* dont l'effet est de consulter la valeur courante de la variable `a` et d'*affecter* cette valeur à la variable `c`, ce qui revient à remplacer le contenu de `c` par celui de `a` (le contenu précédent de `c` est alors perdu).

10. Dans certains langages de programmation comme *Python*, l'indentation des instructions présente une valeur syntaxique.

11. Ces explications viendront au chapitre 4.

Cette instruction fait partie de la séquence d'instructions

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

qui a pour effet de permuter les valeurs des variables `a` et `b`, en utilisant `c` comme un espace de travail auxiliaire.

L'instruction `printf(" ... ")` qui suit la déclaration des variables est une *invocation de fonction* : "`printf`" n'est pas un mot-clé du langage C, mais l'identificateur d'une *fonction*. Cette fonction correspond à un fragment de code défini ailleurs, dont l'exécution peut être déclenchée à cet endroit du programme. (Il s'agit ici d'une fonction de la *bibliothèque standard (standard library)*, présente dans toutes les installations de l'environnement de programmation C, et qui contient l'implémentation d'opérations d'usage courant.) Dans le cas présent, la fonction `printf` permet d'afficher un message à l'écran ; ce message, qui lui est fourni en *argument*, demande à l'utilisateur d'entrer deux entiers positifs.

De la même façon, l'instruction `scanf("%d %d", &a, &b)` invoque la fonction `scanf` de la bibliothèque standard. À ce stade du cours, il n'est pas encore possible d'expliquer le détail des mécanismes qui interviennent dans cette instruction d'invocation. Intuitivement, cette instruction attend que l'utilisateur introduise au clavier deux nombres entiers, et écrit la valeur de ces nombres dans les variables `a` et `b`.

Une fois la valeur de `a` et de `b` connue, la première étape de l'algorithme d'Euclide consiste à les permuter dans le cas particulier où l'on a `a` > `b`. C'est l'objet de l'instruction suivante du programme. Une instruction de la forme

`if` (*expression E*)
instruction I

évalue d'abord l'expression *E*. Une *expression* fournit le moyen de calculer une valeur, en partant de données élémentaires comme la valeur de variables ou de constantes, et en leur appliquant des *opérateurs*. Par exemple, l'expression `a < b`, obtenue en appliquant l'opérateur "<" aux sous-expressions élémentaires `a` et `b`, compare la valeur des variables `a` et `b`, et fournit un résultat considéré comme vrai si `a` possède une valeur strictement inférieure à celle de `b`. Ensuite, si l'évaluation de *E* fournit un résultat vrai, et seulement dans ce cas, alors l'instruction *I* est exécutée. Dans cette instruction, "`if`" est un *mot-clé* du langage, c'est-à-dire un mot réservé qui ne peut pas être utilisé comme identificateur. Remarquons que dans notre programme, l'instruction *I* est remplacée par un bloc, comme le permet le mécanisme de programmation structurée. Globalement, l'effet de l'instruction

```
if (a > b)
  {
    c = a;
    a = b;
    b = c;
  }
```

est donc bien de permuter les valeurs de `a` et de `b`, en s'aidant de `c`, lorsque la condition `a > b` est satisfaite.

L'instruction suivante est une instruction de *boucle*. Une instruction de la forme

`while (`*expression E*`)`
instruction I

basée sur le mot-clé `while` évalue d'abord l'expression *E*. Si le résultat obtenu est vrai, alors l'instruction *I* est exécutée, et le processus recommence : on évalue à nouveau *E*, on exécute *I* si le résultat est vrai, et ainsi de suite. L'opération se termine dès que l'évaluation de *E* cesse de produire un résultat vrai. Si cela se produit à la première évaluation de *E*, alors *I* n'est pas exécutée du tout.

L'instruction `while` figurant dans le programme détermine si la valeur de `a` est non nulle (l'opérateur `!=` teste si ses deux opérandes possèdent une valeur différente) et, tant que cette condition est satisfaite, exécute la séquence

```
c = a;
a = b % a;
b = c;
```

La seule instruction inédite de cette séquence est `a = b % a`. Cette instruction évalue l'expression `b % a`, et écrit le résultat de cette évaluation dans `a`. L'opérateur "`%`" est l'opérateur de *modulo*, qui calcule le reste de la division de sa première opérande par la seconde. Notre séquence d'instructions a donc pour effet de remplacer la valeur de la paire (a, b) par $(b \bmod a, a)$, qui est bien l'opération devant être réalisée par une itération de l'algorithme d'Euclide.

Ensuite, la dernière instruction du programme invoque la fonction `printf` afin d'afficher à l'écran le résultat du calcul, contenu dans `b`. Une fois encore, les détails de cette instruction seront précisément expliqués dans la suite du cours.

Il reste pour terminer une ligne du programme que nous n'avons pas décrite : celle qui contient `#include <stdio.h>`. Il ne s'agit pas à proprement parler d'une instruction du langage C, mais d'une *directive de prétraitement* (ou *directive de compilation*). Son but consiste à

indiquer au compilateur de commencer par traiter le contenu d'un fichier appelé `stdio.h` avant de compiler la suite du programme. Ce fichier fait partie de la bibliothèque standard et contient, entre autres, la déclaration des fonctions `printf` et `scanf`, que le compilateur doit connaître pour pouvoir gérer correctement les instructions d'invocation de ces fonctions. Nous reviendrons sur ces mécanismes dans le chapitre 4.

1.3.4 La compilation et l'exécution d'un programme

Montrons maintenant comment compiler et exécuter le programme de la figure 1.2 sur un ordinateur personnel. Il existe différents types de compilateurs et d'environnements de développement pour le langage C. Nous illustrerons les exemples de ce cours à l'aide de *GCC* (*the Gnu Compiler Collection*), qui est disponible pour une très grande variété d'architectures de processeurs et de systèmes d'exploitation, et permet également de compiler d'autres langages que C.

La façon la plus simple d'invoquer le compilateur GCC est de le faire en *ligne de commande* : en supposant que le code source du programme se trouve dans un fichier appelé `pgcd.c` la commande

```
gcc -o pgcd pgcd.c
```

invoque le compilateur en lui passant en arguments le nom "`pgcd.c`" de ce fichier, précédé par l'option "`-o pgcd`". Cette option spécifie que le résultat de la compilation doit être écrit dans un fichier appelé `pgcd`. En d'autres termes, cette commande demande au compilateur de traduire le fichier source `pgcd.c` en un fichier exécutable `pgcd`. Cette situation est illustrée à la figure 1.3.

Si cette compilation réussit, ce qui doit être le cas si le contenu de `pgcd.c` correspond exactement au code fourni à la figure 1.2, alors elle génère un fichier `pgcd` qui peut être exécuté. Dans un environnement de programmation *Unix* (par exemple, *Linux*), cette exécution peut être lancée à l'aide de la commande

```
./pgcd
```

qui demande le chargement et l'exécution du programme contenu dans le fichier `pgcd` situé dans le répertoire courant ("`.`").

Un exemple complet de compilation et d'exécution de ce programme est donné ci-dessous (le préfixe "`%`" distingue les commandes entrées par l'utilisateur) :

```
% gcc -o pgcd pgcd.c
% ./pgcd
Entrez deux entiers strictement positifs: 19675050 8178000
Le PGCD est égal à 4350.
```

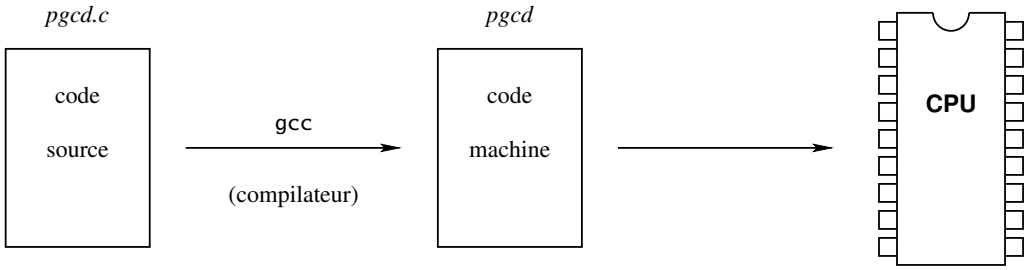

FIGURE 1.3 – Compilation d'un programme

1.4 Quelques considérations théoriques

1.4.1 Les ressources consommées

L'exécution de l'algorithme d'Euclide de la figure 1.2 sur un ordinateur personnel donne l'impression de s'effectuer instantanément [12]. Ce n'est pas toujours le cas ; par exemple, si nous avions programmé le calcul du PGCD à l'aide de la première solution obtenue dans la section 1.2.2, le temps nécessaire à l'exécution du programme aurait été considérablement plus élevé pour de grandes valeurs d'entrée.

On voit donc que différents algorithmes résolvant le même problème peuvent montrer des performances différentes. On abordera la question de mesurer les performances d'un algorithme au chapitre 3. De façon générale, il y a deux types de ressources quantitatives consommées au cours de l'exécution d'un programme : le *temps d'exécution* du programme, et la taille de l'*espace mémoire* nécessaire à cette exécution (c'est-à-dire la quantité des données à retenir). Nous introduirons une notion de *complexité* visant à décrire la quantité de ces ressources nécessaires à un algorithme d'une façon la plus indépendante possible des détails de l'environnement d'exécution.

Un algorithme possède donc une *complexité en temps* et une *complexité en espace*. Celles-ci sont déterminées dans le cas le plus défavorable possible, en d'autres termes, pour les instances du problème qui conduisent au temps d'exécution (pour la complexité en temps) ou à la quantité de mémoire consommée (pour la complexité en espace) les plus élevés possibles. L'idée est que si l'on est à même d'établir qu'un algorithme est efficace dans le pire des cas, alors on obtient la garantie qu'il l'est toujours. Notons qu'il existe également des notions de complexité moyenne, calculées par rapport à une distribution de probabilité associée aux instances possibles du problème étudié. Ces notions de complexité moyenne sont beaucoup plus difficiles à mettre

12. La durée réelle de cette exécution se compte en microsecondes.

en œuvre.

1.4.2 Les limitations de l'informatique

Comme cela a déjà été évoqué à la section 1.1.2, l'informatique possède des limitations fondamentales. En particulier, on peut démontrer l'existence de problèmes pour lesquels il n'existe aucune solution algorithmique efficace, voire même aucune solution du tout.

Un exemple de problème figurant dans la première catégorie est celui qui consiste à décider une formule de l'*arithmétique de Presburger*. Il s'agit de l'arithmétique des variables entières que l'on peut comparer et additionner, en combinant des sous-formules à l'aide d'opérateurs booléens et en quantifiant existentiellement et universellement les variables. Par exemple la formule

$$\exists x \in \mathbb{N}\ \forall y \in \mathbb{N}\ (\exists z \in \mathbb{N}\ y = z + z) \Rightarrow y < x$$

exprime que l'ensemble des naturels pairs est borné. (Cette formule est donc fausse.) Il a été démontré que le problème consistant à déterminer si une telle formule est vraie ou fausse peut être résolu algorithmiquement, mais qu'il n'existe aucun algorithme efficace capable de le faire [13]. En d'autres termes, pour tout algorithme prenant une formule en entrée et déterminant correctement si cette formule est vraie ou fausse, il existe des formules pour lesquelles le temps d'exécution de cet algorithme est prohibitivement élevé.

Pour des exemples de problèmes *indécidables*, c'est-à-dire n'admettant aucune solution algorithmique générale, on peut citer pour commencer le problème de l'arrêt déjà mentionné à la section 1.1.2, qui consiste à déterminer si l'exécution d'un programme fourni en entrée s'arrête ou non [14]. Un autre problème est celui de déterminer si une formule de la théorie additive et multiplicative des entiers est vraie ou non [15] ; il s'agit d'une extension de l'arithmétique de Presburger à laquelle on ajoute un opérateur permettant de multiplier des variables. Un dernier exemple est le *dixième problème d'Hilbert*, consistant à déterminer si un polynôme donné à plusieurs variables possède ou non un zéro entier [16]. Les concepts et les principes permettant d'établir de tels résultats sont notamment étudiés dans le cours *Theory of computation*.

13. Michael J. Fischer, Michael O. Rabin, 1974. *Super-exponential complexity of Presbuger arithmetic*. Proceedings of the SIAM-AMS Symposium in Applied Mathematics. 7 : 27–41.

14. Alan L. Turing, 1937. *On computable numbers, with an application to the Entscheidungsproblem*. Proceedings of the London Mathematical Society, s2-42 (1) : 230-–265.

15. Kurt Gödel, 1931. *Über formal unentscheidbare Sätze der Principia Mathematica und verwandter Systeme, I*. Monatshefte für Mathematik und Physik, 38 (1), 173—198.

16. Martin D. Davis, Hilary W. Putnam, Julia H. Robinson, 1961. *The decision problem for exponential Diophantine equations*. Annals of mathematics, second series, 74(3) :425–436. Yuri V. Matiyasevich, 1970. *Enumerable sets are Diophantine*. Doklady Akademii Nauk SSSR, 191(2) :279–282.

1.5 Exercices

Les algorithmes qui font l'objet des exercices suivants peuvent être exprimés en *pseudo-code*, c'est-à-dire dans un formalisme similaire à celui qui a été utilisé à la section 1.2.2 pour décrire l'algorithme d'Euclide et la méthode de Héron. La programmation en langage C de ces algorithmes fera partie des exercices proposés au chapitre 2.

1. Écrire un algorithme capable de résoudre une équation du second degré

   $$ax^2 + bx + c = 0$$

   pour des valeurs quelconques de a, b et c.

2. La *multifactorielle* $M(n, k)$ des deux entiers strictement positifs n et k est définie comme le produit $M(n, k) = n(n - k)(n - 2k) \cdots$ de tous les entiers strictement positifs m tels que $m \leq n$ et $n - m$ est divisible par k. Ecrire un algorithme capable de calculer la multifactorielle de deux nombres donnés.

3. Écrire un algorithme capable de calculer la somme, le produit, la moyenne arithmétique, le plus grand et le plus petit élément d'un ensemble (fini) quelconque de nombres réels fournis séquentiellement.

4. Un nombre entier positif est *premier* s'il possède exactement deux diviseurs : 1 et lui-même. Écrire un algorithme capable de déterminer si un nombre donné est premier ou non.

5. Écrire un algorithme capable de calculer la valeur de $\lfloor \log_2 n \rfloor$ pour un nombre entier strictement positif n donné, où $\lfloor x \rfloor$ dénote le plus grand entier inférieur ou égal à x.

Chapitre 2

Les bases du langage C

2.1 Introduction

Ce chapitre contient une première introduction au langage de programmation C, suffisante pour commencer à écrire des programmes simples. Les mécanismes plus avancés de ce langage seront ensuite étudiés aux chapitres 4, 5 et 6.

Le langage C est un langage de programmation de bas niveau, ce qui signifie que les instructions figurant dans les programmes restent proches de celles que le processeur est directement capable d'exécuter. Apprendre à programmer en commençant par le langage C permet d'acquérir une bonne compréhension des mécanismes qui régissent l'exécution des programmes, notamment la façon dont le processeur accède à la mémoire de l'ordinateur. Un autre avantage de ce langage est qu'il est simple. Cette simplicité ne fait cependant pas obstacle à son utilisation pour des projets complexes : de très nombreux logiciels de grande taille sont programmés en C. Par exemple, le noyau du système d'exploitation *Linux*, qui équipe notamment une majorité des *smartphones* modernes [1], est principalement rédigé en C, et comprend plusieurs dizaines de millions de lignes de code.

Un exemple de programme rédigé en langage C a déjà été donné au Chapitre 1, à la figure 1.2. La structure de ce programme constitue la forme la plus simple de programme C que l'on peut écrire :

```
int main()
{
  ...
}
```

1. Ceux basés sur le système d'exploitation *Android*.

Dans ce fragment de code, "`...`" ne fait pas partie de la syntaxe du programme, mais marque l'endroit où l'on peut placer les *instructions exécutables* qui appartiennent au *bloc* délimité par les accolades "`{`" et "`}`". Ce bloc est celui qui est exécuté lorsque le programme est lancé. Comme cela a déjà été expliqué, les instructions d'un bloc sont terminées par des points-virgules "`;`" et sont exécutées en séquence. De plus, le langage C est *structuré*, ce qui signifie qu'un bloc peut dans la plupart des cas prendre la place d'une instruction exécutable individuelle.

Nous allons étudier dans ce chapitre les différentes sortes d'instructions exécutables permises par le langage C. La signification exacte des éléments de la ligne de code `int main()` sera expliquée au chapitre 4. Nous verrons également dans ce chapitre comment structurer des programmes plus complexes, en répartissant leur code source dans des fichiers séparés.

2.2 Les variables

Le premier élément de langage que nous allons étudier est la *variable*. Dans un programme, une variable représente un emplacement réservé au sein de la mémoire de l'ordinateur, capable de retenir une valeur pendant l'exécution d'un programme. Nous avons vu à la section 1.2.1 qu'à un certain niveau d'abstraction, on pouvait considérer que la mémoire de l'ordinateur est composée d'un ensemble de cases capables de retenir des données. Une variable correspond à une de ces cases. Comme son nom l'indique, une variable peut généralement changer de valeur pendant l'exécution d'un programme ; celui-ci dispose d'instructions lui permettant de consulter (*lecture* en mémoire) ou de modifier (*écriture*) les variables.

Une variable est caractérisée par deux éléments d'information. Premièrement, elle possède un *identificateur*, qui est un nom permettant d'y faire référence dans le programme. En langage C, les identificateurs sont formés par une suite de symboles comprenant les lettres minuscules (de "`a`" à "`z`"), les lettres majuscules (de "`A`" à "`Z`"), le caractère de soulignement ("`_`") et les chiffres (de "`0`" à "`9`"). Un identificateur ne peut cependant pas commencer par un chiffre. Un certain nombre de mots appelés *mots-clés* sont réservés par le langage et ne peuvent pas être utilisés comme identificateurs.

Dans les identificateurs, les majuscules et les minuscules sont distinguées ; par exemple, deux variables appelées `Delta` et `delta` sont considérées comme étant distinctes. Le caractère de soulignement est souvent utilisé pour écrire des identificateurs formés par plusieurs mots, par exemple `nb_chiffres`. Une bonne pratique de programmation est de veiller à ce que le nom des variables d'un programme soit cohérent, et décrive le plus précisément possible les données qu'elles contiennent et l'utilisation qui en est faite.

En plus d'être caractérisée par son identificateur, une variable possède un *type*. Le type d'une variable détermine la nature des valeurs que cette variable peut prendre, ainsi que la façon dont ces valeurs sont représentées dans la mémoire de l'ordinateur. Les opérations de traitement de

données que l'on peut appliquer au contenu d'une variable dépendent du type de celle-ci. Les types que l'on peut employer sont très variés. Pour des données simples telles que des nombres entiers ou réels, on utilise des *types de base*, ou *types primitifs*, qui sont prédéfinis dans le langage. Le programmeur a également la possibilité de définir ses propres types adaptés à des données plus complexes. Les mécanismes permettant de manipuler de tels types seront abordés aux chapitres 5 et 7.

2.2.1 Les types de base

Les principaux types de base du langage C sont les suivants.

`char`	Caractère, ou petit entier
`int`	Entier
`float`	Nombre réel (simple précision)
`double`	Nombre réel (double précision)

Ces types ne sont pas précisément définis ; leurs détails d'implémentation peuvent différer d'un environnement de programmation à un autre. Dans ce cours, nous allons souvent prendre l'exemple de l'architecture x86-64, qui est celle de la plupart des ordinateurs personnels actuels, mais il ne faut pas être surpris de constater des différences si l'on programme dans un autre environnement.

Le type `char`

Le type de base `char` permet de représenter un *caractère*, c'est-à-dire un symbole dans un texte. Il correspond aussi dans les versions modernes du langage C à la plus petite unité adressable de la mémoire. Intuitivement, dans le modèle abstrait d'ordinateur que nous avons introduit à la section 1.2.1, on peut considérer qu'il s'agit d'une case de la mémoire.

Dans la très grande majorité des architectures de processeurs actuellement utilisées, l'unité élémentaire de mémoire contient 8 bits d'information, et donc le type `char` permet de représenter une valeur de 8 bits, aussi appelée *octet*. Un *bit* correspond à l'information contenue dans une variable binaire dont les deux valeurs possibles sont équiprobables. Une donnée représentée sur 1 bit possède donc deux valeurs possibles. Plus généralement, une donnée représentée sur n bits peut prendre $2 \times 2 \times \cdots \times 2 = 2^n$ valeurs distinctes.

Pour la plupart des architectures, une variable de type `char` peut donc prendre $2^8 = 256$ valeurs. Ces valeurs peuvent indifféremment être vues commes des nombres entiers (nous en préciserons l'intervalle un peu plus loin), ou comme des caractères. La correspondance entre ces entiers et les caractères est établie par l'intermédiaire d'une table dont les détails dépendent de

l'environnement utilisé. La majorité des systèmes informatiques actuels emploient une table de correspondance dérivée du *code ASCII (American Standard Code for Information Interchange)*. Dans ce code, par exemple, le caractère "`a`" possède le code 97. Cela signifie que si une variable de type `char` contient la valeur 97, alors le programmeur peut décider de l'interpréter comme représentant l'entier 97 ou bien le caractère "`a`", selon les besoins du programme qu'il est en train de rédiger.

Bien sûr, 256 caractères ne suffisent pas à représenter tous les symboles qui peuvent apparaître dans un texte. Des procédés de représentation permettant d'encoder de très grands alphabets à l'aide d'un nombre variable d'octets existent mais sortent du cadre de ce cours [2]. À ce stade, il est suffisant de considérer que les caractères que l'on est amené à traiter appartiennent à un alphabet réduit comprenant moins de 256 symboles. Ces symboles comprennent en particulier ceux dont on a besoin pour programmer en C, ainsi que ceux qui sont nécessaires pour écrire des textes en français ou en anglais.

Le type `int`

Le type `int` est probablement le type de base le plus important, et sert à représenter une valeur entière. La norme du langage C n'impose pas la façon dont la valeur d'une variable de type `int` est représentée en mémoire ; cela permet aux concepteurs d'un compilateur de choisir la représentation qui est la plus facilement manipulable par le processeur, et donc la plus efficace. Dans une très grande majorité des environnements de programmation en langage C pour l'architecture x86-64, une variable de type `int` contient 32 bits d'information, et peut donc prendre 2^{32} valeurs distinctes.

Par défaut, une variable de type `int` est *signée*, ce qui signifie qu'elle peut prendre des valeurs aussi bien positives que négatives. Nous n'allons pas dans ce cours aborder les détails de la représentation interne des valeurs signées[2], mais seulement mentionner qu'avec une représentation sur 32 bits, un entier signé peut prendre n'importe quelle valeur dans l'intervalle

$$\begin{aligned} &[-2^{31},\, 2^{31}-1] \\ =\ &[-2147483648,\, 2147483647]. \end{aligned}$$

Pour certaines applications, il n'est cependant pas utile de considérer des valeurs négatives. On peut alors placer le modificateur `unsigned` avant le type `int` ou `char`, afin signifier que les valeurs doivent être *non signées*, c'est-à-dire se limiter aux entiers positifs ou nuls. Le modificateur `signed` existe également pour indiquer explicitement qu'un type doit être signé.

Pour des entiers sur 32 bits, le type `unsigned int` permet de représenter les nombres ap-

2. Ils seront étudiés dans le cours d'*Organisation des ordinateurs*.

partenant à l'intervalle

$$\begin{aligned}&[0,\ 2^{32}-1]\\ =\ &[0,\ 4294967295].\end{aligned}$$

Les types `unsigned char` et `signed char` correspondent quant à eux respectivement à

$$\begin{aligned}&[0,\ 2^{8}-1]\\ =\ &[0,\ 255].\end{aligned}$$

et

$$\begin{aligned}&[-2^{7},\ 2^{7}-1]\\ =\ &[-128,\ 127].\end{aligned}$$

Notons que le type `int` est signé par défaut, et que `signed` et `unsigned` employés seuls sont équivalents (respectivement) à `signed int` et `unsigned int`.

Il existe aussi d'autres modificateurs que l'on peut appliquer au type `int`, qui servent à augmenter (`long` ou `long long`) ou diminuer (`short`) la taille des représentations. Leur effet exact dépend de l'architecture et de l'environnement de programmation utilisés. Par exemple, pour l'architecture x86-64, le type `long long int` représente un entier sur 64 bits. Ces modificateurs peuvent être combinés avec `signed` ou `unsigned`, par exemple, les valeurs de type `unsigned long long int` appartiennent à l'intervalle

$$\begin{aligned}&[0,\ 2^{64}-1]\\ =\ &[0,\ 18446744073709551615].\end{aligned}$$

Signalons enfin que quand on utilise ces modificateurs, le type `int` est implicite et peut être omis. Par exemple, on peut abréger `unsigned long long int` en `unsigned long long`.

Les types `float` et `double`

Les types de base `float` et `double` servent à représenter des nombres réels. Comme cela a été discuté à la section 1.2.2, ces représentations ne sont pas exactes, mais approximent les nombres à une certaine précision près. Il est important de toujours garder cela à l'esprit quand on manipule des variables réelles.

Les types réels `float` et `double` correspondent respectivement à des représentations en *simple* et en *double précision*. Leur implémentation peut varier selon l'environnement de programmation, mais le standard le plus couramment employé, appelé *IEEE754*, correspond à des représentations sur respectivement 32 et 64 bits. De nombreux processeurs modernes possèdent

des circuits permettant de manipuler efficacement des valeurs de type `double` ; il s'agit donc du type que l'on utilisera le plus souvent pour représenter des réels. Ce type représente les nombres avec une précision supérieure à 15 chiffres décimaux significatifs, ce qui suffit largement pour la plupart des applications. Signalons enfin qu'il existe aussi le type `long double` permettant d'encore augmenter la précision des représentations

2.2.2 La déclaration des variables

Pour pouvoir utiliser une variable dans un programme, il faut préalablement la *déclarer*. En langage C, l'instruction de déclaration d'une ou de plusieurs variables prend la forme suivante.

[`const` *] type identificateur [= valeur]*
[, identificateur [= valeur]]
⋮
[, identificateur [= valeur]] `;`

Dans une telle spécification de syntaxe, les crochets en italique "*[*" et "*]*" délimitent des éléments qui sont optionnels. La forme la plus simple de définition de variable est donc

type identificateur `;`

où *type* et *identificateur* sont bien sûr le type et l'identificateur de la variable déclarée. La forme générale de définition de variables permet de déclarer dans une même instruction plusieurs variables partageant le même type, et de leur attribuer une valeur initiale. Le mot-clé `const` permet de déclarer des *constantes*, c'est-à-dire des variables dont la valeur ne peut pas être modifiée.

Des exemples de déclarations de variables sont donnés ici :

```
int            i = 0, j = -1, k;
unsigned long  code_barre;
char           symbole = 'a', marqueur = '\n';
const double   avogadro = 6.02214179e23;
```

Dans cet exemple, toutes les variables sauf `k` et `code_barre` sont initialisées. Les expressions employées pour initialiser `symbole` et `marqueur` montrent la syntaxe particulière des constantes de type caractère : l'évaluation de l'expression `'a'` produit comme valeur le code du caractère "a" dans le jeu de symboles utilisé. Comme nous l'avons vu précédemment, ce code vaut 97 dans la plupart des environnements de programmation, donc les expressions `'a'` et `97` sont équivalentes. Le symbole "\" qui apparaît dans l'expression d'initialisation de `marqueur` est un

caractère d'échappement. Il permet, en le faisant suivre d'un autre caractère, de spécifier des *caractères spéciaux* qui ne sont pas imprimables mais provoquent une action particulière quand ils sont affichés. Par exemple, le caractère désigné par "\n" représente un *retour chariot* et un *saut de ligne* ; en d'autres termes, il termine la ligne courante de texte et passe à la ligne suivante. Pour spécifier le caractère "\" lui-même, on écrit "\\".

Pour terminer, l'expression d'initialisation de la constante `avogadro` montre comment écrire des constantes réelles en notation scientifique.

2.2.3 La portée d'une déclaration

En langage C, la déclaration d'une variable doit figurer avant les instructions qui utilisent cette dernière. Cette déclaration sert à indiquer au compilateur comment la variable doit être représentée en mémoire et comment elle doit être manipulée. Par exemple, une opération d'addition n'est pas implémentée de la même façon selon que ses opérandes sont des nombres entiers ou des nombres réels.

Lorsqu'une variable est déclarée au sein d'un programme, la *portée* de cette variable désigne l'ensemble des endroits du programme où cette variable peut être utilisée. En langage C, la portée d'une variable commence à sa déclaration, et se termine à la fin du plus petit bloc qui contient cette déclaration. Il est permis que plusieurs variables déclarées dans des blocs différents partagent le même identificateur ; dans ce cas, la déclaration la plus interne masque celles qui figurent dans les blocs extérieurs.

Pour illustrer cela, nous considérons le fragment de programme de la figure 2.1. Nous ne nous intéressons qu'aux déclarations de variables, en remplaçant les autres instructions du programme par des pointillés "⋮". Les numéros figurant au début des lignes de code ne font pas partie de la syntaxe du programme, mais sont ajoutés pour permettre de faire facilement référence aux instructions.

Ce morceau de programme définit plusieurs blocs imbriqués. L'instruction située à la ligne 8 déclare une variable appelée `x`, de type `long`. La portée de cette variable commence juste après cette déclaration, et se termine à la fin du plus petit bloc qui la contient, c'est-à-dire à la ligne 10. Cela signifie que l'emplacement de mémoire correspondant à la variable est alloué à la ligne 8, et libéré à la ligne 10. Les instructions qui peuvent faire référence à cette variable sont celles situées à la ligne 9 du programme.

Le programme définit également à la ligne 4 une autre variable appelée `x`, cette fois de type `int`. La portée de cette variable commence après cette déclaration, et se termine à la ligne 12. Remarquons qu'aux lignes 9 et 10, cette variable est masquée par cette déclarée à la ligne 8. Les instructions qui peuvent utiliser la variable déclarée à la ligne 4 sont donc celles figurant aux lignes de code 5, 6, 7 et 11. Cette variable est néanmoins toujours présente en mémoire même

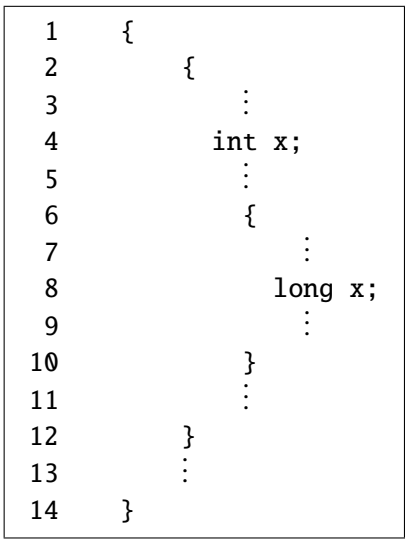

FIGURE 2.1 – Portée d'une variable

quand elle est masquée : si une instruction située à la ligne 5 y écrit une valeur, alors cette valeur pourra être lue à la ligne 11. L'emplacement de mémoire alloué à cette variable n'est libéré qu'à la ligne 12.

Pour des raisons de lisibilité, il est conseillé de placer les définitions de variables au début de leur bloc. Il ne s'agit cependant pas d'une contrainte imposée par le langage C.

2.3 Les expressions

Les expressions sont une catégorie particulière d'instructions exécutables, servant à exprimer des opérations de traitement de données. Lorsqu'une expression est *évaluée*, ces opérations sont effectuées, et fournissent une valeur qui constitue le résultat de cette évaluation.

Les expressions les plus simples que l'on peut écrire sont les *littéraux*. Il s'agit de valeurs constantes, par exemple `42`, `3.142592654` ou `'a'`. Leur évaluation fournit la valeur correspondante.

L'identificateur d'une variable forme également une expression, par exemple `avogadro` ou `nb_chiffres`. Bien sûr, pour qu'une telle expression soit valide, il faut qu'elle soit située dans la portée de la variable correspondante. L'évaluation d'une telle expression fournit la valeur courante de cette variable ; en d'autres termes, cette évaluation effectue une lecture de la variable.

On peut construire des expressions plus complexes en appliquant des *opérateurs* à des sous-

expressions. Par exemple, l'expression

```
4.0 / 3.0
```

s'obtient en appliquant l'opérateur de division arithmétique "/" aux deux sous-expressions littérales `4.0` et `3.0`. L'évaluation de cette expression fournit la valeur réelle 1,333333 . . .

L'expression

```
(4.0 / 3.0) * pi * r * r * r
```
(2.1)

applique l'opérateur de multiplication arithmétique "*" aux sous-expressions `(4.0 / 3.0)`, qui possède la même valeur que celle de `4.0 / 3.0`, et `pi`, qui lit la valeur d'une variable nommée `pi`. Le résultat forme la sous-expression

```
(4.0 / 3.0) * pi
```
.

Ensuite, l'opérateur "*" est appliqué successivement trois fois à l'expression obtenue et à la sous-expression `r`. En supposant que la variable `pi` contienne une approximation de π, et que `r` représente le rayon d'une sphère, l'évaluation de l'expression (2.1) calcule donc le volume de cette sphère.

Nous allons à présent passer en revue les principaux opérateurs définis par le langage C.

2.3.1 Les opérateurs arithmétiques

Les opérateurs figurant dans le tableau suivant permettent d'effectuer des additions, des soustractions, des multiplications et des divisions.

`+`	Addition
`-`	Soustraction
`*`	Multiplication
`/`	Division
`%`	Reste de la division (modulo)

La nature de l'opération effectuée dépend du type de ses opérandes. Par exemple, l'évaluation de l'expression

```
2 / 3
```

calcule la *division entière* de 2 et de 3, qui vaut 0. En revanche, celle de l'expression

```
2.0 / 3.0
```

retourne 0,666666 . . ., car l'opérateur de division est maintenant appliqué à des valeurs réelles.

L'opérateur "%" n'est applicable qu'à des entiers ; il retourne le reste de la division de sa première opérande par la deuxième. (Nous avons déjà vu un exemple d'application de cet opérateur dans l'implémentation de l'algorithme d'Euclide au chapitre 1.) Cet opérateur correspond à l'opération mathématique de *modulo* quand il est appliqué à des valeurs positives [3].

Lorsqu'une expression comprend plusieurs opérateurs, ceux-ci sont évalués dans un ordre qui dépend de leur *priorité*, ou *précédence*. Celle-ci est définie d'une façon similaire aux règles de l'algèbre : Les opérateurs "*", "/" et "%" sont plus prioritaires que "+" et "-". Des parenthèses peuvent être utilisées pour forcer explicitement un ordre d'évaluation des opérateurs. Par exemple, les expressions

```
a * a + b * b
```

et

```
(a * a) + (b * b)
```

sont équivalentes.

Enfin, les opérateurs "+" et "-" possèdent également une forme *unaire*, c'est-à-dire qui n'admet qu'une seule opérande. Par exemple, si `x` est une variable entière ou réelle, l'expression `-x` calcule l'opposé de la valeur de `x`.

2.3.2 Les opérateurs de comparaison

Les opérateurs suivants permettent de comparer des valeurs.

<	Plus petit que
>	Plus grand que
<=	Plus petit ou égal à
>=	Plus grand ou égal à
==	Égal à
!=	Différent de

Leur évaluation retourne une *valeur booléenne*, c'est-à-dire qui est soit vraie, soit fausse. En langage C, la valeur booléenne vraie est représentée [4] par n'importe quel entier différent de 0, et fausse par l'entier 0. Par exemple, l'évaluation de

```
5 <= 2
```

3. Le reste de la division diffère du modulo pour des opérandes négatives. Par exemple, on a $-7 \bmod 3 = 2$, alors que le reste de la division de -7 par 3 vaut -1.

4. Les versions modernes du langage C définissent un type de base dédié aux valeurs booléennes. Nous ne l'emploierons pas dans ce cours, car il n'est pas universellement utilisé.

retourne 0. Celle de

```
10 == 5 + 5
```

retourne un entier non nul. (La valeur de celui-ci n'est pas précisément connue, et peut potentiellement varier selon l'environnement de programmation, ou d'une évaluation à une autre.) Notons que l'opérateur de test d'égalité s'écrit avec deux symboles "="; nous verrons à la section 2.3.4 que celui formé d'un seul symbole "=" possède une sémantique différente. La confusion entre ces deux opérateurs est une erreur de programmation fréquente en C.

2.3.3 Les opérateurs booléens

On est souvent amené à devoir combiner plusieurs tests au sein d'une même expression. Les opérateurs suivants sont destinés à être appliqués à des valeurs booléennes, en d'autres termes, à des entiers dont on s'intéresse seulement au fait qu'ils sont nuls ou non nuls.

`&&`	Et logique
`||`	Ou logique
`!`	Négation (opérateur unaire)

L'évaluation de l'opérateur "`&&`" retourne une valeur booléenne vraie si et seulement si ses deux opérandes sont vraies. Par exemple, l'expression

```
c >= 'a' && c <= 'z'
```

teste si la valeur de la variable `c` est une lettre minuscule, en faisant l'hypothèse que ces lettres apparaissent dans l'ordre du dictionnaire et de façon consécutive dans le jeu de caractères utilisé. (Cette propriété est vraie pour tous les systèmes informatiques modernes.)

L'opérateur "`||`" détermine si au moins une de ses deux opérandes est vraie. Par exemple, l'expression

```
x >= 0 || y >= 0
```

teste si les deux variables `x` et `y` ne possèdent pas simultanément une valeur strictement négative.

L'opérateur unaire "`!`" transforme une valeur vraie en une valeur fausse et réciproquement. Par exemple, si `x` est une variable entière [5], les expressions

```
!(x >= 1000)
```

5. Cette équivalence n'est pas vraie pour une variable réelle, qui peut prendre des *valeurs exceptionnelles* qui ne sont pas des nombres, et ne sont donc ni supérieures, ni inférieures à un nombre donné.

et

```
x < 1000
```

sont équivalentes.

Les opérateurs "&&" et "||" présentent une particularité importante : ils s'évaluent en *circuit court*. Cela signifie que leur évaluation s'effectue de gauche à droite (en d'autres termes, leur opérande de gauche est évaluée avant celle de droite), et se termine dès que la valeur finale de l'expression est connue. Par exemple, l'évaluation de

```
n != 0 && m / n > 1
```
(2.2)

commence par évaluer la sous-expression `n != 0` qui constitue l'opérande de gauche de l'opérateur "&&". Si le résultat obtenu est faux, alors il n'est pas nécessaire d'évaluer l'opérande de droite de "&&", car on sait déjà que l'évaluation de (2.2) produira un résultat faux. En revanche, si l'évaluation de `n != 0` retourne un résultat vrai, alors il devient nécessaire d'évaluer aussi la sous-expression `m / n > 1` pour déterminer le résultat de l'évaluation de (2.2). Ce mécanisme permet donc ici d'éviter d'effectuer une division par zéro lorsque la valeur de `n` est nulle. (L'effet d'une telle division est indéterminé, et peut aller jusqu'à interrompre l'exécution du programme.)

2.3.4 Les opérateurs d'affectation

L'opérateur d'*affectation* "=" permet d'écrire une valeur dans une variable. Son opérande de gauche est ce que l'on appelle une *valeur à gauche*, et sert à spécifier l'endroit dans la mémoire de l'ordinateur où l'opération d'écriture va être effectuée. Dans sa forme la plus simple, il s'agit de l'identificateur d'une variable. L'opérande de droite fournit la valeur qui sera écrite lors de l'évaluation de l'expression. Cette valeur est aussi celle qui constitue le résultat de cette évaluation.

Par exemple, pour évaluer l'expression

```
x = x + 2
```
,

on commence par évaluer l'opérande de droite `x + 2` de l'opérateur "=", ce qui revient à lire la valeur de `x` et lui ajouter 2. Ensuite, le résultat obtenu est écrit dans la variable `x`. En résumé, l'évaluation de cette expression a *incrémenté* la valeur de `x` de deux unités. Le résultat de cette évaluation correspond à la valeur écrite dans `x`. Il s'agit d'une évaluation qui présente un *effet de bord* : son effet ne consiste pas seulement à produire une valeur, mais aussi de réaliser une autre opération (modifier la valeur de `x`).

L'évaluation de l'expression

```
(a = b) < 10
```

recopie la valeur de la variable `b` dans `a`, et teste ensuite si cette valeur est strictement inférieure à 10. L'évaluation de l'expression

```
i = j = k = 0
```

affecte la valeur 0 à `k`. Ensuite, cette valeur est affectée à `j`, puis de la même façon à `i`. Cette expression permet donc d'initialiser les trois variables avec la même valeur 0.

Il existe d'autres opérateurs d'affectation qui offrent des raccourcis d'écriture, il s'agit des opérateurs "+=", "-=" "*=", "/=" et "%=". Le principe est simple : une expression de la forme

var α= *expr*,

avec $\alpha \in \{+, -, *, /, \%\}$, est équivalente à

var = *var* α *expr*.

Par exemple, l'expression

```
x += 2
```

équivaut à

```
x = x + 2
```

.

2.3.5 Les opérateurs d'incrément et de décrément

Les opérateurs d'*incrément* "++" et de *décrément* "--" sont des opérateurs unaires qui s'appliquent à des valeurs à gauche, par exemple l'identificateur d'une variable. Il peuvent être placés à droite ou à gauche de leur opérande, leur sémantique étant différente dans les deux cas.

Ces opérateurs présentent un effet de bord qui consiste à incrémenter ("++") ou décrémenter ("--") leur opérande. S'ils sont placés à droite, ils retournent la valeur de cette opérande avant l'opération d'incrément ou de décrément. S'ils sont placés à gauche il retournent la valeur de l'opérande après son incrément ou son décrément.

Par exemple, si l'on suppose que `x` et `y` sont deux variables entières initialisées à 0, l'évaluation de l'expression

```
x = y++
```

commence par évaluler la sous-expression `y++`, ce qui a pour effet d'incrémenter `y`. La valeur de cette variable devient donc égale à 1. Étant donné que l'opérateur "++" est placé à droite, l'évaluation de `y++` retourne la valeur de `y` avant l'opération d'incrément, c'est-à-dire 0. C'est cette valeur 0 qui est ensuite affectée à `x`.

En supposant toujours que `x` et `y` sont initialisées à 0, l'évaluation de l'expression

```
x = --y
```

décrémente `y`, dont la valeur devient donc égale à -1, et écrit ensuite cette valeur dans `x`, qui prend donc aussi la valeur -1.

Les opérateurs d'incrément et de décrément permettent d'écrire des expressions qui sont syntaxiquement correctes, mais dont la sémantique n'est pas précisément définie par la norme du langage C, par exemple

```
x = --x + x++
```

.

L'évaluation de telles expressions retourne un résultat indéfini. Leur usage est donc à proscrire.

Étant donné que les opérateurs d'incrément et de décrément présentent un effet de bord, leur effet dépend du fait qu'ils sont évalués ou non. Par exemple, l'évaluation de l'expression

```
x > 0 && --x
```

décrémente `x` seulement si sa valeur est initialement strictement positive, car l'opérateur "`&&`" fonctionne en circuit court comme expliqué à la section 2.3.3. L'évaluation de cette expression retourne une valeur booléenne vraie si et seulement si la valeur initiale de `x` est strictement supérieure à 0 et différente de 0 après avoir été décrémentée, c'est-à-dire si elle est strictement supérieure à 1.

2.3.6 L'opérateur virgule

L'opérateur virgule "`,`" n'est pas indispensable, mais permet de simplifier certaines constructions. Il est utilisé lorsque l'on souhaite évaluer plusieurs expressions dans un contexte où la syntaxe du langage C ne permet que d'en écrire une seule.

Cet opérateur prend deux opérandes et évalue d'abord celle de gauche et puis celle de droite. Le résultat prend le type et la valeur de l'opérande de droite.

Considérons par exemple l'expression

```
x = (y++, y++)
```

où `x` et `y` sont deux variables entières initialisées à 0. Cette expression emploie des parenthèses car la priorité de l'opérateur "`,`" est inférieure à celle de "=".

L'évaluation du membre de droite de l'opérateur "=" va commencer par évaluer la sous-expression `y++` située à gauche de "`,`", ce qui va incrémenter `y` et retourner 0. Ensuite, on évalue l'opérande de droite `y++` de "`,`", ce qui incrémente à nouveau `y` et retourne 1. Cette derrnière valeur est celle qui constitue le résultat de l'évaluation de `y++, y++`. En définitive, `x` devient donc égal à 1 et `y` à 2.

2.3.7 La conversion de type

Chaque expression possède un *type*, qui caractérise la nature des valeurs produites lors de l'évaluation de cette expression, ainsi que les opérations que l'on peut faire subir à ces valeurs. Les principaux types de base définis par le langage C ont été introduits à la section 2.2.1.

Il est parfois permis, sous certaines conditions, d'affecter à une variable d'un type donné une valeur d'un autre type. Cela est par exemple autorisé pour tous les types numériques, c'est-à-dire les types entiers et réels ainsi que le type `char` qui est considéré comme un cas particulier de type entier. Dans ce cas, une opération de *conversion de type* est implicitement réalisée.

Par exemple, le fragment de code

```
double p = 3.1416;
int x;

x = p;
```

déclare une variable réelle `p` et une variable entière `x`, et affecte la valeur de la première à la deuxième. Cette opération provoquera la conversion de la valeur réelle 3,1416 en un entier, c'est-à-dire son arrondi vers la valeur 3, qui sera celle écrite dans `x`.

Dans certains cas, on souhaite spécifier explicitement qu'une telle conversion de type doit être effectuée. On emploie pour cela l'opérateur de conversion de type, dont la syntaxe est

(*type*) *expr* .

Cette expression s'évalue en évaluant la sous-expression *expr*, et en convertissant ensuite le résultat obtenu vers le type *type*. Le procédé utilisé pour convertir une valeur d'un type vers un autre dépend des types impliqués ; la spécification du langage C contient une description détaillée de cette opération pour un certain nombre de paires de types.

Pour illustrer cette opération, nous considérons la séquence d'instructions suivante.

```
double a = 2.0, b = 3.0, x, y;

x = a / b;
y = (int) a / (int) b;
```

Ces instructions affectent à la variable réelle `x` le résultat de la division de `a` par `b`. Étant donné que ces deux variables sont déclarées avec le type `double`, l'opération qui va être effectuée est une division réelle, et `x` va prendre la valeur 0,666666... qui est le quotient de 2 par 3.

La deuxième instruction d'affectation est différente : elle convertit d'abord le résultat de l'évaluation des deux sous-expressions `a` et `b` vers des entiers avant de leur appliquer l'opérateur de division "/". L'opération effectuée sera donc une division entière de 2 par 3, dont le résultat 0 sera la valeur écrite dans `x`. Remarquons que dans la sous-expression `(int) a / (int) b`, la priorité de l'opérateur de conversion de type est supérieure à celle de l'opérateur de division. De plus, l'évaluation de cette sous-expression produit un résultat de type entier. Une opération de conversion de ce résultat vers un réel est donc implicitement effectuée avant de l'affecter à `y`.

2.3.8 Les expressions conditionnelles

On va maintenant introduire un opérateur qui, comme l'opérateur virgule, n'est pas essentiel mais permet parfois de simplifier l'écriture de certaines instructions. Une *expresssion conditionnelle* admet la syntaxe suivante.

cond ? *expr1* : *expr2*

Cette expression comporte un seul opérateur, représenté par les deux symboles "`?`" et "`:`", appliqué aux trois sous-expressions *cond*, *expr1* et *expr2*.

L'évaluation de cette expression commence par l'évaluation de *cond*, dont le résultat est interprété comme une valeur booléenne. Si cette valeur est vraie (en d'autres termes, si elle est différente de 0), alors *expr1* est évaluée, sinon c'est *expr2* qui l'est. Dans tous les cas, il n'y a qu'une et une seule sous-expression parmi *expr1* et *expr2* qui est évaluée. Le résultat de cette évaluation est celui qui est retourné par l'évaluation de l'expression complète.

Par exemple, l'expression

```
max = (a > b) ? a : b
```

s'évalue en déterminant d'abord si la condition `a > b` est vraie. Si c'est le cas, alors l'évaluation de la sous-expression `(a > b) ? a : b` retourne la valeur de `a`, sinon celle de `b`. L'expression complète a donc pour effet d'affecter à `max` la plus grande valeur parmi celles de `a` et de `b`.

Les sous-expressions qui apparaissent dans une expresssion conditionnelle peuvent produire des effets de bord. Par exemple, l'évaluation de l'expression

```
incr_x ? x++ : y++
```

incrémente `x` si la valeur de `incr_x` est vraie, et `y` dans le cas contraire.

2.4 Les instructions de contrôle

Nous avons vu que, par défaut, les instructions qui composent un bloc sont exécutées séquentiellement, dans l'ordre où elles apparaissent. Ce mécanisme n'est cependant pas suffisant pour implémenter la plupart des algorithmes ; on a besoin de pouvoir exprimer que certaines instructions doivent être exécutées plusieurs fois, ou que leur exécution doit être conditionnée au résultat d'une opération précédemment effectuée. Par exemple, dans l'algorithme d'Euclide étudié à la section 1.3.3, l'opération consistant à permuter la valeur des deux variables `a` et `b` ne doit être effectuée que si la condition `a > b` est satisfaite. De plus, dans la suite de cet algorithme, les opérations de calcul constituant une itération doivent être répétées tant que la valeur de `a` est différente de 0.

Les *instructions de contrôle* sont celles qui permettent de modifier l'ordre séquentiel normal des instructions exécutables d'un bloc. Il en existe de plusieurs types.

2.4.1 Le choix conditionnel binaire

En langage C, l'instruction de *choix conditionnel binaire* possède la syntaxe suivante.

```
if (expr)
    instr1 ;
[ else
    instr2 ; ]
```

Pour rappel, les les crochets en italique "*[*" et "*]*" délimitent des éléments optionnels. Dans le cas présent, cela signifie que la partie de l'instruction commençant par le mot-clé `else` peut être omise. Dans cette spécification de syntaxe, *expr* représente une expression s'évaluant en une valeur entière, et *instr1* et *instr2* sont des instructions exécutables, pouvant être remplacées par des blocs. Remarquons que les parenthèses qui entourent *expr* font partie de la syntaxe de l'instruction et sont obligatoires.

Lorsque cette instruction est exécutée, elle commence par évaluer *expr*, et interprète le résultat de cette évaluation comme une valeur booléenne. Si celle ci est vraie, alors l'instruction *instr1* est exécutée. Sinon, l'instruction *instr2* est exécutée, à condition que la partie `else` de l'instruction soit présente. (Dans le cas contraire, l'instruction ne fait rien d'autre après avoir évalué *expr*.)

On peut représenter graphiquement la sémantique d'une telle instruction à l'aide d'un *organigramme*, qui est un diagramme montrant les différents chemins d'exécution possible des opérations. L'organigramme de l'instruction `if` est donné à la figure 2.2. Pour rappel, une valeur booléenne est considérée comme étant vraie si sa représentation entière est différente de 0.

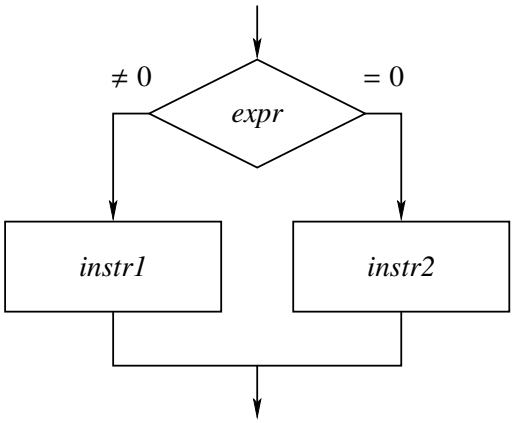

FIGURE 2.2 – Organigramme de l'instruction `if`

En langage C, le mot-clé `else` ne peut être utilisé que pour commencer la deuxième partie d'une instruction `if`. Quand on utilise `else`, il faut toujours garder à l'esprit qu'il se rapporte toujours au mot-clé `if` le plus récent du même bloc (à condition que celui-ci ne possède déjà pas une partie `else`). Il s'agit d'une source d'erreur fréquente. Considérons par exemple le fragment de programme suivant.

```
if (expr1)
    if (expr2)
        {
            ⋮
        }
else
    {
        ⋮
    }
```

Dans cet exemple, le mot-clé `else` se rapporte au deuxième `if` et non au premier. L'indentation laisse cependant supposer que ce n'était pas l'intention du programmeur qui a rédigé ce code. Pour que le mot-clé `else` se rapporte au premier `if`, il aurait fallu écrire, par exemple :

```
if (expr1)
  {
    if (expr2)
        {
            ⋮
        }
  }
else
    {
        ⋮
    }
```

Pour illustrer l'utilisation de l'instruction `if`, nous allons montrer comment rédiger un programme C capable de déterminer si une année donnée est bissextile ou non. Les années bissextiles sont celles qui comprennent un 29 février. Elles sont ajoutées au calendrier de façon à rendre le nombre moyen de jours d'une année le plus proche possible de la durée d'un cycle complet des saisons.

La règle de base est qu'une année est bissextile si elle est divisible par 4. Sur le long terme, cette règle ne conduit cependant pas à une approximation suffisamment précise de la durée moyenne d'une année. On introduit donc une exception : les années divisibles par 100 ne sont pas bissextiles, sauf celles divisibles par 400.

On voit donc que pour déterminer si une année donnée est bissextile, il faut considérer 4 cas possibles : elle n'est pas divisible par 4, elle est divisible par 4 sans être divisible par 100, elle est divisible par 100 sans être divisible par 400, et elle est divisible par 400. Un programme C implémentant ce processus de décision est donné à la figure 2.3.

La structure de ce programme et l'utilisation des fonctions `printf` et `scanf` sont semblables à celles de l'implémentation de l'algorithme d'Euclide à la section 1.3.3. La procédure de décision est implémentée en déterminant d'abord si l'année est divisible par 400, puis par 100, puis par 4 ; cette stratégie permet de minimiser le nombre de tests effectués. Pour tester si l'année est divisible par n, on évalue une expression de la forme

`!(annee % n)` .

En effet, si c'est le cas, alors le reste de la division d'`annee` par n vaut 0, qui représente une valeur booléenne fausse. L'opérateur "`!`" transforme alors cette valeur en une valeur vraie. Réciproquement, si `annee` n'est pas divisible par n, alors l'expression s'évalue en une valeur booléenne fausse.

Les deux premières instructions `if` du programme ont donc pour effet de placer dans la variable `est_bissextile` une valeur booléenne vraie ou fausse, selon que l'année est bissextile

```
#include <stdio.h>

int main()
{
  int annee, est_bissextile;

  printf("Entrez une année: ");
  scanf("%d", &annee);

  if (!(annee % 400))
    est_bissextile = 1;
  else
    if (!(annee % 100))
      est_bissextile = 0;
    else
      est_bissextile = !(annee % 4);

  printf("Cette année ");

  if (est_bissextile)
    printf("est");
  else
    printf("n'est pas");

  printf(" bissextile.\n");
}
```

FIGURE 2.3 – Détermination d'une année bissextile

ou non. La troisième et dernière instruction `if` sert alors à afficher le message approprié, en fonction de la valeur contenue dans `est_bissextile`.

2.4.2 Le choix conditionnel multiple

L'instruction `switch` permet d'évaluer une expression et d'ensuite aiguiller l'exécution du programme vers différents endroits d'un bloc en fonction du résultat obtenu. Sa syntaxe est la suivante.

```
switch (expr)
  {
    case val1:
      seq1;
      break;
    case val2:
      seq2;
      break;
        ⋮
  [ default:
      seqn; ]
    }
```

Le nombre d'occurrences de "`case`" est quelconque, et la présence de "`default:`" est optionnelle. L'expression *expr* doit être de type entier (en ce compris `char` qui est considéré comme un cas particulier de type entier). Les littéraux *val1*, *val2* doivent être des constantes entières. Les éléments *seq1*, *seq2*, ... de l'instruction sont des séquences [6] d'instructions exécutables.

Les étiquettes "`case` *val1*`:`", "`case` *val2*`:`", ... et "`default:`" peuvent être vues comme des *points d'entrée* dans le bloc qui suit "`switch (`*expr*`)`"; ce bloc constitue le *corps* de cette instruction. L'instruction `switch` fonctionne de la façon suivante : après avoir évalué *expr*, si la valeur obtenue est égale à *val1*, alors l'exécution du corps commence à l'endroit spécifié par cette étiquette, c'est-à-dire par *seq1*. Si l'évaluation de *expr* retourne *val2*, on entre dans le corps après l'étiquette correspondante, c'est-à-dire en exécutant *seq2*, et ainsi de suite en fonction du nombre d'étiquettes présentes. Si la valeur retournée par l'évaluation de *expr* est différente de *val1*, *val2*, ..., alors on entre dans le corps après "`default:`", à condition que cette étiquette figure dans l'instruction. Dans le cas contraire, aucune instruction du corps n'est exécutée.

6. La spécification du langage C autorise en fait une forme plus générale pour le corps d'une instruction `switch`, mais celle étudiée ici suffit dans la très grande majorité des cas.

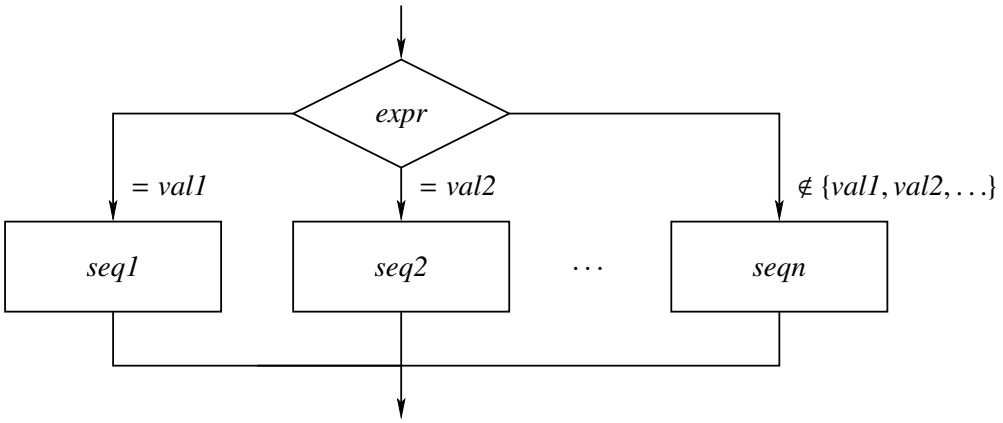

FIGURE 2.4 – Organigramme de l'instruction `switch`

Les instructions `break` qui séparent les différentes parties du corps ont pour effet de terminer l'exécution de celui-ci. (Nous reviendrons sur cette instruction à la section 2.4.4.) Si, par exemple, l'évaluation de *expr* retourne *val2*, alors cela garantit que seule la séquences *seq2* sera exécutée. Il est permis d'omettre ces instructions `break` lorsque l'on souhaite que l'exécution d'une séquence soit suivie par celle de la séquence suivante. Le plus souvent, ce n'est pas l'intention du programmeur, et l'oubli d'un `break` entre deux parties du corps d'un `switch` est une erreur de programmation fréquente. (C'est pour cette raison que les instructions `break` ne sont pas indiquées comme étant optionnelles dans la syntaxe de l'instruction `switch`.) Il est fortement conseillé, dans le cas où l'on omet délibérément une instruction `break` entre deux parties du corps d'un `switch`, d'écrire un commentaire signalant qu'il ne s'agit pas d'une erreur. (La façon d'exprimer de tels commentaires sera expliquée à la section 2.5.) Le seul cas où cela n'est pas nécessaire est celui de plusieurs étiquettes `case` qui correspondent au même point d'entrée, en d'autres termes qui sont séparées par des séquences vides.

Un organigramme précisant la sémantique de l'instruction `switch` est fourni à la figure 2.4.

Le programme de la figure 2.5 illustre l'utilisation de l'instruction `switch` pour implémenter un choix conditionnel avec un grand nombre de branches. Ce programme lit le numéro d'une carte à jouer et en affiche la description. Le cas d'un numéro compris entre 2 et 10 montre comment grouper des étiquettes `case` en les séparant par des séquences vides. Dans ce programme, la partie "`default:`" du corps de l'instruction `switch` est un exemple de *programmation défensive* : elle permet au programme de gérer correctement le cas d'un numéro de carte invalide entré par l'utilisateur.

Il est probablement utile pour aider à comprendre ce programme de donner quelques explications sur le principe de fonctionnement de la fonction `printf` (nous y reviendrons plus en détail au chapitre 4). Cette fonction reçoit un premier *argument* qui est une *chaîne de caractères* ; l'effet de `printf` est d'afficher cette chaîne en y remplaçant les occurrences de *marqueurs de format*

```c
#include <stdio.h>

int main()
{
  int carte;

  printf("Entrez un nombre entre 1 et 13: ");
  scanf("%d", &carte);

  printf("Cette carte est ");

  switch (carte)
  {
    case 1:
      printf("un as");
      break;

    case 2:
    case 3:
    case 4:
    case 5:
    case 6:
    case 7:
    case 8:
    case 9:
    case 10:
      printf("un %d", carte);
      break;

    case 11:
      printf("un valet");
      break;

    case 12:
      printf("une dame");
      break;

    case 13:
      printf("un roi");
      break;

    default:
      printf("invalide");
  }

  printf(".\n");
}
```

FIGURE 2.5 – Description d'une carte à jouer

par la valeur des arguments suivants. Par exemple, dans l'instruction

```
printf("un %d", carte)
```
,

la chaîne à afficher `"un %d"` contient un marqueur de format "%d" qui correspond à un entier signé. Ce marqueur sera donc remplacé par la valeur de l'argument suivant de l'appel, c'est-à-dire `carte`, affichée comme une valeur entière. Par exemple, si `carte` vaut 7, alors la chaîne qui sera affichée est `"un 7"`.

Signalons pour terminer que l'on aurait pu écrire un programme équivalent à celui de la figure 2.5 en n'employant uniquement que des choix binaires, c'est-à-dire en imbriquant plusieurs instructions `if` les unes dans les autres. Utiliser une instruction `switch` est cependant plus efficace, car le compilateur est capable de la traduire en des instructions machine qui s'exécutent plus rapidement.

2.4.3 Les instructions de boucle

Nous passons maintenant aux *instructions de boucle*, qui permettent de répéter plusieurs fois l'exécution d'un morceau de code. En langage C, il en existe trois.

La boucle `while`

Nous avons déjà rencontré cette instruction dans l'implémentation de l'algorithme d'Euclide à la section 1.3.3. Sa syntaxe est proche de celle de l'instruction `if`, si ce n'est que l'instruction `while` ne possède pas de partie commençant par `else`.

`while` (*expr*)
instr ;

L'expression *expr*, appelée le *gardien* de la boucle, s'évalue en une valeur booléenne. L'instruction *instr* constitue le *corps* de la boucle ; elle peut bien sûr être remplacée par un bloc.

La sémantique de l'instruction `while` est donnée par l'organigramme de la figure 2.6. L'instruction commence par évaluer le gardien *expr*. Si le résultat obtenu est vrai, alors le corps *instr* de l'instruction est exécuté. Ensuite, le même processus se répète : l'expression *expr* est évaluée, *instr* est exécutée si le résultat obtenu est vrai, et ainsi de suite. L'instruction `while` se termine dès que l'évaluation du gardien *expr* retourne un résultat faux. Si cela se produit lors de sa première évaluation, alors le corps *instr* de l'instruction n'est pas exécuté du tout. Si cela ne se produit jamais, alors *instr* est exécutée indéfiniment, et l'instruction `while` ne se termine pas. (On parle alors de *boucle infinie*.)

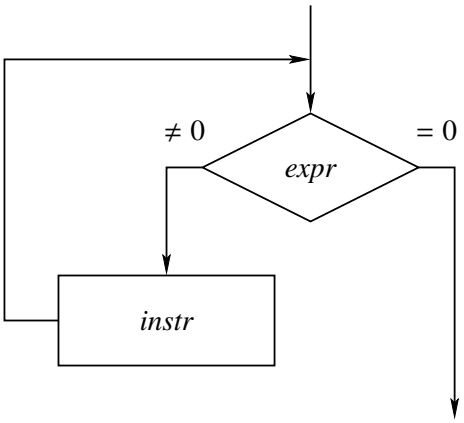

FIGURE 2.6 – Organigramme de l'instruction `while`

```
#include <stdio.h>

int main()
{
  unsigned i;

  i = 0;

  while (i < 100)
    {
      printf("%d\n", i);
      i++;
    }
}
```

FIGURE 2.7 – Compteur simple

Comme exemple de programme exploitant l'instruction `while`, nous proposons celui de la figure 2.7, qui énumère simplement tous les entiers de 0 à 99. Ce programme utilise un *compteur de boucle*[7] `i` initialisé à 0, et un gardien qui conditionne l'exécution d'une itération au fait que la valeur de ce compteur est strictement inférieure à 100. Le corps de la boucle affiche la valeur courante de `i`, et puis l'incrémente.

7. L'utilisation de variables nommées `i`, `j`, `k`, ... comme compteurs est une ancienne convention provenant du langage Fortran.

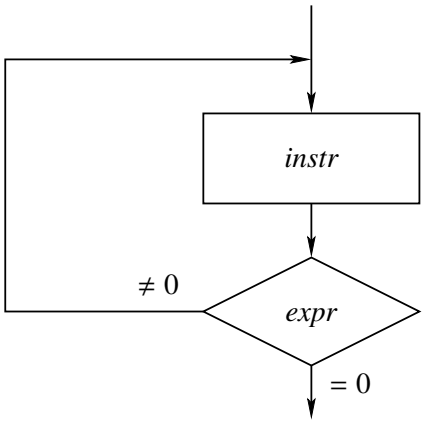

FIGURE 2.8 – Organigramme de l'instruction `do ... while`

La boucle `do ... while`

La deuxième instruction de boucle du langage C est l'instruction `do ... while`. Celle-ci partage l'usage d'un mot-clé avec l'instruction `while` que nous venons d'étudier, mais possède une syntaxe différente.

```
do
    instr;
while (expr);
```

Tout comme pour l'instruction `while`, le gardien *expr* s'évalue en une valeur booléenne, et le corps *instr* de la boucle peut être remplacé par un bloc.

La sémantique de cette opération est décrite par l'organigramme de la figure 2.8. L'instruction commence par exécuter le corps *instr* de la boucle, et puis seulement ensuite évalue *expr*. Si le résultat obtenu est vrai, alors on exécute à nouveau *instr*, on évalue *expr*, et ainsi de suite. Dès que l'évaluation de *expr* retourne une valeur fausse, l'instruction `do ... while` termine son exécution.

La différence entre les instructions `while` et `do ... while` est donc que la première évalue le gardien de boucle avant d'effectuer une itération, et la seconde après. Dans le cas particulier où le gardien est faux la première fois qu'il est évalué, l'instruction `while` n'effectue donc aucune itération, alors que l'instruction `do ... while` en effectue une.

On utilise l'instruction `while` quand on est amené à décider si une nouvelle itération est ou non nécessaire avant d'effectuer celle-ci. On emploie `do ... while` quand la décision de continuer ou non la boucle ne peut être prise après une itération, par exemple parce qu'elle

dépend de données produites par cette itération. En pratique, on constate que le premier cas est plus fréquent que le second pour les problèmes de programmation courants.

Par exemple, quand on utilise une boucle pour appliquer une certaine opération de traitement aux éléments d'un ensemble, il est nécessaire de déterminer avant d'effectuer cette opération s'il reste au moins un élément à traiter. Si l'ensemble considéré est vide, alors aucun traitement ne doit être effectué. On emploie donc dans ce cas une instruction `while`.

Pour illustrer une application de `do ... while`, examinons le problème qui consiste à tirer un nombre premier au hasard. Si l'on dispose d'un générateur de nombres aléatoires, une façon simple [8] de résoudre ce problème consiste à tirer des nombres au hasard en s'arrêtant dès que l'on obtient un nombre premier. On peut programmer cet algorithme à l'aide d'une boucle `do ... while` qui teste après chaque itération si celle-ci a produit ou non un nombre premier. Un autre exemple est donné par l'algorithme de Héron étudié à la section 1.2.2 : dans cet algorithme, on décide d'effectuer ou non une itération supplémentaire sur base d'une dernière estimation qui a été calculée par l'itération courante. Une instruction `do ... while` est donc indiquée pour implémenter cet algorithme.

La boucle `for`

La troisième instruction de boucle du langage C est l'instruction `for`, dont la syntaxe est plus compliquée :

```
for (expr1; expr2; expr3)
    instr;
```

Cette instruction comprend quatre composants. Tout comme pour les deux autres instructions de boucle, l'instruction *instr* constitue le corps de la boucle, et peut être remplacée par un bloc. Les parenthèses qui suivent le mot-clé `for` sont obligatoires, et doivent contenir exactement trois expressions [9] *expr1*, *expr2* et *expr3* séparées par des points-virgules "`;`".

La sémantique de l'instruction `for` est identique à celle de la construction suivante.

8. Il ne s'agit pas de la meilleure solution.
9. Signalons que les versions modernes du langage C autorisent aussi de remplacer *expr1* par une déclaration de variable.

```
#include <stdio.h>

int main()
{
  unsigned i;

  for (i = 0; i < 100; i++)
    printf("%d\n", i);
}
```

FIGURE 2.9 – Compteur simple (boucle `for`)

```
{
  expr1;
  while (expr2)
    {
      instr;
      expr3;
    }
}
```

Cela signifie qu'*expr1* est évaluée une seule fois, avant de commencer à itérer la boucle. On emploie habituellement cette expression pour initialiser la valeur des variables manipulées dans la boucle. L'expression *expr2* est le gardien de boucle ; elle est évaluée avant chaque itération afin de déterminer si celle-ci doit avoir lieu. Cette évaluation doit retourner une valeur booléenne. Enfin, *expr3* est évaluée à la fin de chaque itération, après avoir exécuté le corps *instr* de la boucle. On exprime donc dans *expr3* les opérations qui doivent être effectuées pour passer d'une itération de la boucle à la suivante.

Il est permis de laisser *expr1*, *expr2*, *expr3* et/ou *instr* vides. Dans le cas d'*expr2*, cela revient à exprimer un gardien qui est toujours vrai. En particulier, l'instruction

```
for (;;);
```

implémente une boucle infinie qui n'effectue aucune opération utile.

L'instruction `for` permet de réécrire de façon un peu plus simple le programme de la figure 2.7. La nouvelle version est donnée à la figure 2.9. Il est facile de voir que ces deux programmes sont équivalents.

Un exemple un peu plus intéressant de programme utilisant l'instruction `for` est fourni à la figure 2.10. Ce programme est chargé de générer des tables de multiplication, et contient deux boucles imbriquées. La boucle sur `i` énumère tous les entiers de 1 à 9. Pour chacun d'entre

```
#include <stdio.h>

int main()
{
  unsigned i, j;

  for (i = 1; i <= 9; i++)
    {
      printf("\nTable pour %d\n", i);
      printf("=============\n\n");

      for (j = 1; j <= 9; j++)
        printf("%d x %d = %d\n", j, i, i * j);
    }
}
```

FIGURE 2.10 – Générateur de tables de multiplication

eux, la boucle sur `j` multiplie successivement cet entier par toutes les valeurs de 1 à 9. Au total, l'instruction `printf(...)` la plus interne est donc exécutée 81 fois.

2.4.4 Les instructions de rupture de séquence

Nous avons vu à la section 2.4.2 qu'une instruction `break` située dans le corps d'une instruction `switch` termine immédiatement l'exécution de cette dernière. L'instruction `break` peut également être employée dans le corps d'une instruction `while`, `do ... while` ou `for`, avec le même effet : son exécution sort de la boucle en terminant immédiatement l'itération en cours.

L'instruction `break` ne peut être employée que dans le corps d'une instruction `switch`, `while`, `do ... while` ou `for`. Dans le cas de plusieurs instructions imbriquées, l'instruction `break` se rapporte uniquement à la plus interne d'entre elles.

Examinons par exemple le fragment de code suivant.

```
for (i = 0;; i++)
  if (i > 100)
    break;
```

On a ici une boucle dont le gardien est toujours vrai. Si son corps était vide, elle effectuerait donc indéfiniment des itérations, en incrémentant le compteur de boucle `i` à chaque étape. Cependant, le corps de cette boucle contient une instruction de choix conditionnel qui provoque

la terminaison de la boucle dès que la condition `i > 100` devient satisfaite. Au total, ce code exécutera donc 102 itérations de la boucle, pour les valeurs de `i` allant de 0 à 101.

Il existe une autre instruction de rupture de séquence appelée `continue`. Cette instruction ne peut être utilisée que dans le corps d'une instruction de boucle `while`, `do ... while` ou `for` (à la différence de `break`, il n'est pas permis de l'employer dans le corps d'une instruction `switch`). L'effet de `continue` est de terminer l'itération courante de la boucle et de passer immédiatement à l'itération suivante. Dans le cas d'une boucle

```
for (expr1; expr2; expr3)
    instr;
```

cela signifie que l'exécution de `continue` au sein de *instr* provoque la terminaison immédiate de *instr*, suivie par une évaluation de *expr3*, et ensuite d'une évaluation du gardien *expr2* afin de déterminer s'il faut à nouveau effectuer une itération.

Tout comme pour `break`, une instruction `continue` placée dans plusieurs boucles imbriquées se rapporte uniquement à celle qui est la plus interne.

Il ne faut pas abuser des instructions `break` et `continue`, mais leur utilisation permet dans certains cas de simplifier la structure d'un programme et donc de le rendre plus lisible. Par exemple, le morceau de code suivant montre comment programmer une opération qui doit être effectuée pour toutes les paires (i, j) telles que $0 \leq i < 100$, $1 \leq j < i$ et i ne divise pas j.

```
for (i = 0; i < 100; i++)
  for (j = 1; j < i; j++)
    {
      if (!(i % j))
        continue;
       ⋮
    }
```

On aurait bien sûr pu ajouter une branche `else` à l'instruction `if`, contenant l'opération à effectuer, mais l'utilisation de `continue` permet ici d'éviter un niveau supplémentaire d'indentation, ce qui améliore la lisibilité du code.

2.5 Les commentaires

Les programmes que l'on rédige ne sont pas uniquement destinés à être traités par un compilateur ; il est également important qu'ils soient lisibles et facilement compréhensibles pour

les programmeurs amenés les mettre au point ou à y apporter des modifications. Une première étape dans ce sens consiste bien sûr à choisir des identificateurs, tant pour les variables que pour d'autres éléments de programmes que nous étudierons plus tard, qui possèdent un nom clair, cohérent, et décrivant précisément l'usage qui en est fait.

En addition à cela, on peut également insérer des *commentaires* dans les programmes, qui sont des fragments de texte ignorés par le compilateur. Nous avons notamment déjà évoqué à la section 2.4.2 la pertinence d'écrire un tel commentaire lorsque l'on omet une instruction `break` entre deux parties du corps d'une instruction `switch`.

En langage C, les commentaires sont délimités par "`/*`" et "`*/`". Le texte situé entre ces marqueurs n'est pas lu par le compilateur. Ces commentaires ne peuvent pas être imbriqués. Il existe également une autre forme de commentaire admise par les versions modernes du langage, inspirée du langage de programmation C++ : un commentaire peut aussi commencer par "`//`", et se terminer à la fin de la ligne courante. L'exemple suivant montre un morceau de code contenant les deux formes de commentaires.

```
/* Position de l'objet */

double x, y;  // Coordonnées
double h;     // Hauteur
```

2.6 Exercices

1. En supposant que les variables `x`, `y` et `z` sont entières et initialisées à zéro, quelle sera leur valeur après l'exécution de chacune des instructions C suivantes ?

   (a) `z = x++ + --y;`

   (b) `z -= x++ && y++ ? 1 : 2;`

   (c) `z += ++x && ++y ? 1 : 2;`

   (d) `z = x > 0 || !++y ? x++ : ++x;`

   (e) `z = (x++, x > 0) ? !x : 1 - !x;`

2. Écrire des fragments de code C équivalents à ceux donnés ci-dessous, mais utilisant l'instruction de contrôle `while`.

   (a)
   ```
   for (i = 0; i < 1000; i += 10)
     j += i;
   ```

(b)
```
i = j = 0;
do
  {
    j += i;
    i += 2;
  }
while (j < k);
```

(c)
```
for (i = 0, j = 0; i < 1000; i++)
  {
    if (f(i) > f(j))
       continue;
    j += i;
  }
```

(d)
```
for (;;);
```

3. Écrire des fragments de code C équivalents à ceux donnés ci-dessous, mais utilisant l'instruction de contrôle `for`.

(a)
```
i = 2;
while (i < 100000)
  {
    j += i;
    i *= 2;
  }
```

(b)
```
p = premier();
while (!est_dernier(p))
  {
    traiter(p);
    p = suivant(p);
  }
```

(c)
```
while (attendre());
```

(d)
```
i = 0;
j = 0;
while (1)
  {
    j += i++;
    if (j >= k)
      break;
  }
```

4. Écrire des programmes C implémentant les algorithmes obtenus pour les exercices de la section 1.5

   Note : Nous donnons ici quelques éléments relatifs à la manipulation des nombres réels, qui fait l'objet du premier exercice. Pour les fonctions `printf` et `scanf`, le marqueur de format correspondant au type `double` est "`%lf`". Pour calculer la racine carrée et la valeur absolue d'un nombre, on peut employer (réciproquement) les fonctions `sqrt` et `fabs` de la bibliothèque standard. Pour cela, la directive `#include <math.h>` doit figurer au début du programme. Enfin, il faut explicitement indiquer au compilateur que la bibliothèque de fonctions mathématiques doit être liée au programme ; pour le compilateur `GCC`, cela se fait en ajoutant l'option "`-lm`" en fin de la ligne de commande.

5. Les *nombres de Fibonacci* $F_1, F_2, F_3, \ldots$ sont définis par les règles suivantes :

$$\begin{aligned} F_1 &= 1 \\ F_2 &= 1 \\ F_n &= F_{n-1} + F_{n-2} \text{ pour tout } n > 2. \end{aligned}$$

   Écrire un programme C capable de calculer efficacement la valeur de F_n pour un indice n donné.

6. Une *suite de Collatz* s'obtient en calculant, pour un nombre entier $z_0 > 1$ donné, les nombres $z_1, z_2, z_3, \ldots$ de la façon suivante. Pour tout $i \geq 0$:

$$\begin{cases} \text{Si } z_i \textit{ est pair}, \text{ alors } z_{i+1} = \dfrac{z_i}{2}. \\ \text{Si } z_i \textit{ est impair}, \text{ alors } z_{i+1} = 3z_i + 1. \end{cases}$$

   Écrire un programme C capable de calculer, pour un nombre z_0 entré au clavier, le nombre d'étapes nécessaires pour atteindre le nombre 1 (c'est-à-dire la plus petite valeur de i pour laquelle on a $z_i = 1$). Par exemple, pour $z_0 = 3$, le programme doit afficher 7, puisque l'on a $z_1 = 10$, $z_2 = 5$, $z_3 = 16$, $z_4 = 8$, $z_5 = 4$, $z_6 = 2$ et $z_7 = 1$.

   Note : La question de savoir si la valeur 1 est toujours atteinte à partir d'une valeur quelconque de z_0 est un problème ouvert en mathématiques.

7. Écrire un programme C permettant de calculer efficacement la valeur d'un *coefficient binomial*

$$C_n^m = \frac{n!}{m!(n-m)!},$$

où n et m sont des entiers donnés tels que $0 \leq m \leq n$, et où $p! = p.(p-1)\ldots 2.1$.

8. Écrire un programme C capable de factoriser un nombre entier entré au clavier. Le résultat doit être affiché sous la forme

facteur_premier^puissance facteur_premier^puissance ...

où les facteurs premiers apparaissent dans l'ordre croissant.
Par exemple, pour le nombre 147015, le programme doit afficher "`3^5 5 11^2`".