Algorithmique 1 : méthodologie de la programmation impérative
Support de cours et d'exercices

Avertissements : ne figurent sur ce support que des jalons de l'enseignement dispensé en cours ainsi que des énoncés d'exercices ; pour le langage C en particulier, il est nécessaire de se reporter à la norme C23 et aux ouvrages ; ce support est accompagné d'un support complémentaire de travaux pratiques, subdivisé en fiches, et d'une archive de sources C.

Buts de l'enseignement

La matière *Algorithmique 1* propose :

— des compléments aux bases du langage C introduites par la matière *Informatique : Bases de la programmation impérative*, dont la construction structure, la gestion standard des flots, l'arithmétique des pointeurs, le type pointeur générique et la compilation séparée ;

— l'introduction et l'utilisation systématique de la logique de Hoare, laquelle est une démarche qui permet, *simultanément*, de construire un programme solution d'un problème posé et d'apporter la preuve que ce programme est bien solution du problème ;

— une introduction à l'analyse des algorithmes, cadre général qui permet d'évaluer l'efficacité des solutions et de les comparer ;

— un exposé de quelques-uns des problèmes à la fois classiques et fondamentaux en informatique avec certaines de leurs solutions en programmation impérative accompagnées de preuves et d'analyses.

Références principales

Les ouvrages figurant dans la liste ci-dessous sont disponibles à la bibliothèque universitaire du site du Madrillet.

Aitken et Jones, *Le langage C*, Pearson, 2012.

Backhouse, *Construction et vérification de programmes*, Masson, 1989.

Bouvier, George et Le Lionnais, *Dictionnaire des Mathématiques*, Presses Universitaires de France, 2009.

Braquelaire, *Méthodologie de la programmation en C. Norme C 99, API POSIX*, Dunod, 2005.

Cormen, Leiserson, Rivest et Stein, *Algorithmique*, Dunod, 2010.

Dijkstra, *A Short Introduction to the Art of Programming*, `http://www.cs.utexas.edu/users/EWD/`, note 316 (1971).

Dijkstra, *Programming methodologies, their objectives and their nature*, `http://www.cs.utexas.edu/users/EWD/`, note 469 (année non déterminée).

Graham, Knuth et Patashnik, *Mathématiques concrètes. Fondations pour l'informatique*, International Thomson Publishing France, 1998.

Ifrah, *Histoire universelle des chiffres*, Robert Laffont, 1994.

JTC1/SC22/WG14-C, *(International standardization working group for the programming language C), ISO/IEC 9899:2023* (E), `https://www.open-std.org/jtc1/sc22/wg14/www/docs/n3096.pdf`, pour la norme C23.

Kernighan et Pike, *La programmation en pratique*, Vuibert, 2001.

Kernighan et Ritchie, *Le langage C. Norme ANSI*, Dunod, 2004.

Lazard, *Pratique performante du langage C. Cours, techniques et exercices corrigés*, Ellipse, 2013.

Meyer et Baudoin, *Méthodes de programmation*, Eyrolles, 1984.

Wirth, *Introduction à la programmation systématique*, Masson, 1983.

Plan général

Partie I. Boite à outils. 1. Prérequis. 2. Construction de programmes corrects. 3. C : compléments aux bases. 4. C : types agrégés et unions. 5. C : flots. 6. C : développements. 7. Analyse des algorithmes.

Partie II. Sélection d'algorithmes fondamentaux. 8. Algorithmes numériques. 9. Algorithmes de recherche. 10. Algorithmes de tri.

Exercices proposés

Chaque exercice appartient à l'une des quatre catégories suivantes :

— échauffement. Il s'agit soit de questions de cours, soit d'un exercice proposant une ou des applications immédiates du cours. Tout un chacun se doit de les travailler sans attendre la séance de travaux dirigés. Le numéro de ces exercices est précédé par la marque ▶ **É** ;

— de base, marque ▶ **B**. Exercice permettant d'appliquer et de mettre en œuvre les notions et résultats exposés en cours ;

— modérément difficile, marque ▶ **MD**. Exercice d'approfondissement, exigeant la maitrise de ces mêmes notions et résultats ;

— difficile, marque ▶ **D**. Exercice requérant tout à la fois la maitrise complète des notions et résultats, un certain recul, un intérêt pour la théorie et... du temps.

Seule une partie des exercices proposés pourra être traitée dans la durée impartie aux cours et aux séances de travaux dirigés.

Contrainte

Tout code C doit être compilable par `gcc` branche 13.3 (minimum) avec les options `-std=c2x`, `-Wall`, `-Wconversion`, `-Werror`, `-Wextra`, `-Wpedantic`, `-Wwrite-strings` et `-O2`.

MÀJ 18-01 ⊖`-Wstack-protector`⊖

Partie I. Boite à outils

1 Prérequis en mathématiques et en programmation

1.1 Notations mathématiques

Algèbre

$\sum_{k=m}^{n} a_k$; $\sum_{k=m}^{n} a_k$	somme $a_m + a_{m+1} + \cdots + a_n$ avec m et n naturels ; lorsque $m > n$, la somme est « vide » et vaut 0
$\prod_{k=m}^{n} a_k$; $\prod_{k=m}^{n} a_k$	produit $a_m \cdot a_{m+1} \cdot \ldots \cdot a_n$ avec m et n naturels ; lorsque $m > n$, le produit est « vide » et vaut 1

Analyse élémentaire

$\langle a_k \rangle$	suite de terme général a_k, d'index $\mathbb{N}$
$\langle a_k \rangle_{P(k)}$	suite extraite de $\langle a_k \rangle$ d'index $\{k \in \mathbb{N} \mid P(k)\}$
$\sum_{P(x)} f(x)$; $\sum_{P(x)} f(x)$	somme des images par f de tous les x satisfaisant la propriété P ; lorsque l'ensemble décrit par P est vide, la somme est « vide » et vaut 0
$\prod_{P(x)} f(x)$; $\prod_{P(x)} f(x)$	produit des images par f de tous les x satisfaisant la propriété P ; lorsque l'ensemble décrit par P est vide, le produit est « vide » et vaut 1

Arithmétique

$\simeq$	sensiblement égal à
$<$	strictement inférieur à
$\leqslant$	inférieur (ou égal) à
$>$	strictement supérieur à
$\geqslant$	supérieur (ou égal) à
$\lfloor x \rfloor$	partie entière inférieure du réel x : plus grand entier $\leqslant x$
$\lceil x \rceil$	partie entière supérieure du réel x : plus petit entier $\geqslant x$
$a \bmod b$	reste de la division de l'entier a par l'entier non nul b : $a - b\lfloor a/b \rfloor$
F_n	nombre de Fibonacci de rang n, défini par la récurrence : $F_0 = 0$; $F_1 = 1$; $F_n = F_{n-1} + F_{n-2}$ pour $n \geqslant 2$
H_n	nombre harmonique de rang n : $H_n = \sum_{k=1}^{n} 1/k$ avec n naturel
$n!$	factorielle n : $n! = \prod_{k=1}^{n} k$ avec n naturel
$[P]$	fonction « zéro-un » : $[P]$ vaut 1 si la propriété P est vraie et 0 sinon

Ensembles

$\emptyset$	ensemble vide
$\mathbb{N}$	ensemble des (entiers) naturels
$\mathbb{Z}$	ensemble des entiers (algébriques, rationnels ou relatifs)
$\max E$	maximum de l'ensemble $E \neq \emptyset$ relativement à l'ordre sur E
$\min E$	minimum de l'ensemble $E \neq \emptyset$ relativement à l'ordre sur E

Fonctions usuelles

ln	fonction logarithme népérien (ou naturel), de base e
$\log_b$	fonction logarithme de base b : $\log_b(x) = \ln x / \ln b$
lb	fonction logarithme binaire, de base 2 : $\mathrm{lb}\, x = \log_2(x)$

Logique

$\neg$	négation, « non »
$\wedge$	conjonction, « et »
$\vee$	disjonction, « ou »
$\Rightarrow$	inférence, « et donc »

1.2 Matériels mathématiques

Approximations

$e \simeq 2{,}718$; $\quad \ln 2 \simeq 0{,}693$; $\quad 2^{10} \simeq 10^3$.

Bases de combinatoire des mots

Soit A un ensemble.

Un mot (fini) sur A est une suite finie $u = \langle a_0, a_1, \ldots, a_{n-1} \rangle$ où $n \geqslant 0$ et $a_k \in A$ pour $0 \leqslant k \leqslant n-1$. L'entier n est la longueur de u, notée $|u|$. Lorsque $n = 0$, u est le mot vide, noté $\langle\rangle$ ou ε (« epsilon »).

La notation A^* désigne l'ensemble des mots sur A. Un élément de A est une lettre et A est l'alphabet.

Si $u = \langle a_0, a_1, \ldots, a_{m-1} \rangle$ et $v = \langle b_0, b_1, \ldots, b_{n-1} \rangle$ sont deux mots de A^*, leur concaténation est le mot de A^* noté $u \cdot v$ et défini par $u \cdot v = \langle a_0, a_1, \ldots, a_{m-1}, b_0, b_1, \ldots, b_{n-1} \rangle$.

Soit u un mot de A^*. Le mot v est un facteur (ou segment) de u s'il existe deux mots u' et u'' de A^* tels que $u = u' \cdot v \cdot u''$. Si $u' = \varepsilon$, v est un préfixe de u ; si $u'' = \varepsilon$, v en est un suffixe.

Un sous-mot d'un mot de A^* est une suite extraite de ce mot.

Un sous-mot d'un mot, et donc aussi un facteur, un préfixe ou un suffixe de ce mot, est qualifié de propre lorsqu'il ne s'agit pas du mot lui-même.

Pour tout mot u de A^* sont notés :

— $u[k]$, la lettre de u située à l'indice k de u, avec $0 \leqslant k \leqslant |u| - 1$;

— $u[j \dots k]$, le facteur de u débutant à l'indice j et se terminant à l'indice k, avec $0 \leqslant j \leqslant |u|$ et $-1 \leqslant k \leqslant |u| - 1$: $u[j \dots k] = \langle u[j] \rangle \cdot \langle u[j+1] \rangle \cdot \ldots \cdot \langle u[k] \rangle$.

La notation des facteurs peut être étendue avec une utilisation des crochets comme pour les intervalles réels. Le préfixe de longueur k du mot u peut ainsi être noté $u[0 \dots k[$.

Pour tout mot u de A^* et tout naturel k, la puissance k-ième de u, notée u^k, est le mot défini par la récurrence $u^k = \varepsilon$ si $k = 0$ et $u^k = u \cdot u^{k-1}$ sinon.

Lorsque le contexte le permet, les chevrons et les virgules utilisés dans les notations des mots comme délimiteurs et séparateurs ainsi que l'opérateur de concaténation sont omis.

Parties entières aux égalités

Pour tout réel x :

$\lfloor x \rfloor = x \Leftrightarrow x$ est un entier $\Leftrightarrow \lceil x \rceil = x$;
$\lceil x \rceil - \lfloor x \rfloor = [x$ n'est pas un entier$]$;

$\lfloor -x \rfloor = -\lceil x \rceil$;
$\lceil -x \rceil = -\lfloor x \rfloor$.

Pour tout réel x, pour tout entier n :

$n \leqslant x < n+1 \Leftrightarrow \lfloor x \rfloor = n \Leftrightarrow x - 1 < n \leqslant x$;
$n - 1 < x \leqslant n \Leftrightarrow \lceil x \rceil = n \Leftrightarrow x \leqslant n < x + 1$;

$\lfloor x + n \rfloor = \lfloor x \rfloor + n$;
$\lceil x + n \rceil = \lceil x \rceil + n$.

Pour tout entier n :

$\lfloor n/2 \rfloor = \lceil (n-1)/2 \rceil$;
$\lceil n/2 \rceil = \lfloor (n+1)/2 \rfloor$;

$\lfloor n/2 \rfloor + \lceil n/2 \rceil = n$.

Parties entières aux inégalités

Pour tous réels x, y :

$x - 1 < \lfloor x \rfloor \leqslant x \leqslant \lceil x \rceil < x + 1$;

$\lfloor x \rfloor + \lfloor y \rfloor \leqslant \lfloor x + y \rfloor \leqslant \lfloor x \rfloor + \lfloor y \rfloor + 1$;
$\lceil x \rceil + \lceil y \rceil - 1 \leqslant \lceil x + y \rceil \leqslant \lceil x \rceil + \lceil y \rceil$.

Pour tous réels x, y, pour tout entier n :

$\lfloor x \rfloor < n \Leftrightarrow x < n$;
$n < \lceil x \rceil \Leftrightarrow n < x$;
$\lceil x \rceil \leqslant n \Leftrightarrow x \leqslant n$;
$n \leqslant \lfloor x \rfloor \Leftrightarrow n \leqslant x$;

$x < n \leqslant y \Leftrightarrow \lfloor x \rfloor < n \leqslant \lfloor y \rfloor$;
$x \leqslant n < y \Leftrightarrow \lceil x \rceil \leqslant n < \lceil y \rceil$.

Pour tous entiers p, q :

$p \leqslant q \Rightarrow p \leqslant \lfloor (p+q)/2 \rfloor \leqslant \lceil (p+q)/2 \rceil \leqslant q$;
$p < q \Rightarrow p < \lceil (p+q)/2 \rceil \wedge \lfloor (p+q)/2 \rfloor < q$.

Sommations aux égalités

Pour tout naturel n :

$$\sum_{k=0}^{n} k = \frac{n(n+1)}{2} ;$$

$$\sum_{k=0}^{n} k^2 = \frac{n(n+\frac{1}{2})(n+1)}{3} ;$$

Pour tout naturel n, pour tout réel non nul x :

$$\sum_{k=0}^{n} x^k = \begin{cases} n+1, & \text{si } x = 1 ; \\ \dfrac{x^{n+1}-1}{x-1}, & \text{sinon.} \end{cases}$$

Sommations aux inégalités

Soit f une fonction réelle continue sur l'intervalle $[p\,;\,q]$ où p et q sont des entiers. Alors :

$$\sum_{k=p}^{q-1} f(k) \leqslant \int_p^q f(x)\,dx \leqslant \sum_{k=p+1}^{q} f(k), \quad \text{si } f \text{ est croissante ;}$$

$$\sum_{k=p+1}^{q} f(k) \leqslant \int_p^q f(x)\,dx \leqslant \sum_{k=p}^{q-1} f(k), \quad \text{si } f \text{ est décroissante.}$$

Il s'ensuit par exemple que, pour tout naturel non nul n :

$\ln n \leqslant H_n \leqslant \ln n + 1$;
$n \ln n - n + 1 \leqslant \ln n! \leqslant n \ln n$.

Suites entières strictement monotones

Toute suite entière strictement monotone bornée est finie. Si elle n'est pas vide, sa longueur est majorée par la valeur absolue de la différence de ses bornes plus une unité.

1.3 Prérequis en programmation

Notion de type

Un type (de données) est un ensemble fini de valeurs muni d'opérateurs applicables à ces valeurs. Ce qui s'écrit volontiers sous la forme :

type = ensemble fini de valeurs + opérateurs.

Types

— Types de base : caractère « de base », entier, booléen, flottant.
— Types dérivés : tableau, chaine de caractères, pointeur.

Notion de variable

Une variable est une zone (ou case) mémoire stockant une donnée pouvant varier au cours de l'exécution d'un programme.

Elle possède un identificateur (ou nom), un type, une adresse mémoire et une valeur :

variable = identificateur + type + adresse + valeur.

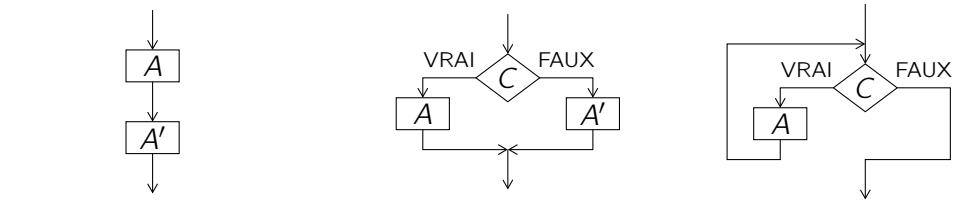
Instructions élémentaires

— Affectation à une variable de la valeur d'une expression.
— Écriture avec transcodage sur la sortie standard de la valeur d'une expression.
— Affectation à une variable d'une valeur lue sur l'entrée standard avec transcodage.

Instructions composées

Voici, présentées sous forme d'organigramme, les trois instructions composées fondamentales, où A et A' sont des instructions, C, une condition :

composition séquentielle | *composition alternative « si sinon »* | *composition itérative « (boucle) tant que »*



Notion de programme

Un programme source est un énoncé, exprimé dans un langage de programmation, d'un procédé de calcul, solution d'un problème. Un programme exécutable est la traduction d'un programme source en langage machine.

Un programme reçoit des données, transmet un résultat et modifie des données-résultat.

Notion de sous-programme

Un sous-programme est une séquence d'instructions, interne à un programme, nommée et éventuellement paramétrée. Recourir aux sous-programmes permet d'éviter la duplication et d'améliorer la lisibilité des programmes.

Deux catégories de sous-programmes peuvent être distinguées : tandis qu'une fonction renvoie un résultat, une procédure n'en renvoie pas. Autrement dit l'appel d'une fonction est assimilé à une valeur, celui d'une procédure, à une instruction.

Exercices

▶ É 1.1

Apportez la preuve des propriétés énoncées dans la section 1.2 : parties entières, sommations, suites entières strictement monotones.

▶ É 1.2

Sans calculatrice : récitez les puissances de 2 jusqu'à 2^{16} ; donnez des approximations satisfaisantes de 2^{32} et 2^{64} ; donnez la partie entière supérieure du logarithme binaire de 8 milliards.

▶ É 1.3

Soient E et F deux types, f une fonction de $E \times E$ dans F. On propose un nouveau calcul, g, des images de f. Il s'agit donc de montrer que g produit le même résultat que f. Pour ce faire, on choisit de tester tous les couples (x, y) de $E \times E$.

Supposez que vous disposiez d'un processeur capable d'effectuer chaque test élémentaire à une cadence de 4 GHz. Combien de temps faut-il pour effectuer le test complet si les éléments de E ont une représentation qui utilise à plein un codage sur 16 bits ? sur 32 bits ? sur 64 bits ? Concluez.

▶ É 1.4

Le triplet $(E, \top, e)$ est un *monoïde* lorsque $\top$ est une loi interne à E, associative et unifère, d'élément neutre e. Si $(E, \top, e)$ et $(F, \bot, f)$ sont deux monoïdes, l'application $\varphi : E \to F$ est un *morphisme de monoïdes* lorsque $\varphi(e) = f$ et $\varphi(x \top y) = \varphi(x) \bot \varphi(y)$ pour tous $x, y \in E$ (les symboles $\top$ et $\bot$ se disent « truc » et « anti-truc »).

Montrez que, pour tout alphabet A, la longueur est un morphisme du monoïde $(A^*, \cdot, \varepsilon)$ vers le monoïde $(\mathbb{N}, +, 0)$.

2 Construction de programmes corrects

2.1 Cadre général

Algorithmes, programmes, algorithmique

Définitions : algorithme et programme

— Un *algorithme* est la description d'un procédé de calcul qui répond à un problème posé. Il se présente sous la forme d'une suite finie d'instructions organisées selon des règles bien précises, à effectuer sur un nombre fini d'objets. La solution est produite en un temps fini.

— Un *programme* est une traduction, ou, plus spécifiquement, une implantation, d'un algorithme dans un langage de programmation.

L'un des avantages du passage par un énoncé algorithmique est la possibilité de ne pas se focaliser sur des détails techniques inhérents aux langages de programmation. Pour parvenir à cela, le style de l'énoncé doit être à la fois :

— précis, ne laissant place à aucune ambigüité pour des traductions aisées ;

— souple, car prétendant à une universalité ;

— concis, afin d'être mémorisable.

Définition : algorithmique

L'*algorithmique* est la science de la description des algorithmes connus, de la production de nouveaux algorithmes, ainsi que de leur analyse.

Modèles

Le problème contient dans son énoncé même le *modèle (explicite)*, lequel précise le cadre de la formulation de la solution.

Le *modèle implicite* retenu ici est celui des langages de programmation appartenant au paradigme de la *programmation impérative* :

— tout programme décrit une suite d'instructions faisant passer ses variables d'un état initial, condition sur les données, à un état final, condition sur le résultat ;

— l'instruction élémentaire la plus emblématique en est l'opération d'affectation. Notée sous la forme « $e_1 \leftarrow e_2$ », elle signifie que la valeur de l'expression de droite, e_2, est rangée dans l'objet désigné par l'expression de gauche, e_1 ;

— les règles de composition des instructions peuvent être réduites aux seules trois compositions séquentielle, « A ; A' », alternative, « **si** C **alors** A **sinon** A' **fin si** », et itérative, « **tant que** C **faire** A **fin tant que** ».

Correction

Définitions : correction, correction partielle et terminaison

Établir la *correction* d'un algorithme ou d'un programme eu égard à un problème posé, c'est apporter la preuve qu'il est bien solution de ce problème.

Cette preuve s'établit en deux temps :

— preuve de sa *correction partielle*, où l'on montre que, sous réserve que le calcul se termine, le résultat produit par le calcul est bien le résultat attendu, pour toutes les données possibles ;

— preuve de sa *terminaison*, où l'on montre que le calcul se termine, pour toutes les données possibles.

Quand bien même le langage algorithmique n'est pas complètement formalisé, la proximité entre les énoncés algorithmiques et leurs traductions autorise à établir l'*idée* de la preuve d'un énoncé algorithmique, laquelle *passe* ensuite aux programmes.

2.2 Preuve de programmes : correction partielle

Logique de Hoare

L'outil utilisé ici pour établir la correction partielle d'un programme ou d'un algorithme est la *logique de Hoare*[1]. Cette théorie repose sur quatre notions, un axiome et trois théorèmes.

Définition : assertion

Une *assertion* est l'affirmation d'une relation entre les valeurs de variables d'un programme.

Définitions : schéma pré-post, pré-assertion, post-assertion

— Le triplet (P, A, P') forme un *schéma pré-post* lorsque l'exécution de l'instruction A, commencée dans un état des variables satisfaisant l'assertion P, donne, *si elle se termine*, un état des variables qui satisfait l'assertion P'.

— Du schéma pré-post (P, A, P'), on dit de l'hypothèse P qu'elle en est la *pré-assertion*, de la conséquence P', la *post-assertion*.

Axiome : axiome de base de la logique de Hoare

Aucune évaluation ne modifie l'état des variables.

Théorème : règle d'inférence pour la composition séquentielle

Soient P, P' et P'' des assertions, A et A' des instructions. Alors :

$(P, A, P') \wedge (P', A', P'') \Rightarrow (P, (A\,;\, A'), P'')$.

Théorème : règle d'inférence pour la composition alternative

Soient P et P' des assertions, C une expression booléenne, A et A' des instructions. Alors :

$(P \wedge C, A, P') \wedge (P \wedge \neg C, A', P') \Rightarrow (P,$ **si** C **alors** A **sinon** A' **fin si**, $P')$.

Théorème : règle d'inférence pour la composition itérative

Soient P une assertion, C une expression booléenne, A une instruction. Alors :

$(P \wedge C, A, P) \Rightarrow (P,$ **tant que** C **faire** A **fin tant que**, $P \wedge \neg C)$.

Rôle des assertions

Les assertions ont pour rôle de décrire l'état des variables tout le long du chemin qui mène de l'état initial à l'état final.

Définitions : assertions d'entrée et de sortie

— L'*assertion d'entrée* est l'assertion décrivant l'état initial des variables du programme, c'est-à-dire les contraintes sur les données.

— L'*assertion de sortie* est l'assertion décrivant l'état final, c'est-à-dire les contraintes sur le résultat en fonction des données.

Définition : spécification

Spécifier un programme, ou encore, donner sa *spécification*, c'est dire ce qu'il réalise, mais pas la manière dont il le réalise.

Autrement dit, si le triplet

(assertion d'entrée, corps du programme, assertion de sortie)

est le schéma pré-post d'un programme, le couple

(assertion d'entrée, assertion de sortie)

en constitue la spécification.

1. Charles Antony Richard (Tony ou C.A.R.) Hoare est un informaticien britannique.

Invariants

Définition : invariant

Soit A une instruction. L'assertion P est un *invariant* pour A lorsque le triplet (P, A, P) forme un schéma pré-post.

Définition : invariant de boucle

Soit B une boucle **tant que** de condition C et de corps A. L'assertion P est un *invariant de boucle* pour B lorsque le triplet $(P \wedge C, A, P)$ forme un schéma pré-post.

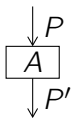
Corolaire : invariant de boucle

Soient B une boucle **tant que** de condition C et de corps A, P un invariant de boucle pour B. Alors l'assertion P est un invariant pour B. De plus, si l'assertion P est satisfaite *avant* B, elle l'est encore *avant* et *après* chacune des évaluations de C et chacune des itérations de A.

La notion d'invariant de boucle joue un rôle clé dans la construction de programmes par des méthodes systématiques : trouver un « bon » invariant de boucle P associé à une « bonne » condition d'itération C est une étape décisive dans la résolution d'un problème.

Notations

Tout schéma pré-post (P, A, P') peut également être signifié par les trois notations suivantes :

notation logique	*notation énoncé*	*organigramme*
$P \overset{A}{\Longrightarrow} P'$	*Assertion : P* *A* *Assertion : P'*	

2.3 Preuve de programmes : terminaison

Définition : quantité de contrôle

Une *quantité de contrôle* associée à une boucle donnée est une expression entière, dépendant des valeurs des variables du programme, dont la suite des valeurs considérées lors de l'évaluation de la condition de la boucle est strictement monotone et bornée.

Corolaire : invariant de boucle et quantité de contrôle

Soit B une boucle **tant que** de condition C et de corps A. Soit P un invariant de boucle pour B. Alors, si l'assertion P est satisfaite avant la boucle, que l'instruction A se termine sous l'hypothèse $P \wedge C$ et qu'une quantité de contrôle est associée à B, l'exécution de B se termine.

Les quantités de contrôle sont des outils de choix pour établir la terminaison des programmes : montrer la terminaison d'un programme revient en effet à montrer que l'exécution de chacune de ses éventuelles boucles s'achève [2].

2.4 Construction typique de programmes corrects

Nombre de problèmes admettent comme solution une instruction composée d'une phase d'initialisation, d'une phase d'itérations et d'une phase de finalisation (très souvent vide) :

A
tant que C **faire**
    A'
A''

2. La programmation récursive — un programme qui, dans sa propre définition, fait appel à lui-même, directement ou indirectement, est dit récursif ; voir *Algorithmique 2* [3] — est ici proscrite.

3. La matière *Algorithmique 2* est dispensée en deuxième année de la licence Informatique.

Pour démontrer la validité d'un tel énoncé eu égard à la spécification, deux méthodes sont à priori envisageables : démontrer la validité à postériori ; obtenir cette expression à partir du « langage des assertions ». L'expérience montre que la première méthode est (en général) vaine tandis que la seconde est particulièrement fructueuse. C'est la seconde qui est exposée ci-dessous et qui est systématiquement employée par la suite.

Si l'on veut construire un programme « typique » de spécification (P, R) et établir sa correction simultanément, il faut effectuer un raisonnement par récurrence :

— *énoncé du futur invariant de boucle* : on suppose d'abord le problème partiellement résolu. Ce que l'on exprime à l'aide d'une assertion Q ;

— *sortie de la future boucle* : on met ensuite en évidence une condition C et une instruction A'' telles que $(Q \wedge \neg C, A'', R)$ forme un schéma pré-post et l'exécution de A'' se termine sous l'hypothèse $Q \wedge \neg C$;

— *induction (ou hérédité)* : puis on met en évidence une instruction A' et une quantité entière q bornée telles que $(Q \wedge C, A', Q)$ forme un schéma pré-post et, sous l'hypothèse $Q \wedge C$, l'exécution de A' se termine et fait soit strictement croitre soit strictement décroitre la quantité q ;

— *base (ou initialisation)* : on met enfin en évidence une instruction A telle que (P, A, Q) forme un schéma pré-post et l'exécution de A se termine sous l'hypothèse P.

On obtient ainsi un algorithme qui s'énonce sous la forme

Assertion d'entrée : P
A
Invariant de boucle : Q
Quantité de contrôle : q
tant que *C* **faire**
    A′
A″
Assertion de sortie : R

et qui est correct par construction :

— sa correction partielle s'obtient par application du corolaire « invariant de boucle » puis de la règle d'inférence pour la composition séquentielle ;

— son exécution se termine car les exécutions des instructions A, **tant que** C **faire** A' **fin tant que** et A'' se terminent.

2.5 Énoncé des algorithmes et des programmes

Dans le modèle d'expression des algorithmes retenu ici, chaque donnée, chaque résultat et chaque donnée-résultat est une variable. La valeur de chaque donnée et donnée-résultat est initialisée par l'environnement, celle de chaque résultat et donnée-résultat est récupérée par l'environnement. Toutefois, afin de simplifier le discours en n'introduisant pas de notations supplémentaires signifiant la valeur initiale de chacune des données, les données sont considérées comme des variables non modifiables.

L'en-tête des algorithmes comporte au moins l'une des trois rubriques *reçoit*, *transmet* et *modifie*, et, éventuellement, la rubrique *nécessite* de manière à, au minimum, nommer les données, les variables-résultat, les données-résultat et les variables auxiliaires. La convention, contestable, de ne pas déclarer les variables auxiliaires est typique en algorithmique.

Définition : énoncé détaillé

L'énoncé d'un algorithme dans lequel figurent la spécification, exprimée en renseignant les deux rubriques *assertion d'entrée* et *assertion de sortie*, les invariants de boucle et les quantités de contrôle est dit *détaillé* : il comporte tous les éléments de la preuve de sa correction.

2.6 Exemple : échange de valeurs

Problème

Décrire une méthode d'échange du contenu de deux variables x et y de même type.

Analyse

La solution « $t \leftarrow x$; $x \leftarrow y$; $y \leftarrow t$ » où t est une variable auxiliaire est bien connue : il s'agit d'en établir la correction puis de l'énoncer dans le style imposé.

Spécification

L'assertion d'entrée de l'algorithme est

$x = a \wedge y = b$,

où a et b désignent les valeurs initiales de x et y, et l'assertion de sortie

$x = b \wedge y = a$.

Preuve de la correction partielle

$$\begin{aligned} x = a \wedge y = b &\xrightarrow{t \leftarrow x} y = b \wedge t = a \\ &\xrightarrow{x \leftarrow y} t = a \wedge x = b \\ &\xrightarrow{y \leftarrow t} x = b \wedge y = a. \end{aligned}$$

Preuve de la terminaison

La séquence n'étant composée que d'affectations réduites à des copies de contenus, son exécution se termine.

Énoncé de l'algorithme

Algorithme : échange des valeurs de deux variables
Modifie : les valeurs des variables x et y en les échangeant
Assertion d'entrée : $x = a \wedge y = b$
Assertion de sortie : $x = b \wedge y = a$

$t \leftarrow x$
$x \leftarrow y$
$y \leftarrow t$

Implantation

Voici l'une des implantations possibles en C de l'algorithme précédent :

```
1 //  int_swap : échange les valeurs des deux variables de type int d'adresses p1
2 //    et p2.
3 //  AE : *p1 == a && *p2 == b
4 //  AS : *p1 == b && *p2 == a
5 void int_swap(int *p1, int *p2) {
6   int t = *p1;
7   *p1 = *p2;
8   *p2 = t;
9 }
```

Il faut souligner au passage les conventions d'écriture des éléments de preuve dans les commentaires : `AE` et `AS` sont les abréviations respectives de « assertion d'entrée » et « assertion de sortie » ; la formulation s'efforce de rester au plus proche de l'expressivité du C. Ces conventions sont complétées dans l'exemple suivant.

2.7 Exemple : factorielle

Problème

Étant donné un naturel n, décrire une méthode de calcul de $n!$.

Analyse

En fin de calcul, la valeur d'une certaine variable naturelle f doit être $n!$.

Spécification

L'assertion d'entrée de l'algorithme est

P : aucune

et l'assertion de sortie

$R : f = n!$.

Énoncé du futur invariant de boucle

Supposer le problème partiellement résolu *peut amener* à l'expression de l'assertion Q définie par :

$Q : 0 \leqslant k \leqslant n \wedge f = k!$

où k est une variable entière.

Sortie de la future boucle

Soit C la condition définie par

$C : k < n$.

Alors

$Q \wedge \neg C \Rightarrow R$

puisque

$$\begin{aligned} Q \wedge \neg C &\Rightarrow 0 \leqslant k \leqslant n \wedge f = k! \wedge k \geqslant n \\ &\Rightarrow k = n \wedge f = k! \\ &\Rightarrow f = n!. \end{aligned}$$

Induction

Soit A' l'instruction définie par

$$A' : \left| \begin{array}{l} k \leftarrow k + 1 \\ f \leftarrow f \cdot k \end{array} \right.$$

Alors

$Q \wedge C \stackrel{A'}{\Longrightarrow} Q$

puisque

$$\begin{aligned} Q \wedge C \Rightarrow & \quad 0 \leqslant k \leqslant n \wedge f = k! \wedge k < n \\ \Rightarrow & \quad 0 \leqslant k < n \wedge f = k! \\ \Rightarrow & \quad 0 \leqslant k \leqslant n - 1 \wedge f = k! \\ \stackrel{k \leftarrow k+1}{\Longrightarrow} & \quad 1 \leqslant k \leqslant n \wedge f = (k-1)! \\ \stackrel{f \leftarrow f \cdot k}{\Longrightarrow} & \quad 1 \leqslant k \leqslant n \wedge f = k! \\ \Rightarrow & \quad 0 \leqslant k \leqslant n \wedge f = k!. \end{aligned}$$

De plus, la quantité k est entière, croit d'une unité lors de l'exécution de l'instruction A' et est majorée par n.

Base

Soit A l'instruction définie par

$$A : \left| \begin{array}{l} k \leftarrow 0 \\ f \leftarrow 1 \end{array} \right.$$

Alors

$$P \overset{A}{\Longrightarrow} Q$$

puisque

$$\begin{array}{ll} P \Rightarrow & n \geqslant 0 \\ \overset{k \leftarrow 0}{\Longrightarrow} & n \geqslant 0 \wedge k = 0 \\ \overset{f \leftarrow 1}{\Longrightarrow} & n \geqslant 0 \wedge k = 0 \wedge f = 1 \\ \Rightarrow & 0 \leqslant k \leqslant n \wedge f = k!. \end{array}$$

Énoncé de l'algorithme

Algorithme : factorielle
Reçoit : un naturel n
Transmet : la factorielle de n, valeur terminale de la variable naturelle f
Assertion d'entrée : aucune
Assertion de sortie : $f = n!$

$k \leftarrow 0$
$f \leftarrow 1$
Invariant de boucle : $0 \leqslant k \leqslant n \wedge f = k!$
Quantité de contrôle : k
tant que $k < n$ **faire**
    $k \leftarrow k + 1$
    $f \leftarrow f \cdot k$

Implantation

Voici l'une des implantations possibles en C de l'algorithme précédent :

```c
//  int_fact : renvoie n!, la factorielle de n.
//  AE : n >= 0
//  AS : int_fact == n!
int int_fact(int n) {
  int k = 0;
  int f = 1;
  //  IB : 0 <= k && k <= n && f == k!
  //  QC : k
  while (k < n) {
    k = k + 1;
    f = f * k;
  }
  return f;
}
```

Suite des conventions d'écriture des éléments de preuve : `IB` et `QC` sont les abréviations de « invariant de boucle » et « quantité de contrôle » ; l'identificateur de la fonction est utilisé dans l'assertion de sortie pour signifier la valeur renvoyée.

Exercices

► **É 2.1**

1) Qu'entend-on par « algorithme » ? par « programme » ?

2) Qu'a-t-on montré lorsque l'on a établi la correction partielle d'un programme ? Que reste-t-il à montrer pour en avoir établi la correction ?

3) Qu'est-ce qu'une assertion ? Quel est le rôle des assertions ?

4) Qu'est-ce qu'un schéma-pré-post ? une pré-assertion ? une post-assertion ?

5) Qu'est-ce que la spécification d'un programme ?

6) Donnez l'axiome et les trois règles d'inférence fondamentales de la logique de Hoare.

7) Qu'est-ce qu'un invariant ? un invariant de boucle ?

8) Qu'est-ce qu'une quantité de contrôle ? Quel est le rôle des quantités de contrôle ?

9) Qu'entend-on par « énoncé détaillé » d'un algorithme ?

► **É 2.2**

Soient x, y et k trois variables entières. Donnez, selon les cas, la plus fine des formulations possibles à la post-assertion Q ou la plus large des formulations possibles à la pré-assertion P dans les schémas pré-post suivants :

1) $x = 5 \wedge y = 2 \xrightarrow{y \leftarrow y+1} Q$.

2) $x = 5 \wedge y = 2 \xrightarrow{y \leftarrow y+x} Q$.

3) $x = y + 1 \xrightarrow{y \leftarrow y+1} Q$.

4) $x = y \xrightarrow{y \leftarrow y+1} Q$.

5) $x = y \xrightarrow{y \leftarrow y-1} Q$.

6) $x \neq 0 \wedge y$ est un multiple de $x \xrightarrow{y \leftarrow y-x} Q$.

7) $y = kx \xrightarrow{y \leftarrow y+x} Q$.

8) $y = (k+1)x \xrightarrow{k \leftarrow k+1} Q$.

9) $y = (k+1)^2 x \xrightarrow{k \leftarrow k+1} Q$.

10) $y = k^2 x \xrightarrow{k \leftarrow k+1} Q$.

11) $y = k^2 x \xrightarrow{x \leftarrow k \cdot x} Q$.

12) $P \xrightarrow{y \leftarrow y+1} y = 1$.

13) $P \xrightarrow{y \leftarrow y+1} x = y$.

14) $P \xrightarrow{y \leftarrow y+1} x = y + 1$.

15) $P \xrightarrow{y \leftarrow y+k} y \leqslant x$.

16) $P \xrightarrow{y \leftarrow 0} y \geqslant 0$.

► **É 2.3**

Pour ranger dans l'ordre croissant de leur taille une ribambelle d'objets, la méthode suivante est appliquée :

tant que deux objets à des places différentes sont tels que celui situé devant l'autre lui est de taille strictement supérieure **faire**
    échanger ces deux objets de place

Établissez sa correction partielle (l'établissement de la terminaison de la méthode, et donc celle de sa correction, est l'objet d'un exercice qui figure dans le dernier chapitre de ce support).

► **É 2.4**

Soient n et k des variables entières et b une variable booléenne. Soient A et A' des instructions qui se terminent et qui ne modifient pas l'état de ces trois variables et C une condition qui ne

modifie pas l'état des trois variables. Montrez, à force d'invariants de boucle et de quantités de contrôle appropriés, que chacune des séquences d'instructions qui suivent se termine :

1)
```
k ← 0
tant que k < n faire
    k ← k + 1
    A
```

2)
```
k ← 1
tant que k ⩽ n faire
    k ← k + 1
    A
```

3)
```
k ← 0
tant que k < n faire
    A
    k ← k + 1
```

4)
```
k ← n
tant que k > 0 faire
    k ← k − 1
    A
```

5)
```
k ← n
tant que k > 0 faire
    A
    k ← ⌊k/2⌋
```

6)
```
b ← FAUX
k ← 0
tant que ¬b ∧ k < n faire
    si C alors
        A
        b ← VRAI
    sinon
        A′
        k ← k + 1
```

Suggestion : traitez à part le cas où la valeur de n est strictement négative.

► **É 2.5**

Revisitez la solution proposée plus haut pour le calcul de la factorielle d'un naturel n : procédez cette fois en décroissant sur les facteurs du produit.

► **É 2.6**

Construisez deux algorithmes corrects qui, pour tout naturel n donné, calculent H_n, le nombre harmonique de rang n : tandis que le premier procèdera en sommant des termes décroissants, le second procèdera en sommant des termes croissants.

► **É 2.7**

Construisez un algorithme correct, de données un naturel non nul a et un naturel n, de résultat le terme $u_{a,n}$ défini par la récurrence

$$u_{a,n} = \begin{cases} a, & \text{si } n = 0\,; \\ u_{a,n-1}/2, & \text{si } n \geqslant 1 \text{ et } u_{a,n-1} \text{ est pair}\,; \\ 3\,u_{a,n-1} + 1, & \text{si } n \geqslant 1 \text{ et } u_{a,n-1} \text{ est impair.} \end{cases}$$

La suite $\langle u_{a,n} \rangle$ est connue sous le nom de suite de Collatz de germe a.

► **B 2.8**

1) Comme beaucoup d'autres avant vous, remarquez que l'aire d'un carré dont la longueur du côté est $k+1$ se déduit de celle du carré dont la longueur du côté est k à l'aide de quelques additions. Déduisez-en un algorithme qui calcule le carré de tout naturel donné à l'aide d'additions.

2) Dans les même conditions, passez au calcul du cube.

► **B 2.9**

Donnez des algorithmes qui calculent la somme partielle jusqu'à l'indice n des séries suivantes en procédant selon les puissances croissantes de x :

1) $\displaystyle\sum_{k\geqslant 0} x^k$.

2) $\displaystyle\sum_{k\geqslant 0} \frac{x^k}{k!}$.

3) $\displaystyle\sum_{k\geqslant 0} (-1)^k \frac{\prod_{j=1}^{k}(2j-1)}{\prod_{j=1}^{k}(2j)} x^k$.

Contraintes : ne faites appel à aucune fonction ; utilisez au plus trois variables. Suggestion : déduisez le terme courant de la série du terme précédent. Remarques : ces séries sont respectivement les développements en série au voisinage de 0 des fonctions $x \mapsto 1/(1-x)$, $x \mapsto \exp(x)$ et $x \mapsto 1/\sqrt{1+x}$; il est donc demandé ici de calculer les troncatures à l'ordre n de ces séries.

► **B 2.10**

Construisez un algorithme correct qui, pour tout naturel n donné, calcule F_n, le terme de rang n de la suite de Fibonacci.

► **B 2.11**

Construisez un algorithme correct qui calcule le plus grand diviseur commun à deux naturels non nuls donnés en utilisant exclusivement la formule valide pour tous naturels non nuls p et q :

$$\text{pgcd}(p,q) = \begin{cases} p, & \text{si } p = q\,; \\ \text{pgcd}(p-q,q), & \text{si } p > q\,; \\ \text{pgcd}(p,q-p), & \text{si } p < q. \end{cases}$$

► **B 2.12**

Considérez un modèle sur les naturels dans lequel les seules opérations définies sont l'affectation, la comparaison à zéro, l'incrémentation et la décrémentation.

1) Construisez un algorithme qui reçoit deux naturels et qui transmet leur somme.

2) Construisez un algorithme qui transmet cette fois la différence des deux naturels en supposant que le premier est supérieur ou égal au second.

3) Construisez un algorithme transmettant VRAI si le premier naturel est strictement inférieur au second naturel et FAUX sinon.

4) Dans un modèle enrichi des trois opérations définies par les trois algorithmes précédents, construisez un algorithme qui reçoit deux naturels et qui transmet leur produit.

► **B 2.13**

Étudiez le problème de la division euclidienne dans le modèle additif : étant donnés deux naturels a et b avec $b \neq 0$, décrire une méthode de calcul du quotient et du reste de la division euclidienne de a par b dans un modèle opératoire où seules les comparaisons, additions et soustractions de naturels sont autorisées.

► **B 2.14**

Considérez un modèle sur les naturels dans lequel les seules opérations définies sont l'affectation, la comparaison et la somme. Construisez deux algorithmes corrects qui, pour tout naturel

non nul n donné, calcule, pour le premier, $\lfloor \mathrm{lb}\, n \rfloor$, la partie entière inférieure du logarithme de base 2 de n, et, pour le second, $\lceil \mathrm{lb}\, n \rceil$, sa partie entière supérieure.

► **MD 2.15**

Même énoncé qu'à l'exercice 2.4 mais pour la seule composition :

$k \leftarrow n$
tant que $k > 1$ **faire**
    si $k \bmod 2 = 0$ **alors**
        A
        $k \leftarrow \lfloor k/2 \rfloor$
    sinon
        A'
        $k \leftarrow k + 1$

► **MD 2.16**

Vos algorithmes solutions de l'exercice 2.14 possèdent-ils une condition d'itération qui compare n avec une expression qui croit à chaque tour de boucle ? ou alors qui se ramène à cela ? Dans l'affirmative, établissez deux autres solutions dont la condition d'itération compare une constante comme 0 ou 1 avec une expression qui décroit à chaque tour de boucle. Sinon, faites le contraire.

► **D 2.17**

Montrez qu'il pourrait être envisagé de se restreindre aux seules compositions séquentielle et itérative. Autrement dit, apportez la preuve que la composition alternative peut se déduire des deux autres compositions fondamentales. Qu'y gagnerait-on ? qu'y perdrait-on ?

► **D 2.18**

1) Quelles sont, parmi les trois règles d'inférence énoncées pour les trois formes fondamentales de composition, celles dont la réciproque est vraie ?

2) L'invariant (tout court) d'une boucle en est-il un invariant de boucle ?

3 C : compléments aux bases

3.1 Du programme source au fichier exécutable

Types de fichiers

Tout fichier écrit en langage C est qualifié de *source*. Deux types de fichiers sources sont distingués :

— les fichiers sources contenant les *déclarations*. Ils sont d'extension « `.h` » et sont dits fichiers *en-tête* (« *header* ») ;

— ceux contenant les *définitions*, d'extension « `.c` ». Le qualificatif *complet* est ajouté pour signifier que le fichier source contient la fonction principale `main`.

Un fichier *objet* est un fichier écrit en code machine, directement compréhensible par le processeur. Les fichiers objets sont d'extension « `.o` ».

Lignes

Une *ligne (source)* d'un fichier source débute après un caractère de fin de ligne, sauf pour la première ligne, et se termine par le caractère de fin de ligne suivant qui n'est pas immédiatement précédé du caractère barre oblique inverse `\`.

Directives

Les directives sont partie intégrante du langage C : elles sont utilisées pour spécifier des réécritures des fichiers sources qui interviennent durant une phase de prétraitement.

Une *directive* est une ligne d'un fichier C qui débute par le caractère dièse `#`, se poursuit soit par la fin de ligne — et il s'agit dans ce cas de la « directive nulle » qui est ignorée —, soit par un mot-clé, d'éventuels autres lexèmes[4] et la fin de ligne. Des blancs[5] peuvent être intercalés avant ou après le dièse. Vu la définition d'une ligne, une directive peut s'écrire sur plusieurs lignes de texte consécutives au moyen de `\`.

Le préprocesseur connait dix-sept directives. À l'exception de trois d'entre elles, elles sont couramment utilisées en *Algorithmique 2*[6]. Afin d'appréhender correctement la suite, il convient d'en introduire dès à présent deux, en plus de la directive nulle : **`#define`**, ainsi que les termes « macroconstante » et « macrofonction » qui l'accompagne, et **`#include`**.

Une *macroconstante* est un identificateur défini par la directive **`#define`** dont la valeur est une suite de caractères pouvant être vide. La définition est valide jusqu'à la fin du fichier source (ou jusqu'à ce que la directive **`#undef`** y mette fin).

`#define` *identificateur chaine-de-substitution fin-de-ligne*

Lors du prétraitement, chaque ligne de code est réécrite en remplaçant chacune des occurrences des macroconstantes qui n'est pas un facteur d'un lexème strictement plus long par sa chaine de substitution. Le mécanisme de réécriture est récursif[7] et ne prend fin que lorsque la ligne ne contient plus aucune macroconstante.

La paramétrisation des identificateurs qui suivent la directive **`#define`** est possible. De tels identificateurs sont alors appelés *macrofonctions* :

`#define` *identificateur* **`(`** *paramètres* **`)`** *corps fin-de-ligne*

Lors du prétraitement, les paramètres d'une macrofonction sont évalués avant d'être substitués à leurs occurrences dans le corps de la macrofonction.

L'inclusion d'un fichier est pilotée par la directive **`#include`**. Trois formes sont possibles :

`#include` **`<`***nom-de-fichier***`>`** *fin-de-ligne*
`#include` **`"`***nom-de-fichier***`"`** *fin-de-ligne*
`#include` *expression fin-de-ligne*

La première est typiquement réservée aux en-têtes de la bibliothèque C standard ; elle est aussi utilisée pour les en-têtes spécifiques de l'environnement de programmation ou qui proviennent d'une bibliothèque correctement installée. La deuxième est utilisée pour les fichiers sources du programme. La troisième doit se ramener à l'une des deux premières une fois l'expression évaluée.

Construction du fichier exécutable

La construction d'un fichier exécutable à partir d'un fichier source complet comporte quatre phases, impliquant quatre logiciels différents :

1) le *préprocesseur* interprète les directives. À la place de chaque directive d'inclusion, il inclut le fichier en-tête spécifié. Le résultat est un fichier source « expansé ». Il s'agit là de la phase de prétraitement ;

2) le *compilateur* traduit alors le fichier source expansé en un fichier écrit en langage machine ;

3) l'*assembleur* construit ensuite à partir du fichier en langage machine un fichier objet ;

4) l'*éditeur de liens* enfin construit le fichier exécutable en combinant éventuellement plusieurs fichiers objets, dont les bibliothèques signifiées (ici) par l'option `-l` de `gcc`.

Si le fichier source n'est pas complet, la construction s'arrête à l'étape 3) : seul un fichier objet est produit.

4. Un lexème est une « unité minimale de signification appartenant au lexique ». L'équivalent anglo-saxon est *token*. Voir la norme et la version française du KR cités en référence par exemple.
5. Les blancs sont les caractères espace et tabulation.
6. Voir note 3.
7. Voir note 2.

3.2 Objets C

Un objet C est une zone mémoire munie d'un type. Le type d'un objet spécifie l'ensemble de ses valeurs possibles et la manière dont les valeurs sont mémorisées et interprétées. Le cardinal de l'ensemble des valeurs possibles d'un objet dépend de la taille de sa zone mémoire. La zone de mémoire minimale adressable en C est le *byte* : la norme précise qu'un byte est composé d'au moins huit bits et que l'ensemble des caractères de l'environnement d'exécution doit pouvoir être codé sur un byte.

Figurent dans la section suivante quelques rappels et approfondissements sur les « types scalaires ». Ces types sont ceux des objets élémentaires du C : leur taille est fixe, de l'ordre de quelques bytes ; ils sont généralement connus du langage propre à la machine et les opérations associées sont gérées directement par le processeur.

D'autres types sont présentés ultérieurement : les types « agrégés » et « unions » dans le chapitre 4, « fonctions » dans le chapitre 6.

3.3 Types scalaires

Les *types scalaires* se subdivisent en deux familles : les types « arithmétiques » et « pointeurs ».

Les *types arithmétiques* correspondent aux objets qui expriment des nombres. Les opérations arithmétiques (somme, différence, produit…) et de comparaison (tests d'égalité, d'infériorité…) leur sont applicables. Les type arithmétiques sont composés de deux sous-familles : les types « entiers » et « flottants ». Les premiers se distinguent des seconds par leur représentation en numération binaire pure qui leur fait tolérer les opérations bit à bit et les décalages.

Les *types entiers* comprennent le type « booléen », les types « entiers signés », « entiers non signés » et « énumérés », tandis que les *types flottants* se subdivisent en « réels flottants » et « complexes flottants ». Les types « caractères » sont inclus dans les types entiers.

Type booléen

Depuis la norme C99, le C met à disposition le type `bool`[8] et deux constantes, `false` (*faux*) et `true` (*vrai*). Ces deux constantes valent respectivement 0 et 1 lorsqu'elles sont utilisées dans des expressions arithmétiques.

Le résultat de la conversion d'une valeur scalaire sur le type `bool` est `false` si elle est égale à zéro et `true` sinon.

Les opérateurs usuels du type `bool` sont `!~`[9], « non », `~ && ~`, « et », `~ || ~`, « ou ».

Types entiers signés et non signés

Le C dispose de cinq types entiers signés désignés par `signed char`, `short int`, `int`, `long int` et `long long int`. Cette liste de types est donnée dans l'ordre croissant de taille, comptée en bytes. La norme impose un nombre minimal de valeurs représentées : dans l'ordre, 2^8, 2^{16}, 2^{16}, 2^{32} et 2^{64}. Il correspond à chacun des cinq types signés un type non signé (autrement dit, positif) de même taille : `unsigned char`, `unsigned short int`, `unsigned int`, `unsigned long int` et `unsigned long long int`.

Les valeurs limites de tous ces types sont précisées dans l'en-tête standard `<limits.h>`, propre à chaque environnement. Elles sont nommées `SCHAR_MAX`, `SHRT_MAX`, `INT_MAX`, `LONG_MAX` et `LLONG_MAX` pour les limites maximales des types signés. Pour les limites minimales des types signés, le suffixe `_MAX` est remplacé par `_MIN`. Pour les limites maximales des types non signés, `U` remplace le `S` initial de `SCHAR_MAX` et le préfixe `U` est ajouté aux limites suivantes.

8. Depuis la norme C23, `bool`, `false` et `true` sont des mot-clés et l'inclusion de l'en-tête standard `<stdbool.h>` est inutile.

9. La marque « ~ » veut signifier l'emplacement d'un opérande. En cas de besoin, elle est indicée : « $\sim_1$ », « $\sim_2$ », etc., afin de pourvoir se référer à l'opérande sans ambigüité.

Les constantes entières décimales (autrement dit, exprimées en base 10) peuvent être suffixées de combinaisons des lettres `L` et `U` (ou les mêmes en minuscule) pour imposer un codage sur les types **`long`** et **`long long`**, versions signée et non signée.

Les chaines de format des fonctions d'entrée et de sortie des familles `scanf` et `printf` se doivent d'utiliser les suites `hhd`, `hd`, `d`, `ld`, `lld` pour les lectures et écritures en base 10 avec transcodage d'entiers signés. Pour les entiers non signés, les `d` sont remplacés par des `u`.

En sus des opérateurs arithmétiques et des opérateurs de comparaisons, le C dispose de quatre opérateurs de modification de bits, `~` $\sim$, complément binaire, $\sim$ `&` $\sim$, « et » bit à bit, $\sim$ `|` $\sim$, « ou » bit à bit, $\sim$ `^` $\sim$, « ou exclusif » bit à bit, ainsi que de deux opérateurs de décalage, $\sim_1$ `<<` $\sim_2$, de $\sim_2$ bits vers la gauche, $\sim_1$ `>>` $\sim_2$, de $\sim_2$ bits vers la droite. Ces opérations se devraient d'être réservées aux types entiers non signés.

L'en-tête standard `<stdlib.h>` ne fournit que deux fonctions mathématiques sur les entiers : valeur absolue et division, lesquelles ne sont déclinées que sur les types entiers signés **`int`**, **`long int`** et **`long long int`**. Voici leurs *prototypes*, c'est-à-dire leurs déclarations incluant les types de leurs paramètres [10] :

```
#include <stdlib.h>
div_t, ldiv_t, lldiv_t
int abs(int j);
long int labs(long int j);
long long int llabs(long long int j);
div_t div(int numer, int denom);
ldiv_t ldiv(long int numer, long int denom);
lldiv_t lldiv(long long int numer, long long int denom);
```

La division renvoie une structure [11] qui contient dans ses deux champs `quot` et `rem` le quotient et le reste de la division des deux arguments dividende et diviseur ; les trois alias [12] **`div_t`**, **`ldiv_t`** et **`lldiv_t`** sont également définis dans le même en-tête. La norme ne spécifie pas le comportement de ces fonctions lorsque leur résultat ne peut être représenté sur le type de retour.

La norme C99 a introduit une famille d'alias (de types) appelés *entiers étendus* qui, sous réserve d'un codage en complément à deux des types entiers, permettent de spécifier des contraintes de taille ou d'efficacité. Ces alias sont définis dans l'en-tête `<stdint.h>` :

— la taille est précisée pour les *entiers taille exacte*. L'alias est préfixé par `int` (signé) ou `uint` (non signé), suivi d'un nombre de bits qui doit valoir 8, 16, 32 ou 64, et suffixé par `_t` ;

— la taille est supérieure ou égale à celle précisée pour les *entiers taille minimale*. L'alias est cette fois préfixé par `int_least` ou `uint_least` ;

— pour les *entiers rapides taille minimale*, la taille est à la fois supérieure ou égale à celle qui est précisée et choisie par le compilateur de telle sorte que les calculs y soient les plus rapides possibles. L'alias est préfixé par `int_fast` ou `uint_fast` ;

— la taille des *entiers taille maximale* permet de représenter tous les entiers disponibles. Les alias sont **`intmax_t`** et **`uintmax_t`**.

Les valeurs limites sont définies dans le même en-tête ; leurs noms s'obtiennent en remplaçant le suffixe `_t` des alias par `_MIN` ou `_MAX` et en capitalisant le tout. Par exemple : `INT32_MIN`, `UINT_LEAST16_MAX`, `INT_FAST8_MIN`, `INTMAX_MAX`.

10. Dans le présent document, toute liste de prototypes de fonctions standards est précédée de la directive `#include` suivie du nom de l'en-tête à laquelle ces fonctions appartiennent (la seule entorse à la norme est la non-transcription de l'attribut **`restrict`**). Cette règle s'applique également aux autres identificateurs standards : alias, macroconstantes, macrofonctions, variables.

11. Structure : voir chapitre 4.

12. Alias : voir section 3.7.

MÀJ 28-01 fonctions, identificateurs standard⊕s⊕

L'en-tête standard `<inttypes.h>` inclut et étend l'en-tête `<stdint.h>` en ajoutant les suites de caractères qui doivent être utilisées comme chaines de format. Leurs noms sont préfixés par `SCN` (pour une utilisation avec `scanf`) ou `PRI` (avec `printf`), suivis d'une lettre spécifiant la conversion, `d` (signé) ou `u` (non signé) pour la base 10 par exemple, puis de *n*, taille exacte, `LEAST`*n*, taille minimale, `FAST`*n*, rapides taille minimale, et `MAX`, taille maximale, où *n* est le nombre de bits. Le même en-tête met à disposition les fonctions valeur absolue et division sur le type entier signé de taille maximale désigné par **`intmax_t`** :

```
#include <inttypes.h>
imaxdiv_t
intmax_t imaxabs(intmax_t j);
imaxdiv_t imaxdiv(intmax_t numer, intmax_t denom);
```

Tout objet et toute expression d'un type entier de taille inférieure à celle des types **`int`** et **`unsigned int`** bénéficie de la *promotion entière* lors de son évaluation : si le type **`int`** peut contenir toutes les valeurs du type original, il est automatiquement converti dans le type **`int`** ; sinon, dans le type **`unsigned int`**. En conformité avec cette règle, aucune fonction standard n'a de paramètres ni de type de retour **`char`** ou **`short int`**, signé ou non signé.

Types caractères

Tout caractère de base [13] est codé par un nombre entier. Trois types caractères sont définis en C : **`char`**, **`signed char`** et **`unsigned char`**. La conversion d'un type à l'autre n'induit aucune perte de précision.

Le type **`char`** est le plus utilisé. Mais, selon l'implantation, ses valeurs peuvent être signées ou non signées. Pour le savoir, il suffit de tester les valeurs limites `CHAR_MIN` et `CHAR_MAX` de l'en-tête standard `<limits.h>` : ces limites sont respectivement égales à `SCHAR_MIN` et `SCHAR_MAX` ou à `0` et `UCHAR_MAX` selon que **`char`** est signé ou non. L'implantation se doit de garantir :

— l'existence d'un ensemble « rudimentaire » de caractères, inclus dans l'ensemble des caractères de l'environnement d'exécution, comprenant un caractère de valeur nulle, certains caractères de contrôle, comme la tabulation, la fin de ligne et le retour charriot, les caractères nécessaires à la production de fichiers sources C, à savoir l'espace, les dix chiffres, les vingt-six lettres alphabétiques latines majuscules, leurs pendants minuscules et les symboles de ponctuation ;

— la valeur positive de tout élément de cet ensemble lorsqu'il est codé sur le type **`char`** ;

— le fait que la suite des valeurs des dix chiffres de zéro à neuf soit telle que, hormis le premier, chaque terme vaut un de plus que le précédent [14].

Une constante caractère permet de définir une valeur de type **`int`** codant un caractère de base. Elle est formée d'une paire d'apostrophes encadrant un chiffre, une lettre, un symbole de ponctuation, l'espace, ou une suite d'échappement débutant par la barre oblique inverse. Par exemple : `'0'` ; `'A'`, `'e'` ; `'+'`, `'?'`, `{` ; `'␣'` [15] ; `'\0'` (caractère de valeur nulle, dit « caractère nul »), `'\n'` (fin de ligne), `'\r'` (retour charriot), `'\t'` (tabulation horizontale).

Douze fonctions permettent de tester l'appartenance à une catégorie de caractères donnée. Elles possèdent la même forme de prototype :

```
#include <ctype.h>
int is...(int c);
```

13. Il s'agit des seuls caractères de l'environnement d'exécution. Les caractères « larges » (« *wide characters* ») ne sont pas abordés ici.

14. Il faut bien noter au passage qu'il n'est pas supposé qu'il en est de même pour les lettres latines majuscules d'un côté et minuscules de l'autre.

15. Dans toute constante caractère ou chaine de caractères, le symbole ␣ représente l'espace.

La fonction `isalpha` par exemple permet de tester si son argument est une lettre alphabétique ou non. Figurent dans le même en-tête deux fonctions de conversion :

```
#include <ctype.h>
int tolower(int c);
int toupper(int c);
```

Si l'argument est une lettre majuscule, la première fonction renvoie la lettre minuscule associée ; sinon, elle renvoie l'argument. La seconde fonction effectue la conversion inverse.

Les chaines de format utilisent `c`, que le mode soit signé ou non. Pour les fonctions de sortie de la famille `printf`, l'argument de type **`int`** est converti en **`unsigned int`**.

Types énumérés

Des listes de constantes symboliques (d'identificateurs donc) peuvent être définies dont chaque élément est associé à une constante de type **`int`**. Toute liste de la sorte est appelée *énumération* et appartient à l'ensemble des types dits *énumérés*.

La définition d'une énumération suit la syntaxe [16] :

`enum` *id-énumération* `{` *id-constante* `=` *expression* `,` $\overline{\text{*id-constante* `=` *expression* `,`}}$ `};`

Le `=` permet de fixer explicitement la valeur d'un identificateur. Par défaut, la valeur associée à une constante symbolique est zéro s'il s'agit de la première, un de plus que la valeur de celle qui la précède sinon. La liste peut être terminée par une virgule.

Une fois la définition complète, chacun des identificateurs de constantes *id-constante* peut être utilisé. Lorsque l'identificateur de l'énumération *id-énumération* est précisé dans la définition, **`enum`** *id-énumération* est un nouveau type dont les valeurs sont celles des constantes symboliques qu'il définit et dont les opérateurs sont ceux du type **`int`**. Dans le cas contraire, le type est dit *anonyme*.

Types réels flottants

De manière générale, un *flottant* (ou *réel virgule flottante*), parce que son codage est limité à quelques bytes, représente un ensemble de nombres réels : il n'est qu'une approximation de chacun d'eux.

Le C dispose de trois types flottants : **`float`**, dit *simple précision*, **`double`**, *double précision*, et **`long double`**, *précision étendue*, dont la taille des représentations va en ordre croissant. La norme précise certaines valeurs limites minimales ou maximales requises, non détaillées ici.

Les caractéristiques des trois types figurent dans l'en-tête `<float.h>`. Elles y sont préfixées par `FLT_`, `DBL_` et `LDBL_`. Par exemple : `FLT_DIG` précise le nombre maximal de chiffres après le point décimal représentables sans erreur d'arrondi pour le type **`float`** ; `DBL_EPSILON` indique la différence entre 1 et le plus petit flottant strictement supérieur à 1 représentable sur le type **`double`**.

Les fonctions pour les types flottants figurent dans l'en-tête standard `<math.h>`. Certaines prennent un ou plusieurs arguments d'un type flottant donné et renvoient leur résultat sur ce même type. Elles sont dans ce cas déclinées sur les trois types : la racine du nom des fonctions est celui de la fonction destinée au type **`double`** ; pour les types **`float`** et **`long double`**, la racine est suffixée par `f` et `l`. Par exemple pour la fonction logarithme népérien :

```
#include <math.h>
double log(double x);
float logf(float x);
long double logl(long double x);
```

16. Toute partie soulignée est optionnelle. Toute partie surmontée d'un filet peut ne pas apparaitre ou être répétée une ou plusieurs fois.

Certaines autres sont des fonctions de conversions, y compris vers les types entiers, ou de test. Par exemple, les quatre macrofonctions :

```
#include <math.h>
int isfinite( real-floating x);
int isinf( real-floating x);
int isnan( real-floating x);
int signbit( real-floating x);
```

testent respectivement si leur argument correspond à un nombre (ni infini, ni *NaN*), est infini, égal à la valeur spéciale *NaN*[17] ou négatif (bit de signe égal à zéro).

Les chaines de format pour la base 10 utilisent `e`, `E`, `f`, `F`, `g` ou `G`, lettres qu'il faut faire précéder de `l` pour le type **`double`** et de `L` pour le type **`long double`**.

Types complexes flottants

L'en-tête standard `<complex.h>` introduit les trois types **`float complex`**, **`double complex`** et **`long double complex`** qui codent les nombres complexes sous la forme d'un tableau de deux composants du type flottant associé, le premier composant correspondant à la partie réelle, le second, à la partie imaginaire. La macroconstante `I` représente le nombre imaginaire i et les expressions arithmétiques de la forme a `+ I *` b sont tout à fait légales.

De nombreuses fonctions sont disponibles, avec un mécanisme similaire à celui de l'en-tête `<math.h>`. Les fonctions `creal` et `cimag` par exemple renvoient les parties réelle et imaginaire d'un argument **`double complex`**, tandis que leurs homologues suffixées d'un `f` ou d'un `l` le font pour un **`float complex`** et un **`long double complex`**.

Fonctions mathématiques génériques

L'en-tête standard `<tgmath.h>` permet de simplifier l'utilisation de fonctions disponibles dans `<math.h>` et `<complex.h>` par l'introduction de macrofonctions génériques[18]. Les fonctions logarithme népérien `log`, `logf` et `logl` par exemple y sont disponibles sous le nom `log` ; le type de l'expression `log(`x`)` est celui de l'expression x, que ce type soit **`double`**, **`float`** ou **`long double`**.

Conversions explicites et implicites

Pour tout type ou toute expression de type T, l'opérateur `(`T`)` utilisé en notation préfixée force la conversion de son opérande sur le type désigné. Cette opération de conversion explicite est sans contrôle, laissée à l'entière responsabilité du programmeur.

Les cas de conversion implicite sont les suivants :

— par promotion entière ;

— par affectation : 1) soit au moyen d'une expression d'affectation liée à l'opérateur d'affectation `=`, du type de l'expression située à la droite de l'opérateur vers celui de l'expression située à sa gauche ; 2) soit lors d'un appel de fonction, du type du paramètre réel vers celui du paramètre formel ; 3) soit lors d'un retour de fonction, du type de l'expression qui suit le mot-clé **`return`** vers celui de la fonction ;

— dans toute opération arithmétique : 1) si l'un des deux opérandes est d'un type complexe, c'est l'arithmétique complexe qui est mise en œuvre, forçant éventuellement la conversion de l'autre opérande. Sinon, même chose en remplaçant « complexe » par « flottant ». Sinon les deux opérandes sont d'un type entier ; 2) si l'un des deux opérandes est du type **`long double`**, **`double`** ou

17. L'acronyme *NaN* signifie « *not a number* ». Cette valeur spéciale code tout résultat qui ne peut être exprimé par un nombre ou l'un des deux infinis, $-\infty$ et $+\infty$.

18. En informatique, une expression, une fonction ou un module est générique lorsqu'il est paramétré par un type. Avec l'introduction du mot-clé **`_Generic`** par la norme C11, le C dispose dorénavant d'une forme de généricité pilotée par le type des expressions. Les deux premières lettres du nom de l'en-tête `<tgmath.h>` sont l'acronyme de *type-generic*.

`float`, complexe ou non et dans cet ordre, l'autre opérande est converti dans ce type. Lorsque les deux opérandes sont tous deux soit `signed` soit `unsigned`, si l'un des deux opérandes est du type `long long int`, `long int` ou `int`, dans cet ordre, l'autre opérande est converti dans ce type. Dans le cas contraire, la conversion va vers le type qui permet de représenter au mieux l'autre, priorité étant accordée aux types non signés.

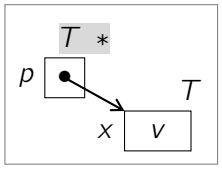
Dans le cas d'une conversion d'une valeur d'un type entier vers un autre type entier, la valeur demeure inchangée si elle peut être représentée sur le second type. Une conversion d'un type flottant vers un autre est indéterminée si la valeur de départ n'est pas dans l'intervalle exprimable par le type d'arrivée ou peut donner lieu à une perte de précision. Il peut également y avoir une perte de précision lors d'une conversion d'un type entier vers un type flottant. Dans le cas inverse, la partie décimale est supprimée ; mais le résultat est indéterminé s'il n'est pas représentable sur le type entier.

La contrainte de compilation `-Wconversion` de `gcc` oblige[19] le programmeur à utiliser un forceur de type dès lors que le compilateur détecte une possible perte de précision.

Pointeurs

Un *pointeur* est une variable ou une constante dont la valeur peut être l'adresse mémoire d'un objet. Si T est un type ou une expression de type, le type *pointeur (d'objets) de (type)* T est noté $T*$. Si p de type pointeur de T a pour valeur l'adresse d'un objet x de type T, on dit de p qu'il *repère* x, qu'il *pointe sur* x ou encore qu'il *référence* x.

le lien entre le pointeur p de type pointeur de T et la variable de nom x, de type T et de valeur v qu'il repère illustré par une flèche

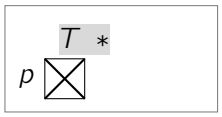


La valeur d'un pointeur peut également être *un pointeur nul*. Dans ce cas, le pointeur ne repère aucun objet. La norme C23 introduit le mot-clé `nullptr`, qui est une constante dont la valeur est celle d'un pointeur nul. La macroconstante `NULL`[20]

```
#include <stddef.h>
NULL
```

a également comme valeur un pointeur nul. Les valeurs pointeurs nuls sont compatibles avec tous les types pointeurs.

l'accès à aucun objet pour le pointeur p de type pointeur de T illustré par une croix



Tandis que l'opérateur de *référence* $\&_\sim$ renvoie l'adresse de l'objet auquel il est appliqué, celui de *déréférence* ou d'*indirection* $*_\sim$, appliqué à un pointeur, désigne l'objet que repère le pointeur.

Une conversion implicite entre pointeurs de deux types différents n'est effectuée que si l'un d'entre eux est le type « pointeur générique » `void *` (voir chapitre 6).

19. Cette contrainte de compilation, pas plus que toutes les autres imposées comme intrinsèques à l'enseignement — voir page 2 —, ne doit pas être vue comme une entrave au programmeur, mais comme une aide précieuse à la production d'un code rigoureux.

20. La macroconstante `NULL` est originellement définie dans l'en-tête `<stddef.h>`. Mais elle est tellement utilisée qu'elle se retrouve l'être dans de nombreux autres, dont `<stdlib.h>` et `<stdio.h>`.

3.4 Préséance des opérateurs

Les opérateurs de construction d'expression sont regroupés selon des niveaux de priorité différents. Ils sont ventilés dans le tableau ci-dessous par *préséance*, c'est-à-dire selon l'ordre décroissant de priorité. Le tableau précise aussi l'*associativité* des opérateurs : la mention « gauche » signifie que si deux opérateurs sont de même niveau, c'est l'opérateur le plus à gauche qui est appliqué en premier ; même chose en remplaçant les occurrences de « gauche » par « droite ».

niveau	*classe*	*opérateur*	*associativité*
1	groupement	`(~)` `[~]` `(type) { ~ }`	gauche
1	sélecteur	`~.~` `~->~`	gauche
2	unaire	`~++` `~--`	gauche

2	unaire	`!~` `~~` `+~` `-~`
		`++~` `--~`
		`*~` `&~`
		`(type) ~`
		`sizeof ~`
			droite

3	multiplicatif	`~ * ~` `~ / ~` `~ % ~`	gauche
4	additif	`~ + ~` `~ - ~`	gauche
5	décalage	`~ << ~` `~ >> ~`	gauche
6	ordre	`~ < ~` `~ <= ~` `~ > ~` `~ >= ~`	gauche
7	identité	`~ == ~` `~ != ~`	gauche
8	binaire	`~ & ~`	gauche
9	binaire	`~ ^ ~`	gauche
10	binaire	`~ | ~`	gauche
11	logique	`~ && ~`	gauche
12	logique	`~ || ~`	gauche
13	alternatif	`~ ? ~ : ~`	droite

14	affectation	`~ = ~`
		`~ += ~` `~ -= ~`
		`~ *= ~` `~ /= ~` `~ %= ~`
		`~ >>= ~` `~ <<= ~`
		`~ &= ~` `~ ^= ~` `~ |= ~`
			droite

15	liste d'expressions	`~, ~`	gauche

Remarques :

— l'opérateur de la classe groupement `(~)` fait référence à la fois au parenthésage des sous-expressions, à l'appel des fonctions et à la déclaration de pointeurs de fonctions (voir chapitre 6 pour les pointeurs de fonctions) ;

— l'opérateur de classe unaire **`sizeof`** produit la taille de son opérande, lequel est une expression ou le nom d'un type placé entre parenthèses. Hormis lorsque l'expression désigne un tableau de longueur variable (voir chapitre 4), le compilateur remplace l'expression de calcul de taille par une constante. La taille est exprimée en bytes et codée sur un type entier non signé de nom **`size_t`**. Lorsque **`sizeof`** est appliqué à un opérande de l'un des types **`char`**, **`signed char`** ou **`unsigned char`**, le résultat est 1 (un). L'alias **`size_t`** est défini dans l'en-tête standard `<stddef.h>`. C'est la lettre `z` qui doit être utilisée dans les chaines de format pour les expressions de ce type.

3.5 Structures de contrôle

En plus des deux compositions fondamentales que sont l'alternative **`if (`**$\sim$**`) `**$\sim$**` else `**$\sim$ et l'itérative **`while (`**$\sim$**`) `**$\sim$, le langage C dispose des compositions :

— conditionnelle **`if (`**$\sim$**`) `**$\sim$;

— sélective **`switch (`**$\sim$**`) `**$\sim$, avec les instructions sélective étiquetée **`case `**$\sim$**`: `**$\sim$, sélective par défaut **`default: `**$\sim$ et de terminaison d'instruction sélective **`break;`** ;

— répétitive **`do `**$\sim$**` while (`**$\sim$**`);`** ;

— itérative **`for (`**$\sim_1$**`; `**$\sim_2$**`; `**$\sim_3$**`) `**$\sim_4$, où $\sim_1$ est une expression ou alors, possibilité offerte depuis la norme C99, une déclaration, de portée réduite à la composition, suivie d'une expression.

3.6 Instructions de saut

Une *instruction de saut* provoque une rupture dans le déroulement d'une suite d'instructions avec un branchement inconditionnel à un autre point du programme. Deux catégories peuvent être distinguées : « branchement » et « échappement ».

L'*instruction de branchement* **`goto`** *identificateur* **`;`** saute à l'instruction préfixée par l'*étiquette identificateur*. Les deux instructions doivent se situer à l'intérieur d'un même sous-programme.

Les *instructions d'échappement* sont :

— **`continue;`** : saute à l'itération suivante de la boucle englobante la plus proche ;

— **`break;`** : termine l'exécution de la boucle englobante la plus proche ;

— **`return;`** et **`return`** *expression* **`;`** : terminent l'exécution de la fonction courante et rendent le contrôle à l'appelant. Dans le cas d'une fonction de type de retour **`void`** (une procédure donc), l'expression *expression* ne doit pas être mentionnée. Dans le cas contraire, elle est obligatoire et d'un type compatible avec le type de retour de la fonction ; la valeur de l'expression est renvoyée à l'appelant comme valeur de l'expression d'appel.

Hormis une instruction préfixée par le mot-clé **`return`** au sein d'une fonction de type de retour non **`void`**, aucune de ces instructions de saut n'est indispensable : tout programme doit pouvoir s'énoncer à l'aide des seules structures de contrôles. Mais si l'énoncé résultant n'est pas lisible ou, dans une moindre mesure, amène à un programme inefficace, il devient intéressant d'y recourir [21]. Ainsi l'instruction **`goto`** est-elle d'un recours précieux dans la gestion des erreurs au sein d'une fonction de taille importante. Et l'autorisation de l'utilisation de multiples **`return`** dans une fonction pousse à traiter les cas triviaux en premier, ce qui ne rend la programmation que plus fluide.

3.7 Compléments aux déclarations

Le qualificatif de type **`const`** indique au compilateur que l'objet auquel il s'applique peut être initialisé mais ne peut en aucun cas être modifié. La seule difficulté dans son emploi est le cas des pointeurs constants, où il doit être placé *après* l'opérateur $*$. Ainsi, si T est un type ou l'expression d'un type, l'expression T **`* const`** décrit un pointeur de valeur non modifiable référençant un objet de type T tandis que les expressions **`const`** T $*$ et T **`const`** $*$ décrivent un pointeur de T référençant une zone non modifiable.

Le constructeur **`typedef`** permet de créer un nouvel identificateur capable de désigner un type. De tels identificateurs sont appelés *alias*, *noms de types* ou encore *types* **`typedef`**. Si *déclaration* **`;`** est valide et permet la déclaration d'un objet de nom *identificateur*, la définition

`typedef` *declaration* **`;`**

est valide et attribue le nom *identificateur* [22] au type de l'objet.

21. Ce n'est pas au prétexte de la logique de Hoare que branchements et échappements doivent être interdits. C'est au contraire au programmeur d'arriver à faire en sorte de faire cohabiter la première avec les secondes.

22. Par abus, l'identificateur ainsi introduit est souvent qualifié de type alors qu'il n'est qu'un alias.

3.8 Erreurs, terminaisons et diagnostics

Notifications d'erreur

Le symbole `errno`

```
#include <errno.h>
errno
EDOM, ERANGE
```

est assimilable à une variable de type **`int`**. Certaines fonctions de la bibliothèque standard lui affectent un code d'erreur non nul lors de leur exécution. Parmi les trois erreurs définies par la norme C23 : `EDOM`, qui indique qu'un ou des arguments n'appartiennent pas au domaine de définition de la fonction ; `ERANGE`, qui indique un dépassement de capacité.

La procédure `perror`

```
#include <stdio.h>
void perror(const char *s);
```

affiche un message correspondant à la valeur courante de `errno` (un message de « succès » si cette valeur est nulle, d'erreur sinon). Si son argument n'est pas un pointeur nul, le message est précédé de la chaine spécifiée par l'argument et du caractère `':'`.

Terminaisons

La procédure `exit`

```
#include <stdlib.h>
void exit(int status);
EXIT_FAILURE, EXIT_SUCCESS
```

provoque la terminaison normale du programme en signifiant à l'environnement d'exécution un succès ou un échec selon que son argument vaut l'une des deux macroconstantes `EXIT_SUCCESS` ou `EXIT_FAILURE`. Toute instruction de la forme **`return`** *expression*`;` qui figure dans la fonction principale `main` est équivalente à `exit(`*expression*`);`.

Diagnostics

La bibliothèque standard propose un mécanisme rudimentaire de définition d'assertion :

```
#include <assert.h>
void assert( scalar  expression);
```

Si l'expression passée comme paramètre à la procédure `assert` est fausse (si donc elle vaut zéro), un message est envoyé sur la sortie erreur puis une terminaison anormale du programme est provoquée. Le message mentionne au moins le texte de l'expression, le nom du fichier source, la ligne du fichier source et le nom de la fonction dans lesquels figure l'appel. Le mécanisme peut être désactivé par définition de la macroconstante `NDEBUG` (*no debug*) avant l'inclusion de l'en-tête.

3.9 Lisibilité

Diverses règles de lisibilité peuvent être trouvées dans la littérature qui ont pour but de faciliter la compréhension des énoncés, la détection des erreurs et les modifications. En voici quelques-unes qui sont utilisées tout au long des supports d'*Algorithmique 1*, *2* et *3*[23], ainsi que dans les sources mises à disposition dans le cadre de ces enseignements : indentation des instructions comme des commentaires ; pas plus d'une instruction par ligne ; lignes d'au plus 80 caractères ; décomposition des instructions longues par l'introduction de variables supplémentaires ; double indentation lors du passage à la ligne dans les instructions longues ; nommage cohérent des identificateurs ; chasse aux nombres et chaines magiques ; homogénéité du style retenu.

23. La matière *Algorithmique 3* est dispensée en deuxième année de la licence Informatique.

Exercices

▶ É 3.1

1) Quelle est la hiérarchie des types scalaires du C ? Présentez-la sous la forme d'un arbre.

2) Comment définir un type énuméré ? Quel est l'intérêt de définir ce genre de type ? Par quoi cet intérêt est-il limité ?

3) En quoi la définition de macroconstantes est-elle intéressante ?

4) Quel est l'intérêt de l'attribut `const` ? du constructeur `typedef` ?

5) Que code le symbole `errno` ?

▶ É 3.2

Que signifient les déclarations suivantes ?

1) `char *p1`.

2) `char * const p2`.

3) `const char *p3`.

4) `const char * const p4`.

▶ É 3.3

Traduisez correctement en C les expressions $\lfloor (j+k)/2 \rfloor$ et $\lceil (j+k)/2 \rceil$, où j et k sont des entiers satisfaisant la double inégalité $0 \leqslant j \leqslant k$, en ne faisant appel qu'à des opérations sur les entiers (sans donc recourir à un opérateur de conversion explicite de type ou à l'en-tête standard `<math.h>` et ses fonctions `floor`, `ceil` ou `trunc` par exemple).

▶ B 3.4

```
#include <stdlib.h>
int rand();
RAND_MAX
```

La fonction `rand` renvoie un nombre pseudo-aléatoire compris entre zéro et la macroconstante entière positive `RAND_MAX`. Toutefois, les bits de poids faibles des nombres générés ont la réputation de posséder une distribution très peu aléatoire.

Donnez en conséquence le code de fonctions C qui renvoient, pseudo-aléatoirement :

1) l'entier zéro ou un ;

2) un réel flottant dans l'intervalle $[0\,;\,1[$;

3) un réel flottant dans l'intervalle $[a\,;\,b[$ où a et b sont les valeurs des deux arguments de type `double` passés à la fonction ;

4) un entier du type `int` dont la valeur est comprise au sens large entre `INT_MIN` et `INT_MAX`, macroconstantes définies dans l'en-tête `<limits.h>`.

▶ B 3.5

Écrivez une procédure C de deux paramètres d'un même type entier T qui affiche toutes les valeurs entières de la valeur du premier paramètre à celle du second, en croissant, et sans recourir à un autre type que T.

▶ B 3.6

Proposez deux implantations enfin satisfaisantes de la fonction factorielle (voir page 13 et exercice 2.5) : en cas de débordement, la première mettra fin à l'exécution *via* la procédure `assert` et la seconde renverra zéro. Recourez nécessairement aux macroconstantes limites définies dans les en-têtes standards `<limits.h>` ou `<stdint.h>`.

MÀJ 28-01 en-têtes standard⊕s⊕

▶ MD 3.7

Étudiez les (éventuelles) différences entre la définition mathématique du quotient et du reste de la division euclidienne d'un *entier* a par un *entier* non nul b (voir le GKP par exemple) et la définition des opérations quotient et reste telles que proposées par la norme C23.

4 C : types agrégés et unions

Les constructions « tableau », « structure » et « union » permettent d'obtenir de nouveaux types à partir d'autres déjà définis. Tandis qu'un tableau est un objet *homogène*, assemblant des composants d'un même type, une structure et une union sont des objets *hétérogènes*, qui ont la possibilité d'en assembler de types divers. Les zones mémoires allouées à ces assemblages sont contigües. Mais, si les composants sont agrégés[24] pour un tableau et une structure, ils se chevauchent pour une union. L'accès à chaque composant d'un objet tableau, structure ou union est *direct* : il ne nécessite pas une série d'opérations sur les autres composants.

4.1 Tableaux

Généralités

La déclaration d'un tableau s'effectue au moyen de l'opérateur `[~]`. Lorsqu'il est présent, le paramètre spécifie explicitement la *longueur* du tableau, c'est-à-dire son nombre de composants. L'opérateur peut être utilisé plusieurs fois de suite afin de déclarer un tableau à plusieurs *dimensions*.

Chaque composant d'un tableau est identifié par son *indice*. L'indice du premier composant est zéro, celui du dernier, la longueur du tableau moins un. L'accès à tout composant d'un tableau s'effectue au moyen de l'opérateur `[~]`, le paramètre spécifiant l'indice du composant.

Si l'opérateur `[~]` est utilisé sans paramètre dans la définition d'un tableau, ce dernier doit être initialisé dans cette même définition par une liste de valeurs. Le nombre de valeurs de la liste est obligatoirement non nul et définit implicitement la longueur du tableau. La forme d'une liste de valeurs pour un tableau est :

`{` $\underline{\texttt{[}\textit{indice}\texttt{]} \texttt{=}}$ *valeur* $\overline{\texttt{,}\ \underline{\texttt{[}\textit{indice}\texttt{]} \texttt{=}}\ \textit{valeur}}$ $\underline{\texttt{,}}$ `}`

avec, comme pour une énumération, la possibilité de fixer l'indice courant à l'aide de `=`. À noter que l'initialisation d'un tableau de caractères de base peut être abrégée, les formes :

type-caractère identificateur`[] = "`$c_0 c_1 \ldots c_{n-1}$`";`
type-caractère identificateur`[] = { `c_0`, `c_1`, ... `c_{n-1}`, '\0' };`
type-caractère identificateur`[`$n+1$`] = { `c_0`, `c_1`, ... `c_{n-1}`, '\0' };`

étant strictement équivalentes.

La norme C99 a rendu possible la déclaration de tableaux de *longueur variable*. La longueur et la taille de ce genre de tableau ne sont déterminées qu'à l'exécution du programme, en fonction de l'expression spécifiée à l'opérateur `[~]`. Un espace mémoire est alloué au tableau sur la pile d'exécution jusqu'à la fin de l'exécution du bloc dans lequel la déclaration figure. Attention : programmer de la sorte est dangereux puisque le langage C ne met aucun moyen à disposition pour tester si une telle allocation est possible[25].

Approfondissements

— Un tableau n'est pas une variable mais une suite finie de variables. Il n'est par exemple pas possible de modifier son contenu au moyen d'une simple affectation ou de faire renvoyer son contenu par une fonction.

— Les composants d'un tableau sont juxtaposés. La taille d'un tableau est ainsi le produit de sa longueur par la taille du type de ses composants.

— L'identificateur *a* d'un tableau est une constante symbolique dont la *valeur* est l'adresse du premier composant du tableau : *a* `==` `&`*a*`[0]`.

24. Le terme *agrégé* vient directement de la norme, « *aggregate* ».

25. Pas plus que le C ne le fait pour les tableaux de longueur fixée à la compilation. Le problème de l'allocation au gré des besoins est résolu en C par les fonctions d'allocation dynamique de mémoire proposées par l'en-tête `<stdlib.h>`. Voir *Algorithmique 2*.

— Si T est le type des composants d'un tableau, le type de l'identificateur du tableau est pointeur de T.

— Lorsqu'un tableau est passé en paramètre à une fonction, c'est la valeur de son identificateur qui est transmise et pas son contenu.

— En tant que paramètres formels dans une définition de fonction, les formes suivantes sont équivalentes :

type identificateur `[`*longueur*`]`
type identificateur `[]`
type ∗*identificateur*

— Si a est l'identificateur d'un tableau de composants de type T et p un pointeur de T de valeur a, les notations `a[k]` et `p[k]` désignent le même objet.

— Si la notation `a[k]` est syntaxiquement correcte, elle ne désigne un objet du tableau désigné par a, à savoir son composant d'indice k, que si la valeur de k est comprise entre zéro et la longueur du tableau moins un.

4.2 Structures

Généralités

Chaque composant d'une structure est appelé *champ* et est désigné par un identificateur. La construction :

`struct` *id-structure* `{` *type identificateur*`;` $\overline{\text{*type identificateur*;}}$ `}`

permet de définir une structure. Si l'identificateur de la structure est précisé, **`struct`** *id-structure* est un nouveau type. Sinon, la structure est anonyme.

L'accès à tout champ d'une structure s'effectue au moyen du sélecteur `.` (point) suivi de l'identificateur du champ. Une structure peut être initialisée par une liste de valeurs :

`{` *identificateur* `=` *valeur* `,` $\overline{\text{*identificateur* = *valeur* ,}}$ `}`

Approfondissements

— Une structure peut être renvoyée par une fonction. Cette possibilité devient couteuse dès que la taille de la structure dépasse significativement la taille d'un pointeur.

— La norme impose aux compilateurs le respect de l'ordre des déclarations des champs dans leurs implantations des structures.

— La taille d'une structure peut être plus importante que la simple somme des tailles de ses champs, des bytes pouvant être intercalés entre deux champs pour aligner le second des deux champs avec un mot machine ou ajoutés après le dernier champ [26].

— Si p est un pointeur de structure et f l'identificateur d'un champ, la notation `p->f` peut être utilisée en lieu et place de la notation `(*p).f`.

4.3 Unions

Même chose que pour les structures à l'exception de :

— la syntaxe de construction :

`union` *id-union* `{` *type identificateur*`;` $\overline{\text{*type identificateur*;}}$ `}`

— de manière générale, l'utilisation d'une union se doit, pour rester cohérente, de mémoriser le type du dernier objet qui y a été mémorisé ;

— la taille d'une union est le maximum des tailles des champs qui la compose.

26. Il s'agit des *internal and trailing padding bytes*, littéralement de « remplissage interne et à la traine ».

4.4 Exemples

La fonction principale et ses paramètres

MÀJ 28-01 formes standard⊕s⊕

Les deux formes standards de déclaration de la fonction `main` sont :

```
int main(); //  et « int main(void); » de la C11 à la C18
int main(int argc, char *argv[]);
```

La deuxième forme permet de récupérer les arguments d'un exécutable lancé en ligne de commande à l'aide des deux paramètres `argc` et `argv` :

— la valeur de la variable entière `argc` est toujours positive. Elle n'est nulle que si l'environnement d'exécution n'a pu transmettre d'arguments ;

— `argv[argc]` est un pointeur nul ;

— si `argc` est strictement positif, les composants `argv[0]` à `argv[argc - 1]` du tableau `argv` sont des pointeurs vers des chaines de caractères ;

— si `argc` est strictement positif, la chaine de caractères pointée par `argv[0]` représente le nom du programme, tandis que celles pointées par `argv[1]` jusqu'à `argv[argc - 1]` représentent ses paramètres, dans l'ordre dans lequel ils figurent sur la ligne de commande.

Une procédure de modification de chaines

```
1  //  str_all_stars : remplace tous les caractères de la chaine pointée par s par
2  //    '*' à l'exception du '\0' final.
3  //  AE : s == { c_{0}, c_{1}, ..., c_{n - 1}, '\0' }
4  //  AS : s == { '*' }^n . { '\0' }
5  void str_all_stars(char *s) {
6    size_t k = 0;
7    //  IB : 0 <= k && k <= n
8    //    && s == { '*' }^k . { c_{k}, ..., c_{n - 1}, '\0' }
9    //  QC : k
10   while (s[k] != '\0') {
11     s[k] = '*';
12     ++k;
13   }
14 }
```

Exercices

▶ É 4.1

1) Comment définir un tableau ?

2) Comment déclarer un tableau en tant que paramètre formel ?

3) Quelle distinction est faite entre le contenu et la valeur d'un tableau ?

4) Comment désigner le composant situé à l'indice k d'un tableau d'identificateur a ? d'un tableau de valeur a ? d'un bloc mémoire repéré par un pointeur p de T, où T est le type des composants d'un tableau ?

▶ É 4.2

La bibliothèque standard ne possède pas de fonction de division pour les types entiers non signés (voir page 20). On se propose de combler ce manque en commençant dans cet exercice par le type `unsigned int`.

1) Définissez à cette fin l'alias `udiv_t`, qui désignera une structure contenant les deux champs `quot` et `rem`, tous deux de type `unsigned int`.

2) Donnez corps à la fonction de prototype

```
udiv_t udiv(unsigned int numer, unsigned int denom);
```

qui, si la division de `numer` par `denom` est possible, renverra une structure `udiv_t` dont les deux champs `quot` et `rem` contiendront le quotient et le reste de cette division, et, dans le cas contraire, donnera à `errno` la valeur `EDOM`.

3) Envisagez une autre expression de la fonction sous la forme

```
int udiv(unsigned int numer, unsigned int denom, udiv_t *divptr);
```

Cette fois, si la division de `numer` par `denom` est possible, les champs `quot` et `rem` de la structure pointée par `divptr` seront affectés du quotient et du reste de la division et la fonction renverra zéro, et, dans le cas contraire, elle renverra une valeur non nulle.

▶ É 4.3

1) Déclarez le type `fellow` (quidam) capable de mémoriser : l'identité d'une personne sous la forme de son nom, d'au plus 31 caractères, de son prénom, même chose, de son genre, masculin, féminin ou autre, et de sa date de naissance, jour, mois et année ; sa situation, célibat, mariage, pacs, divorce ou veuvage ; en cas de mariage ou de pacs, l'identité du conjoint et la date de début du contrat ; en cas de divorce ou de veuvage, la date de fin du contrat. Utilisez nécessairement plusieurs `enum`, `struct` et `typedef` et au moins un `union`.

2) Initialisez deux variables de ce type par liste de valeurs : la première avec une situation à choisir parmi célibat, divorce et veuvage ; la deuxième parmi mariage et pacs.

3) Donnez le code d'une procédure de paramètres un quidam et une date qui, si le quidam est marié ou pacsé et que la date fournie est la date anniversaire de son conjoint, lui rappelle de ne pas oublier de souhaiter l'anniversaire dudit conjoint. Ce rappel est à faire par voie d'affichage, avec titres (« M. », « Mme » ou aucun en attendant mieux), noms et prénoms, pour le quidam comme pour son conjoint.

▶ B 4.4

Dans cet exercice, `element` désigne un type dont les valeurs peuvent être comparées deux à deux à l'aide des opérateurs binaires usuels : `==`, `!=`, `<`, `<=`, `>` et `>=`, et les tableaux ont pour composants des objets du type `element`.

1) La *distance de Hamming* qui sépare deux mots de longueurs égales est le nombre de positions où ces mots diffèrent. Donnez le code de la fonction de prototype

```
size_t hamming(const element *a1, const element *a2, size_t n);
```

qui renvoie la distance de Hamming qui sépare les mots associés aux deux tableaux repérés par `a1` et `a2` et dont la longueur est `n`.

2) Donnez le code de la fonction de prototype

```
size_t llcp(const element *a1, size_t n1, const element *a2, size_t n2);
```

qui renvoie la longueur du plus long préfixe commun (*length of the longest common prefix*) aux mots associés aux tableaux repérés par `a1` et `a2`, de longueurs respectives `n1` et `n2`.

3) Donnez le code de la fonction de prototype

```
bool is_palindrome(const element *a, size_t n);
```

qui renvoie `true` si le mot associé au tableau repéré par `a` et dont la longueur est `n` est un palindrome et `false` sinon.

4) Même énoncé mais en remplaçant « un palindrome » par « croissant » :

```
bool is_increasing(const element *a, size_t n);
```

► **MD 4.5**

On aborde ici sur le problème de l'implantation des ensembles et de leurs opérations basiques que sont l'initialisation au vide, le test de vacuité, le test de présence d'un élément, le cardinal, l'ajout d'un élément, le retrait d'un élément.

Pour commencer, trois cas simples sont proposés à l'étude. Dans chacun de ces cas : le nom du type implantant les ensembles devra être défini et noté `set` ; le type hôte des éléments devra être précisé et justifié.

1) Cas particulier où les éléments sont des entiers naturels inférieurs ou égaux à la macro-constante (positive) `ELEMENT_MAX`, avec codage du type `set` par un tableau de booléens de longueur `ELEMENT_MAX + 1` et avec la convention : pour tout `s` du type `set`, pour tout naturel x inférieur ou égal à `ELEMENT_MAX`,

`s[`x`] ==` (x appartient à l'ensemble implanté par `s`).

2) Même cas que précédemment pour les éléments, mais, cette fois, avec codage du type `set` par une structure à deux champs : tandis que l'un des champs mémorise le cardinal sous la forme d'un entier de type `size_t`, l'autre mémorise l'appartenance des éléments à l'ensemble à l'aide d'un tableau de booléens.

3) Même codage du type `set` par la structure précédente, mais, cette fois, avec des éléments entiers compris au sens large entre les macroconstantes `ELEMENT_MIN` et `ELEMENT_MAX` qui peuvent être toutes deux négatives, toutes deux positives, la première négative et la deuxième positive, mais toujours telles que la première est inférieure ou égale à la deuxième.

5 C : flots

Sont présentées dans ce chapitre quelques-unes des opérations qui permettent le transit de données entre une zone mémoire du programme et une unité périphérique : terminal, disque, imprimante... Ces opérations sont de *haut niveau* : il s'agit de fonctions de l'en-tête standard `<stdio.h>` (pour « *standard input output* ») constituant une interface avec des opérations de *bas niveau* qui s'adressent directement au système d'exploitation.

5.1 Généralités

Opérations

— Une *opération de lecture*, ou *entrée*, consiste en un transit de données depuis une unité périphérique vers une zone mémoire du programme.

— Une *opération d'écriture*, ou *sortie*, est une opération symétrique d'une entrée.

Fichiers

— Toute unité physique est indistinctement appelée *fichier* (« *file* »).

— Un *fichier texte* est une suite de caractères subdivisée en *lignes (de texte)*. Toute ligne est composée de zéro ou de plusieurs caractères différents du caractère de fin de ligne `'\n'` plus le caractère de fin de ligne. Selon l'environnement, la dernière ligne peut ne pas être terminée par le caractère de fin de ligne ; ce n'est toutefois pas le cas sous Unix et Linux ou dans un environnement régi par la norme POSIX[27]. La norme C23 précise que l'environnement doit pourvoir gérer des lignes de texte d'au moins 254 caractères, en incluant le caractère de fin de ligne.

— Un fichier, s'il n'est pas texte, est dit *binaire*.

27. POSIX est un ensemble de normes d'interfaçage des programmes avec les systèmes d'exploitation. Les quatre premières lettres du terme POSIX sont l'acronyme de *Portable operating system interface* ; sa dernière lettre n'est autre que la dernière lettre de Unix.

Flots

— Les données manipulées par le programme transitent par un *flot* (« *stream* »).

— Un *flot texte* permet de gérer un fichier texte en conformité avec l'environnement : lors du transit, des caractères d'espacement ou de contrôle peuvent ainsi être ajoutés, modifiés ou supprimés.

— Un *flot binaire* permet le transit de données internes sans aucune altération.

— La distinction de type de flot texte/binaire n'est réelle que pour des systèmes dans lesquels les applications orientées textes utilisent un format spécifique. Il n'y a aucune distinction de la sorte sous les systèmes Unix et Linux ou dans un environnement POSIX.

— Sous certains systèmes, mais ni sous Unix ni sous Linux, les fichiers binaires sont complétés par des caractères nuls au delà des dernières données écrites et à concurrence d'un multiple d'une constante entière strictement positive propre au système [28].

Contrôle d'un flot

Un flot est contrôlé *via* un objet de type `FILE`,

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

qui est une structure dite *contrôleur (du flot)* contenant les informations nécessaires à la gestion des opérations sur le fichier associé au flot.

Figurent parmi ces informations, en plus du type de flot :

— le *mode d'ouverture* du fichier : en lecture, en écriture ou en lecture-écriture ;

— la *position courante sur le fichier*, c'est-à-dire la position physique, précise au caractère près, à partir de laquelle aura lieu la prochaine opération de lecture ou d'écriture ;

— l'adresse de l'éventuel *tampon*, la zone mémoire utilisée pour le transit par paquets des données, ainsi que l'adresse de la *position courante sur ce tampon* ;

— l'*indicateur de fin de fichier atteinte* ;

— l'*indicateur d'erreur*.

MÀJ 28-01 Entrée et sorties standard⊕s⊕

Entrée et sorties standards

Trois flots textes non modifiables sont prédéfinis :

```
#include <stdio.h>
stdin, stdout, stderr
```

— `stdin`, en lecture sur l'entrée standard, par défaut le clavier ;

— `stdout`, en écriture sur la sortie standard, par défaut l'écran ou la fenêtre associée au processus ;

— `stderr`, en écriture immédiate (sans tampon) sur la sortie erreur, par défaut l'écran ou la fenêtre associée au processus.

Redirection

L'entrée standard, la sortie standard et la sortie erreur peuvent être redirigées au moyen de caractères spéciaux donnés sur la ligne de commande :

— « `0<` *nom-de-fichier* » lit les caractères provenant du fichier spécifié au lieu de ceux provenant de l'entrée standard. La mention du descripteur `0` est facultative ;

— « *d*`>` *nom-de-fichier* » écrit les caractères dans le fichier spécifié au lieu de les écrire sur la sortie standard lorsque le descripteur *d* vaut `1` ou sans mention du descripteur, sur la sortie erreur lorsque *d* vaut `2`, sur les deux sorties lorsque *d* vaut `&` ;

— « *d*`>>` *nom-de-fichier* » les y ajoute.

28. Il s'agit des *trailing null characters* qui produisent le *padding null character*. Voir note 26.

5.2 Association, ouverture et fermeture

```
#include <stdio.h>
FILE *fopen(const char *filename, const char *modes);
FILE *freopen(const char *filename, const char *modes, FILE *stream);
int fclose(FILE *stream);
EOF
FOPEN_MAX
FILENAME_MAX
```

Ouverture

La fonction `fopen` permet l'initialisation d'un nouveau flot :

— la chaine de caractères pointée par `filename` spécifie le nom du fichier, avec son éventuel chemin, à associer au flot ;

— la chaine de caractères pointée par `modes` précise à la fois le mode d'ouverture, l'action éventuelle sur le fichier et le positionnement initial. Le premier caractère de la chaine est, soit `r` (« *read* »), pour signifier le mode *lecture*, soit `w` (« *write* »), pour signifier le mode *écriture*, soit `a` (« *append* »), pour signifier le mode *ajout*. Au-delà du premier caractère, la chaine peut se poursuivre avec les caractères `+`, pour signifier le mode *mise à jour*, `b` (« *binary* »), pour spécifier que le flot est binaire, puis `x` (« *exclusive* »), pour signifier le mode *création exclusive* ;

— s'ils interviennent tous les deux dans la chaine pointée par `modes`, les caractères `+` et `b` peuvent être placés indifféremment l'un avant l'autre ;

— l'ouverture en mode écriture vide le fichier s'il existait et le crée sinon ;

— l'ouverture en mode ajout force toutes les sorties à être réalisées en fin de fichier. Sous les systèmes qui complètent les fichiers binaires par des caractères nuls, le mode ajout peut faire effectuer la première sortie au delà des données précédemment écrites [29] ;

— le mode création exclusive ne peut être donné qu'en combinaison avec le mode écriture. Le fichier ne doit pas exister avant ;

— la fonction `fopen` ouvre le fichier après l'avoir éventuellement vidé ou créé, lui associe un flot, un contrôleur et éventuellement un tampon, force le positionnement du flot au début du fichier en modes lecture ou écriture, au début ou à la fin du fichier en mode ajout [30], efface les deux indicateurs de fin de fichier atteinte et d'erreur ;

— en cas de succès, elle renvoie un pointeur vers le contrôleur du flot ;

— en cas d'échec, elle renvoie un pointeur nul.

Réouverture

La fonction `freopen` permet de réinitialiser un flot déjà ouvert. Elle :

— force la fermeture du fichier précédemment associé au flot lié au contrôleur pointé par `stream` ;

— agit ensuite comme `fopen` mais en associant le fichier au contrôleur pointé par `stream`.

Fermeture

La fonction `fclose` réalise l'opération symétrique à celle engagée par la fonction `fopen`. Elle :

— force, le cas échéant, la sortie des données encore dans le tampon, ferme le fichier, rend l'espace alloué au tampon et au contrôleur pointé par `stream`, met fin à l'association flot-fichier ;

— en cas de succès, renvoie zéro ;

— en cas d'échec, renvoie la macroconstante entière strictement négative `EOF`.

29. Par souci de compatibilité avec de tels systèmes, les caractères `a` et `b` ne devraient pas figurer en même temps dans la chaine pointée par `mode`.

30. Le positionnement initial en mode ajout est *implementation-defined*, « dépendant de l'implantation ».

Limites et fermeture implicite

— La macroconstante `FOPEN_MAX` définit le nombre de fichiers que l'implantation certifie pouvoir être ouverts en même temps. La norme impose à ce nombre d'être au moins égal à 8.

— La macroconstante `FILENAME_MAX` définit la taille d'un tableau pouvant contenir le plus long nom de fichier supporté par le système. S'il n'y a pas de limitation *a priori* pour le système, la valeur est arbitrairement grande.

— Toute terminaison normale du programme assure la fermeture des fichiers ouverts. Ce n'est pas le cas pour les terminaisons anormales, notamment avec la procédure `assert`.

5.3 Lecture et écriture de caractères

Les fonctions de lecture et d'écriture d'un seul caractère à la fois codent les caractères sur le type **`int`** afin de pouvoir distinguer un succès, renvoi d'une valeur positive correspondant à la conversion du caractère sur le type **`unsigned char`**, d'un échec, renvoi de `EOF`.

```
#include <stdio.h>
int fgetc(FILE *stream);
int getchar();
int fputc(int c, FILE *stream);
int putchar(int c);
int ungetc(int c, FILE *stream);
```

Lecture de caractères

La fonction `fgetc` :

— entre un caractère depuis le flot lié au contrôleur pointé par `stream` et avance d'une position sur le fichier associé ;

— en cas de succès, renvoie le caractère lu ;

— en cas d'échec, pour cause d'indicateur de fin de fichier déjà activé, de détection de fin de fichier ou d'erreur survenue en lecture, active l'indicateur de fin de fichier dans les deux premiers cas, l'indicateur d'erreur dans le dernier cas et renvoie `EOF` dans les trois cas.

Écriture de caractères

La fonction `fputc` est symétrique de `fgetc`. Elle :

— sort le caractère `c`, converti sur le type **`unsigned char`**, dans le flot lié au contrôleur pointé par `stream` ;

— en cas de succès, renvoie le caractère ;

— en cas d'échec, pour cause d'erreur en écriture, active l'indicateur d'erreur pour le flot et renvoie `EOF`.

Raccourcis

— Les appels `getchar()` et `fgetc(stdin)` sont équivalents.

— De même les appels `putchar(~)` et `fputc(~, stdout)`.

Réinjection de caractères

La fonction `ungetc` :

— réinjecte le caractère `c`, converti sur le type **`unsigned char`**, dans le flot lié au contrôleur pointé par `stream`. Le caractère ainsi réinjecté est disponible pour une nouvelle entrée ;

— en cas de succès, renvoie le caractère réinjecté ;

— en cas d'échec (entre autres causes parce que la valeur du caractère `c` est égale à celle de la macroconstante `EOF`), renvoie `EOF` ;

— la norme ne garantit qu'un nombre de réinjections successives égal à un ;

— la fonction est compatible avec les fonctions de lecture `fgetc`, `fgets`, `fscanf` et `fread`.

5.4 Lecture et écriture de chaines

Les fonctions de lecture et d'écriture de chaines sont spécialement conçues pour traiter l'entrée et la sortie de lignes de texte.

```c
#include <stdio.h>
char *fgets(char *s, int n, FILE *stream);
int fputs(const char *s, FILE *stream);
```

Lecture de chaines

La fonction `fgets` :

— entre, depuis le flot lié au contrôleur pointé par `stream`, un nombre de caractères strictement inférieur au nombre spécifié par `n` en s'arrêtant au premier caractère de fin de ligne `'\n'` rencontré. Les caractères lus sont stockés dans le tableau pointé par `s`. La marque de fin de chaine `'\0'` est placée immédiatement après le dernier caractère stocké dans le tableau ;

— en cas de succès, renvoie `s` ;

— en cas d'échec, pour cause d'erreur en lecture ou de détection de fin de fichier avant la lecture du premier caractère, renvoie un pointeur nul.

Écriture de chaines

La fonction `fputs` est symétrique de `fgets`. Elle :

— sort le contenu de la chaine pointée par `s` dans le flot lié au contrôleur pointé par `stream` ;

— en cas de succès, renvoie une valeur positive ;

— en cas d'échec, renvoie `EOF`.

5.5 Lecture et écriture formatées

Les fonctions d'entrée et de sortie formatées réalisent les opérations de transcodage sous le contrôle d'une chaine de format.

```c
#include <stdio.h>
int fscanf(FILE *stream, const char *format, ...);
int scanf(const char *format, ...);
int fprintf(FILE *stream, const char *format, ...);
int printf(const char *format, ...);
```

Lecture formatée

La fonction `fscanf` permet le transcodage de caractères lus en données internes :

— en cas de succès, elle renvoie le nombre de données qui ont pu être correctement affectées ;

— en cas d'échec, elle renvoie `EOF` si la fin de fichier est atteinte avant toute tentative de transcodage ou parce qu'une erreur est survenue.

Écriture formatée

La fonction `fprintf`, symétrique de `fscanf`, permet le transcodage de données internes en caractères écrits :

— en cas de succès, elle renvoie le nombre de caractères écrits ;

— en cas d'échec, elle renvoie une valeur strictement négative.

Raccourcis

— Les appels `scanf(~)` et `fscanf(stdin, ~)` sont équivalents.

— De même les appels `printf(~)` et `fprintf(stdout, ~)`.

5.6 Lecture et écriture de blocs

Les fonctions de lecture et d'écriture de blocs permettent des transits directs pour les flots binaires. Elles sont particulièrement bien adaptées aux fichiers *homogènes* dont les contenus sont la juxtaposition d'objets de même type.

```
#include <stdio.h>
size_t fread(void *ptr, size_t size, size_t nmemb, FILE *stream);
size_t fwrite(const void *ptr, size_t size, size_t nmemb, FILE *stream);
```

Lecture de blocs

La fonction `fread` :

— entre, depuis le flot lié au contrôleur pointé par `stream` et en utilisant la fonction `fgetc`, `nmemb` objets de taille `size` et les stocke dans le tableau pointé par `ptr` ;

— en cas de succès, renvoie `nmemb` ;

— en cas d'échec, renvoie le nombre d'objets effectivement lus, lequel est strictement inférieur à `nmemb` si `nmemb` n'est pas nul ;

— si `size` ou `nmemb` est nul, renvoie zéro. Dans de tels cas, l'état du flot et celui du tableau demeurent inchangés.

Écriture de blocs

La fonction `fwrite` est symétrique de `fread`. Elle :

— sort les `nmemb` premiers objets de taille `size` du tableau pointé par `ptr` dans le flot lié au contrôleur pointé par `stream` en utilisant la fonction `fputc` ;

— en cas de succès, renvoie `nmemb` ;

— en cas d'échec, renvoie le nombre d'objets effectivement écrits, lequel est strictement inférieur à `nmemb` si `nmemb` n'est pas nul ;

— si `size` ou `nmemb` est nul, renvoie zéro. Dans de tels cas, l'état du flot demeure inchangé.

5.7 Gestion d'un flot

```
#include <stdio.h>
int feof(FILE *stream);
int ferror(FILE *stream);
void clearerr(FILE *stream);
int fflush(FILE *stream);
int fseek(FILE *stream, long int offset, int whence);
SEEK_SET, SEEK_CUR, SEEK_END
long int ftell(FILE *stream);
void rewind(FILE *stream);
```

Indicateurs

— La fonction `feof` renvoie une valeur non nulle lorsque l'indicateur de fin de fichier atteinte est actif pour le flot lié au contrôleur pointé par `stream`.

— Même chose pour `ferror` avec l'indicateur d'erreur.

— La fonction `clearerr` efface les deux indicateurs pour le flot lié au contrôleur pointé par `stream`.

— D'après la norme, les fonctions `fgetc`, et donc `getchar` et `fread`, `fputc`, et donc `putchar` et `fwrite`, `fflush` et `fseek` peuvent activer l'indicateur d'erreur. Rien n'est précisé pour les fonctions `fgets`, `fputs`, `fscanf`, et donc `scanf`, `fprintf`, et donc `printf`, et `ftell`.

Vidage du tampon

Le tampon d'un flot de sortie est vidé dès qu'il est plein ou à la fermeture du flot. Le vidage d'un tel tampon peut être forcé. La fonction `fflush` :

— force la sortie des données dans le tampon du flot lié au contrôleur pointé par `stream` ;

— en cas de succès, renvoie zéro ;

— en cas d'échec, active l'indicateur d'erreur pour le flot et renvoie `EOF`.

Positionnement

Par défaut, les fonctions de lecture et d'écriture opèrent en mode *séquentiel* : chaque lecture ou écriture s'effectue à partir de la position courante sur le fichier et avance cette position d'autant de caractères lus ou écrits. Il s'agit du seul mode possible pour les flots correspondant à des unités physiques séquentielles comme un terminal ou le clavier. Les fichiers ordinaires autorisent des entrées et des sorties en *accès direct* : il est possible de se rendre à n'importe quel endroit du fichier et de modifier en conséquence la position courante.

La fonction `fseek` :

— effectue un déplacement sur fichier associé au flot lié au contrôleur pointé par `stream` et efface l'indicateur de fin de fichier atteinte ;

— `offset` spécifie le déplacement exprimé en nombre de caractères par rapport à la position indiquée par l'origine `whence`. Les valeurs possibles de `whence` sont `SEEK_SET`, pour signifier le début du fichier associé au flot, `SEEK_CUR`, la position courante, et `SEEK_END`, la fin du fichier[31] ;

— en cas de succès, la fonction renvoie zéro ;

— en cas d'échec, elle active l'indicateur d'erreur et renvoie une valeur non nulle.

Attention en mode mise à jour : une sortie ne doit pas être directement suivie d'une entrée sans un appel intermédiaire à la fonction `fflush` ou à une fonction de positionnement (`fseek` ou `rewind` ici) ; une entrée ne doit pas être directement suivie d'une sortie sans un appel intermédiaire à une fonction de positionnement, sauf si l'opération d'entrée rencontre la fin du fichier.

La fonction `ftell` :

— en cas de succès, renvoie la position courante sur le fichier associé au flot lié au contrôleur pointé par `stream` exprimée en nombre de caractères par rapport au début du fichier ;

— en cas d'échec, renvoie `-1` et donne une valeur strictement positive à `errno`.

L'appel `rewind(~)` équivaut à la séquence `() fseek(~, 0, SEEK_SET); clearerr(~)`.

5.8 Manipulations de fichiers

```
#include <stdio.h>
FILE *tmpfile();
TMP_MAX
int remove(const char *filename);
int rename(const char *old, const char *new);
```

Fichiers temporaires

La fonction `tmpfile` :

— crée un fichier temporaire ouvert en mode `w+b`. Ce fichier sera automatiquement supprimé lors de sa fermeture ou d'une terminaison normale du programme ;

— en cas de succès, renvoie un pointeur vers le contrôleur du flot ;

— en cas d'échec, renvoie un pointeur nul.

Au moins `TMP_MAX` fichiers temporaires peuvent être créés. La norme précise que `TMP_MAX` vaut au moins 25.

31. Toujours par souci de compatibilité avec les systèmes qui complètent les fichiers binaires avec des caractères nuls, la norme précise que le comportement de `fseek` avec une origine égale à `SEEK_END` est indéterminé.

Suppression et renommage

— La fonction `remove` supprime le fichier dont le nom est la chaine pointée par `filename`.

— La fonction `rename` force le fichier de nom la chaine pointée par `old` à être désormais nommé par la chaine pointée par `new`.

— En cas de succès, les fonctions `remove` et `rename` renvoient zéro.

— En cas d'échec, elles renvoient une valeur non nulle.

5.9 Idiomes et exigence de correction

Tous les langages informatiques possèdent leurs propres idiomes, ces traits à la fois typiques et conventionnels avec lesquels les programmeurs expérimentés écrivent des blocs de code éprouvés. L'acquisition de tout langage passe inévitablement par la familiarisation avec ses propres idiomes. Le C n'échappe pas à cette règle.

Par exemple, l'une des expressions idiomatiques les plus communes pour effectuer la somme d'entiers qui figurent sur l'entrée standard est :

```
int s = 0;
int x;
while (scanf("%d", &x) == 1) {
  s += x;
}
```

(en mettant toutefois de côté les éventuels problèmes de débordements arithmétiques qui peuvent survenir lors de la lecture ou lors de l'ajout). De même, la boucle standard pour copier tous les caractères d'un fichier associé au flot lié au contrôleur pointé par `src` et ouvert au moins en lecture vers un autre fichier, associé lui au flot lié au contrôleur pointé par `dest` [32] et ouvert au moins en écriture, tout en arrêtant les opérations de lecture et d'écriture dès qu'une erreur survient lors de l'une ou l'autre, s'énonce comme suit :

```
int c;
while ((c = fgetc(src)) != EOF && fputc(c, dest) != EOF) {
}
```

Or, aucune de ces deux boucles n'est immédiatement compatible avec la logique de Hoare. Les expressions de contrôle des boucles font en effet apparaitre à trois reprises des fonctions à effet secondaire [33] et une affectation imbriquée :

— si l'appel à `scanf` réussit, il modifie le contenu de la variable `x` tout en avançant sur l'entrée standard. Et s'il échoue, il active l'un des deux indicateurs de fin de fichier atteinte ou d'erreur ;

— que l'appel à `fgetc` réussisse ou échoue, il modifie, au moyen de l'affectation imbriquée `(c = ...)` dans le premier terme de l'expression de contrôle, le contenu de la variable `c`. S'il réussit, le caractère lu est affecté à `c` et la position sur le fichier est avancée. S'il échoue, la valeur de la macroconstante `EOF` est affectée à `c` et l'un des deux indicateurs est activé pour `src` ;

— en cas de succès de l'appel à `fgetc` : si l'appel à `fputc` réussit à son tour, le contenu du fichier destination est modifié et la position sur ce fichier est avancée ; sinon, l'indicateur d'erreur est activé pour `dest`.

Autrement dit, l'évaluation des expressions de contrôle des boucles modifie des variables, `stdin` et `x` pour la première boucle, `src`, `c` et `dest` pour la deuxième. Ce qui contredit l'unique axiome de la théorie.

32. Avec « src » et « dest » pour « source » et « destination », en français comme en anglais.

33. De manière générale, une opération est *à effet secondaire* (ou *à effet de bord* ; « *side effect* ») lorsque, outre le fait de renvoyer un résultat, elle a la possibilité de modifier son environnement extérieur. C'est le cas de toute fonction C susceptible de modifier le contenu d'une variable non locale, que l'adresse de cette dernière soit passée en paramètre ou non, d'effectuer des entrées-sorties ou d'appeler des fonctions à effet secondaire.

Il est pourtant possible de concilier les deux aspects : le recours raisonné aux idiomes d'un côté, la nécessité de la formulation explicite d'éléments de preuve de l'autre.

Pour le premier des deux exemples, il suffit de sortir l'appel à effet secondaire de l'expression de contrôle en le dupliquant et en mémorisant la valeur renvoyée :

```
int s = 0;
int x;
int rs = scanf("%d", &x);
while (rs == 1) {
  s += x;
  rs = scanf("%d", &x);
}
```

S'il y a dorénavant deux occurrences d'un appel à effet secondaire, l'effet de chacun est bien maitrisé et ne modifie en rien l'évaluation de la condition d'itération. La valeur de la variable `s` est donc la somme de tous les entiers qui ont pu être lus sur l'entrée standard, sans compter le dernier entier lu si la variable `rs`, laquelle stocke la valeur renvoyée par le dernier appel à `scanf`, vaut `1`. Décrire le contenu de la variable `x` est immédiat mais lié à la valeur de `rs` : il s'agit du dernier entier lu sur l'entrée standard si `rs` vaut `1`. Voilà pour l'invariant de boucle. Quant à la quantité de contrôle, il sera systématiquement fait l'hypothèse que l'entrée standard est finie, exactement comme si elle était redirigée vers un fichier (de taille finie). Auquel cas le nombre d'appels à `scanf` est ici majoré par le nombre d'entiers consécutifs présents sur l'entrée plus un, celui des appels qui donne à `rs` une valeur différente de `1`.

D'où l'énoncé détaillé :

```
int s = 0;
int x;
int rs = scanf("%d", &x);
//  IB : rs == (valeur renvoyée par le dernier appel scanf("%d", &x))
//    && x == (dernier entier lu sur l'entrée standard si rs == 1)
//    && s == (somme des entiers lus sur l'entrée standard, sans compter le
//              dernier entier lu si rs == 1)
//  QC : nombre d'appels scanf("%d", &x)
while (rs == 1) {
  s += x;
  rs = scanf("%d", &x);
}
```

À la sortie de la boucle, la valeur de la variable `rs` est `0` ou `EOF`. Dans le premier cas, l'entrée n'est pas épuisée, mais le ou les caractères que vient de lire `scanf` ne constituent pas le début du codage d'un entier sous forme décimale reconnaissable par la chaine de format `"%d"`. Dans le deuxième cas, l'entrée standard est épuisée ou alors une erreur y a été détectée.

Le problème actuel de la somme pourrait être complété par l'exigence que l'entrée standard ne soit constituée que d'entiers pouvant être lus au format `"%d"`. Le tester à la sortie de boucle est immédiat : si l'appel `feof(stdin)` renvoie une valeur non nulle, l'entrée standard est épuisée sans qu'aucune erreur n'ait été signalée, elle n'était constituée que d'entiers au format souhaité et la variable `s` contient la somme de tous ces entiers ; sinon, l'entrée n'est pas épuisée, mais soit la lecture est entravée par un ou des caractères inattendus — premier des cas envisagés dans le paragraphe précédent —, soit une erreur est survenue — deuxième cas. Un test équivalent peut être obtenu en combinant `rs` et un appel à `ferror` : `rs == EOF && !ferror(stdin)`.

Lorsque — comme cela vient d'être évoqué — la valeur finale de la variable `rs` n'est pas utile, l'énoncé détaillé suivant est acceptable :

```
int s = 0;
int x;
//  avec rs == (valeur renvoyée par le dernier appel scanf("%d", &x))
//  IB : x == (dernier entier lu sur l'entrée standard si rs == 1)
//    && s == (somme des entiers lus sur l'entrée standard, sans compter le
//              dernier entier lu si rs == 1)
//  QC : nombre d'appels scanf("%d", &x)
while (scanf("%d", &x) == 1) {
  s += x;
}
```

Il a l'avantage de reprendre la tournure idiomatique. Mais il demande à être interprété : l'expression de contrôle qui suit le mot-clé `while` n'est pas la condition de la boucle au sens de la logique de Hoare ; l'évaluation d'une partie de cette expression, le `scanf`, a lieu avant que ne soit effectuée la partie complémentaire, à savoir le test comparant son résultat avec `1`.

Le deuxième exemple oblige à aller plus loin encore. De manière générale, tout énoncé

```
while (expression) {
  corps
}
```

pour lequel il apparait au moins un appel secondaire ou une affectation imbriquée dans l'expression de contrôle de la boucle demande à être interprété comme

```
bool e = (bool) (expression);
while (e) {
  corps
  e = (bool) (expression);
}
```

où `e` est une variable booléenne libre [34].

Voici à l'avenant une reformulation possible de la boucle du deuxième exemple :

```
bool e;
int c;
A
while (e) {
  A
}
```

avec A :

```
c = fgetc(src);
if (c == EOF) {
  e = false;
} else {
  int d = fputc(c, dest);
  e = (d != EOF);
}
```

L'exécution de la composition A donne aux variables `c` et `e` les valeurs `EOF` et `false` si la source est épuisée ou si une erreur de lecture survient. Sinon, `c` mémorise le dernier caractère lu et `e` vaut `true` ou `false` selon que le caractère obtenu par conversion de `c` sur le type `unsigned char` réussit à être écrit dans le fichier destination ou non. Supposer le problème de la copie en partie résolu avant l'exécution de A amène à l'invariant de boucle suivant :

```
//  IB : c == (valeur renvoyée par le dernier appel fgetc(src))
//    && tous les caractères lus sur src, sauf le dernier si (!e && c != EOF),
//        ont été écrits sur dest
```

34. Une variable est *libre* lorsqu'elle n'a pas été déclarée auparavant.

Le nombre d'appels à `fgetc` est le candidat idéal au poste de quantité de contrôle. En effet, cette expression entière croit strictement à chaque itération et est majorée par la taille du fichier source plus une unité :

```
//  QC : nombre d'appels fgetc(src)
```

Il s'ensuit que, sans avoir à introduire la variable booléenne `e` ni à exprimer la composition A, l'énoncé détaillé suivant est acceptable :

```
int c;
//  avec e == (valeur de l'expression de contrôle de la boucle)
//  IB : c == (valeur renvoyée par le dernier appel fgetc(src))
//    && tous les caractères lus sur src, sauf le dernier si (!e && c != EOF),
//        ont été écrits sur dest
//  QC : nombre d'appels fgetc(src)
while ((c = fgetc(src)) != EOF && fputc(c, dest) != EOF) {
}
```

S'il s'agit à la sortie de la boucle, donc lorsque `!e`, de savoir si la copie a pu être réalisée dans sa totalité, il suffit, comme plus haut, de faire appel à la fonction standard `feof`. Dans le cas où `feof(src)` renvoie une valeur non nulle, `c == EOF` et, d'après l'invariant de boucle, tous les caractères lus sur `src` ont été écrits sur `dest`. Dans le cas contraire, `c != EOF` et le dernier caractère lu n'a pu être écrit. Deux tests équivalents mais beaucoup moins subtils sont possibles : `c == EOF && !ferror(src)` et `!ferror(src) && !ferror(dest)`.

Maintenant qu'a été à la fois affirmée et travaillée l'interprétation à part de l'expression de contrôle de certaines boucles et dès lors qu'elle a été parfaitement assimilée, il devient possible d'obtenir des formulations d'invariants de boucle plus simples encore. Pour le premier exemple :

```
//  sans tenir compte des derniers effets de l'évaluation de l'ECB
//  IB : s == somme des entiers lus sur l'entrée standard
```

et pour le deuxième :

```
//  sans tenir compte des derniers effets de l'évaluation de l'ECB
//  IB : tous les caractères lus sur src ont été écrits sur dest
```

où `ECB` est l'abréviation de « expression de contrôle de la boucle ».

Pour finir, il peut être intéressant de se tourner vers les codes générés en assembleur par le compilateur `gcc` afin de décider laquelle des deux versions, l'idiomatique d'un côté, l'étendue de l'autre, est la meilleure. Que ce soit pour la somme d'entiers, le premier exemple, ou pour la copie de caractères, le deuxième, le compilateur fournit exactement les mêmes suites d'instructions [35] quant à la partie essentielle du traitement : la boucle. Étendre une expression C afin de mieux la comprendre ou d'en pouvoir détailler tous les éléments de preuve n'est donc pas pénalisant.

5.10 Cadre de développement

Pour tous les développements envisagés en C, le modèle suivant est imposé :

— tout fichier ouvert par une fonction doit être explicitement fermé par cette même fonction, à moins que cette fonction ne soit une fonction d'ouverture qui transmette à l'appelant un pointeur vers le contrôleur du flot. Il est donc interdit de confier à toute terminaison normale d'un programme la fermeture des fichiers ouverts [36] ;

35. Options « `-S -O2` » dans un environnement où « `gcc -dumpmachine` » affiche « `x86_64-linux-gnu` ».

36. C'est une saine discipline que de nettoyer avant de sortir. L'utilisation de l'outil de programmation `valgrind` peut aider à détecter si le travail de fermeture a correctement été réalisé. Il suffit de faire précéder le contenu de la ligne de commande de « `valgrind` » puis de regarder si le bilan fourni met en évidence des fuites mémoires ou non.

— afin de pouvoir toujours être en capacité d'exprimer les quantités de contrôle, il est fait l'hypothèse que l'entrée standard est finie ;

— une parfaite compatibilité exigerait de tenir compte : 1) que les fichiers binaires peuvent être complété par des caractères nuls ; 2) qu'il est nécessaire de préciser si le flot est ouvert en mode texte ou binaire. Seule la seconde de ces exigences est retenue ;

— toute erreur testable doit être testée et doit donner lieu à un traitement approprié ;

— la *taille* d'un fichier est le nombre de caractères dont il est constitué ;

— la *longueur* d'un fichier homogène est le nombre des objets dont il est composé. Autrement dit, sa longueur est le quotient de sa taille par la taille (commune) des objets.

5.11 Exemples cumulatifs

Six exemples sont présentés qui balayent assez largement les divers apports du chapitre. Tous ont la forme de fonction dont l'un des paramètres — voire l'unique paramètre — est un pointeur vers un nom de fichier. Quatre d'entre eux sont complets : les spécifications et les définitions sont données. Des extraits choisis des deux autres sont fournis. La notation `^^`[37] est utilisée dans les commentaires pour signifier le « ou exclusif ».

Production de fichiers

Deux versions de la production de fichiers de flottants à partir de l'entrée sont proposées. Il s'agit d'une reprise, à des degrés divers, des deux exemples traités dans la section 5.9. L'expression de contrôle de la boucle suit celle de la copie : opération de lecture avec mémorisation du résultat et test de validité puis opération d'écriture avec test de validité.

La première version crée un fichier texte :

```
1  //  ffloat_copyt_t : tente de créer le fichier texte de nom la chaine pointée
2  //    par filename et d'y écrire les flottants lus au format "%f" sur l'entrée
3  //    standard. Un flottant est écrit par ligne sur le fichier. En cas de
4  //    succès, renvoie zéro. En cas d'échec, parce qu'une erreur survient sur
5  //    l'entrée, que l'entrée n'est pas uniquement constituée de flottants qui
6  //    peuvent être lus au format "%f" ou qu'une erreur survient lors du
7  //    traitement du fichier, renvoie une valeur non nulle.
8  //  AE : aucune
9  //  AS : ffloat_copyt_t == 0 ^^ une erreur est survenue
10 int ffloat_copyt_t(const char *filename) {
11   FILE *f = fopen(filename, "wx");
12   if (f == nullptr) {
13     return -1;
14   }
15   float x;
16   //  sans tenir compte des derniers effets de l'évaluation de l'ECB
17   //  IB : tous les flottants lus sur l'entrée standard ont été écrits sur f
18   //  QC : nombre d'appels scanf("%f", &x)
19   while (scanf("%f", &x) == 1 && fprintf(f, "%f\n", x) >= 0) {
20   }
21   int r = 0;
22   if (!feof(stdin)) {
23     r = -1;
24   }
```

37. Ce n'est qu'une notation, en aucun cas un opérateur du langage C.

```
25   if (fclose(f) != 0) {
26     r = -1;
27   }
28   return r;
29 }
```

Notez que les fonctions indicatrices (ici `feof`) sont assimilées à des fonctions booléennes, contrairement aux autres fonctions sur les fichiers qui renvoient zéro en cas de succès (par exemple `fclose`).

La deuxième version produit un fichier binaire. Plus précisément :

```
1 //  ffloat_copyb_t : tente de créer le fichier homogène de « float » de nom la
2 //    chaine pointée par filename et d'y écrire les flottants lus au format "%f"
3 //    sur l'entrée standard. En cas de succès, renvoie zéro. En cas d'échec,
4 //    parce qu'une erreur survient sur l'entrée, que l'entrée n'est pas
5 //    uniquement constituée de flottants qui peuvent être lus au format "%f" ou
6 //    qu'une erreur survient lors du traitement du fichier, renvoie une valeur
7 //    non nulle.
8 //  AE : aucune
9 //  AS : ffloat_copyb_t == 0 ^^ une erreur est survenue
```

En comparaison de la première version, les seules modifications à apporter sont, outre le nom de la fonction, `ffloat_copyb_t` :

— le mode d'ouverture du fichier, `"wbx"`, avec le `b` par souci de compatibilité ;

— le deuxième membre de l'expression de contrôle de la boucle, à savoir l'appel à la fonction d'écriture et son test de succès, `fwrite(&x, sizeof(float), 1, f) == 1`.

```
10 int ffloat_copyb_t(const char *filename) {
11   FILE *f = fopen(filename, "wbx");
12   if (f == nullptr) {
13     return -1;
14   }
15   float x;
16   //  sans tenir compte des derniers effets de l'évaluation de l'ECB
17   //  IB : tous les flottants lus sur l'entrée standard ont été écrits sur f
18   //  QC : nombre d'appels scanf("%f", &x)
19   while (scanf("%f", &x) == 1 && fwrite(&x, sizeof(float), 1, f) == 1) {
20   }
21   int r = 0;
22   if (!feof(stdin)) {
23     r = -1;
24   }
25   if (fclose(f) != 0) {
26     r = -1;
27   }
28   return r;
29 }
```

Somme

Il s'agit cette fois de calculer la somme des flottants qui figurent dans des fichiers.

La première version, donnée avec une assertion pour chacune des variables, s'adresse aux fichiers textes :

```c
//  ffloat_sum_t : tente de lire la suite des flottants au format "%f" qui
//    figurent dans le fichier texte de nom la chaine pointée par filename. Si
//    une erreur survient lors du traitement du fichier ou que le fichier n'est
//    pas uniquement constitué de flottants qui peuvent être lus au format "%f",
//    renvoie une valeur non nulle. Sinon, affecte la somme à l'objet pointé par
//    sumptr et renvoie zéro.
//  AE : aucune
//  AS : (ffloat_sum_t == 0 && *sumptr == somme des flottants)
//    ^^ (ffloat_sum_t != 0 && une erreur est survenue)
int ffloat_sum_t(const char *filename, float *sumptr) {
  FILE *f = fopen(filename, "r");
  if (f == nullptr) {
    return -1;
  }
  float s = 0.0;
  float x;
  //  avec rs == (valeur renvoyée par le dernier appel fscanf(f, "%f", &x))
  //  IB : x == (dernier flottant lu sur f si rs == 1)
  //    && s == (somme des flottants lus sur f, sans compter le dernier flottant
  //              lu si rs == 1)
  //  QC : nombre d'appels fscanf(f, "%f", &x)
  while (fscanf(f, "%f", &x) == 1) {
    s += x;
  }
  int r = 0;
  if (!feof(f)) {
    r = -1;
  }
  if (fclose(f) != 0) {
    r = -1;
  }
  if (r == 0) {
    *sumptr = s;
  }
  return r;
}
```

La deuxième, avec le même souci du détail, aux fichiers binaires :

```c
//  ffloat_sum_b : tente de calculer la somme des composants du fichier homogène
//    de « float » de nom la chaine pointée par filename. En cas de succès,
//    affecte la somme à l'objet pointé par sumptr et renvoie zéro. En cas
//    d'échec, renvoie une valeur non nulle.
//  AE : aucune
//  AS : (ffloat_sum_b == 0 && *sumptr == somme des composants)
//    ^^ (ffloat_sum_b != 0 && une erreur est survenue)
```

Outre les substitutions du nom de la fonction, `ffloat_sum_b`, et du mode d'ouverture, `"rb"`, cette deuxième version se différencie de la première par sa boucle et les éléments de preuve associés :

```
13   float s = 0.0;
14   float x;
15   //  avec rr == (valeur renvoyée par le dernier appel fread(&x, ..., 1, f))
16   //  IB : x == (dernier composant lu sur f si rr == 1)
17   //    && s == (somme des composants lus sur f, sans compter le dernier
18   //              composant lu si rr == 1)
19   //  QC : nombre d'appels fread(&x, ..., 1, f)
20   while (fread(&x, sizeof(float), 1, f) == 1) {
21     s += x;
22   }
```

Notez le glissement de « flottant » vers « composant » dans les commentaires.

Taille

Les deux définitions données plus bas sont celles d'une même fonction qui calcule la taille du fichier dont le nom lui est passé en paramètre. Voici sa spécification et sa déclaration :

```
1 //  fsize : en cas de succès, renvoie la taille du fichier de nom la chaine
2 //    pointée par filename. En cas d'échec, renvoie -1.
3 //  AE : aucune
4 //  AS : (fsize != -1 && fsize == taille du fichier)
5 //    ^^ (fsize == -1 && une erreur est survenue lors du traitement du fichier)
6 long int fsize(const char *filename);
```

Le type de retour, `long int`, s'accorde avec celui de la fonction `ftell` : la taille d'un fichier n'est autre que le maximum des positions courantes sur le fichier ; si elle peut être calculée, la taille doit donc être codée sur le type de retour de la fonction `ftell`, `long int` ; sinon, une valeur strictement négative peut être renvoyée, et celle à laquelle `ftell` à recours, `-1`, est toute indiquée.

La première définition donne une fonction très lente, puisqu'elle dénombre l'un après l'autre chacun des caractères qui composent le fichier :

```
7  long int fsize(const char *filename) {
8    FILE *f = fopen(filename, "rb");
9    if (f == nullptr) {
10     return -1;
11   }
12   long int n = 0;
13   //  sans tenir compte des derniers effets de l'évaluation de l'ECB
14   //  IB : n == (nombre de caractères lus sur f)
15   //  QC : nombre d'appels fgetc(f)
16   while (fgetc(f) != EOF) {
17     ++n;
18   }
19   if (!feof(f)) {
20     n = -1;
21   }
22   if (fclose(f) != 0) {
23     n = -1;
24   }
25   return n;
26 }
```

La seconde en fournit une assurément plus efficace : elle se contente de se rendre à la fin du fichier afin de connaitre la position courante ; or cette position n'est autre que la quantité attendue, que la requête soit couronnée d'un succès ou d'un échec. Elle s'obtient à partir de la première en remplaçant les lignes 12-21 par :

```
12  long int n;
13  if (fseek(f, 0, SEEK_END) != 0) {
14    n = -1;
15  } else {
16    n = ftell(f);
17  }
```

Exercices

► É 5.1

1) Pourquoi les caractères gérés par les fonctions `fgetc`, `fputc` et `ungetc` sont-ils codés sur le type `int` ?

2) Quelles sont les fonctions dont la norme garantit que le succès ou l'échec de l'exécution puisse être testé par `ferror` ?

3) La taille de tout objet stocké en mémoire est codé sur le type `size_t`. Quel est le type qui code la taille de tout fichier ? Quelle est la conséquence de ce choix ?

4) Sont listés dans l'un des nombreux paragraphes de la norme les seules possibilités quant au contenu de la chaine `mode` passée aux fonctions `fopen` et `freopen`, avec, en vis-à-vis, le mode d'ouverture exprimé de manière à la fois concise et précise. Les contenus des chaines et les expressions ont été reprises dans le tableau qui figure ci-après, mais, malencontreusement, les colonnes du tableau ont été triées chacune dans l'ordre lexicographique. Ré-associez contenus et expressions en faisant correspondre à chaque numéro qui repère un contenu de chaine la lettre qui repère l'expression associée au contenu.

	contenus		*expressions*
1	`a`	A	*append; open or create binary file for update, writing at end-of-file*
2	`ab`	B	*append; open or create binary file for writing at end-of-file*
3	`a+`	C	*append; open or create text file for update, writing at end-of-file*
4	`a+b` *or* `ab+`	D	*append; open or create text file for writing at end-of-file*
5	`r`	E	*create binary file for update*
6	`rb`	F	*create binary file for writing*
7	`r+`	G	*create text file for update*
8	`r+b` *or* `rb+`	H	*create text file for writing*
9	`w`	I	*open binary file for reading*
10	`wb`	J	*open binary file for update (reading and writing)*
11	`wbx`	K	*open text file for reading*
12	`wx`	L	*open text file for update (reading and writing)*
13	`w+`	M	*truncate to zero length or create binary file for update*
14	`w+b` *or* `wb+`	N	*truncate to zero length or create binary file for writing*
15	`w+bx` *or* `wb+x`	O	*truncate to zero length or create text file for update*
16	`w+x`	P	*truncate to zero length or create text file for writing*

► É 5.2

Dans les exemples fournis plus haut, justifiez : le choix du type renvoyé, le mode d'ouverture des fichiers, le choix des fonctions de lecture ou d'écriture, les tests de validité, l'ordre dans lequel ces tests sont effectués, les quantités de contrôle.

► **É 5.3**

Donnez corps à la fonction de prototype

```
bool fexists(const char *filename);
```

qui renvoie `true` si le fichier de nom la chaine pointée par `filename` existe et `false` sinon. Vous commencerez par exercer un regard critique sur l'énoncé en précisant ce qu'il faut comprendre ici de la notion d'« existence ».

► **É 5.4**

Donnez corps à la fonction de prototype

```
long int fnlines(const char *filename);
```

qui, en cas de succès, renvoie le nombre de lignes du fichier texte de nom la chaine pointée par `filename` et, en cas d'échec, `-1`. Vous envisagerez deux cas :

1) celui où toutes les lignes se terminent par le caractère de fin de ligne `'\n'`. Ce cas est simple à mettre en œuvre puisqu'il suffit de compter le nombre de `'\n'` figurant dans le fichier ;

2) celui, plus complexe, où il est autorisé que la dernière ligne ne soit pas terminée par le caractère de fin de ligne.

► **B 5.5**

Donnez corps à la fonction de prototype

```
int fcopy(const char *destfn, const char *srcfn);
```

qui tente de créer le fichier de nom la chaine pointée par `destfn` et de lui donner comme contenu celui du fichier de nom la chaine pointée par `srcfn`. La fonction renvoie zéro en cas de succès, une valeur non nulle en cas d'échec. Vous traiterez deux cas :

1) copie caractère par caractère ;

2) copie bloc par bloc, avec une taille de blocs égale à la macroconstante `BUFSIZ` définie dans l'en-tête `<stdio.h>` comme la taille par défaut des tampons.

► **B 5.6**

Donnez corps à la fonction de prototype

```
int ffloat_addb(const char *filename, float quant);
```

qui tente d'ajouter `quant` à chacun des composants du fichier homogène de `float` de nom la chaine pointée par `filename`. La fonction renvoie zéro en cas de succès, une valeur non nulle en cas d'échec.

► **B 5.7**

Reprenez l'exercice 4.4 en l'adaptant aux fichiers homogènes. Les paramètres passés seront des pointeurs vers les chaines qui spécifient le nom des fichiers. Les valeurs renvoyées seront codées sur le type `long int` pour les deux premières fonctions, la valeur `-1` étant renvoyée en cas d'erreur détectée en cours d'opération. Elles seront codées sur le type `int` pour les deux suivantes, `0` pour FAUX, `1` pour VRAI et `-1` en cas d'erreur.

► **MD 5.8**

Donnez corps à la fonction de prototype

```
int fscan_str(FILE *stream, char *s, size_t n);
```

qui tente de lire sur le flot `stream` les aux plus `n` premiers caractères consécutifs qui ne sont pas des caractères d'espacement (pas de la catégorie `isspace`) et de les copier à partir de l'adresse `s` en les faisant suivre de la marque de fin de chaine `'\0'`. La fonction renvoie `EOF` si la fin de fichier est atteinte ou si une erreur survient avant la lecture du premier caractère qui n'est pas un caractère d'espacement. Elle renvoie sinon `0` si une erreur de lecture ou de réinjection de caractère survient et `1` en cas de succès.

► **MD 5.9**

1) Programmez l'insertion d'un objet donné à un indice donné dans un fichier homogène. Faites de même pour la suppression du composant d'un indice donné.

2) Enchainez avec l'insertion du contenu d'un fichier à un indice donné dans un fichier, la suppression d'un nombre donné de composants à partir d'un indice donné, la suppression complémentaire de la précédente.

6 C : développements

Ce chapitre complète la présentation du langage C : aucune autre notion technique n'est plus ajoutée dans la suite de ce support.

La notion de pointeur est tout d'abord revisitée et approfondie : les pointeurs constituent certainement l'objet du langage le plus important et son traitement est original. Est ensuite abordé le problème de la portée des déclarations des objets dans un fichier source ainsi que celui de la durée de vie de ces objets durant la phase d'exécution. Une introduction à la production d'exécutables à partir de plusieurs fichiers sources est enfin présentée.

6.1 Pointeurs

Pointeurs et tableaux

Travailler sur les composants d'un tableau au moyen de pointeurs sur ces composants peut à la fois rendre le code plus lisible et plus efficace qu'en passant par des indices en référence à l'adresse du premier des composants du tableau. Le langage C introduit à cet effet une « arithmétique » des pointeurs[38]. Il est ainsi possible d'ajouter ou retrancher un entier à un pointeur pour obtenir une adresse située à un nombre de composants d'écart égal à cet entier, de soustraire un pointeur à un autre pour obtenir le nombre de composants qui les séparent et de comparer deux pointeurs pour connaitre leurs positions relatives sur le tableau.

Formellement, soient a l'identificateur d'un tableau de longueur n de composants de type T, p et q deux expressions de type pointeur de T :

— si p repère le composant d'indice k du tableau a, les expressions[39] `(p) + j`, de manière équivalente `j + (p)`, et `(p) - j` sont valides, de type pointeur de T, et repèrent respectivement les composants d'indice $k+j$ et $k-j$ de a, à supposer qu'ils existent, c'est-à-dire lorsque les valeurs de ces expressions entières sont comprises entre 0 et $n-1$;

— si p repère le dernier composant du tableau a, l'expression `(p) + 1`, de manière équivalente `1 + (p)`, est valide, de type pointeur de T, et a pour valeur l'adresse qui suit immédiatement la fin du tableau[40] ;

— symétriquement, si p a pour valeur l'adresse qui suit immédiatement la fin du tableau, l'expression `(p) - 1`, de manière équivalente `-1 + (p)`, est valide, de type pointeur de T, et repère le dernier composant du tableau ;

— toute autre expression de la forme `(p) + j`, `j + (p)` ou `(p) - j` que celles mentionnées ci-dessus est illégale ;

— si p et q, séparément, repèrent des composants du tableau a ou ont pour valeur l'adresse qui suit immédiatement la fin du tableau, autrement dit s'il existe des entiers j et k compris au sens large entre 0 et n tels que `p == a + j` et `q == a + k`, l'expression `(p) - (q)` est valide et vaut

38. Dans cette arithmétique sont définies une addition externe, une distance et une relation d'ordre.

39. Le parenthésage des expressions de type pointeur de T, systématique dans ces paragraphes s'attachant aux notations formelles, n'est pas une obligation : il ne devrait être utilisé qu'en cas de nécessité.

40. Autre expression possible pour cette adresse qui ne pointe sur aucun composant du tableau : celle après le dernier des composants du tableau, *one past the last element of the array object*.

$j - k$. Le résultat de la différence de deux pointeurs est codé sur le type entier signé **`ptrdiff_t`** défini dans l'en-tête `<stddef.h>`. La lettre `t` doit être utilisée dans les chaines de format pour le type **`ptrdiff_t`** ;

— toute autre différence de pointeurs que celles mentionnées ci-dessus est illégale ;

— si les expressions p et q, séparément, repèrent des composants du tableau a ou ont pour valeur l'adresse qui suit immédiatement la fin du tableau, elles peuvent être comparées au moyen des six opérateurs usuels : si $\lozenge$ est l'un de ces opérateurs, l'expression `(`p`) ` $\lozenge$ ` (`q`)` est valide, de type entier, et a pour valeur le résultat de l'évaluation de l'expression `(`p`) - (`q`) ` $\lozenge$ ` 0`. Autrement dit 1 (vrai) si `(`p`) - (`q`) ` $\lozenge$ ` 0`, 0 (faux) sinon ;

— toute autre comparaison d'expressions de pointeurs que celles mentionnées ci-dessus est illégale, sauf à ce que l'une d'elles soit un pointeur nul et que l'opérateur soit `==` ou `!=` ou alors que les deux expressions ne soient pas de type pointeur de T mais de type pointeur générique (voir la sous-section qui suit).

Voici deux conséquences importantes des règles qui précèdent alliées au fait que tout tableau est la juxtaposition de ses composants :

— si l'expression p de type pointeur de T repère un tableau de longueur n de composants de type T, `(`p`) + `k repère un tableau de longueur $n - k$ de composants de type T pour toute valeur entière k comprise entre 0 et n ;

— si p, pointeur de T, appartient à l'ensemble des pointeurs valides au regard de l'arithmétique des pointeurs relativement au tableau a de longueur n de composants de type T, les trois relations suivantes sont satisfaites :

`((char *) `p` - (char *) `a`) % sizeof(`T`) == 0` ;
`(char *) `p` >= (char *) `a ;
`(char *) `p` <= (char *) `a` + `n` * sizeof(`T`)`.

Les opérateurs `++`, `--`, `+=` et `-=` s'appliquent aux variables pointeurs de composants de tableau avec la même sémantique que pour les types arithmétiques.

Il faut ajouter à l'exposé qui vient d'être fait que le travail de passage d'un code utilisant des indices pour accéder aux composants des tableaux à sa version plus performante utilisant des pointeurs vers ces composants peut être confié au compilateur dans les cas simples comme les parcours séquentiels ou les parcours à rebours, qu'ils soient complets ou partiels : `gcc` l'assure dès l'option d'optimisation `-O2` ; si le source originel, avec indices, est sûr, le codage résultant en langage machine, avec pointeurs, le sera aussi [41].

Type pointeur générique

Le type *pointeur générique*[42] est noté **`void *`**. Il est spécialement destiné aux opérations *polymorphes*[43] : tout pointeur peut être converti sur le type **`void *`**, et réciproquement, sans perte d'information quant à l'adresse. Le type d'origine est par contre perdu, proscrivant de fait toute opération d'indirection ou arithmétique.

Les comparaisons d'expressions de type pointeurs génériques sont possibles avec le même sens que celles pour les pointeurs de types donnés. Si les expressions p et q repèrent, séparément, des composants d'un même tableau ou ont pour valeur l'adresse qui suit immédiatement la fin de ce tableau, l'expression `(`p`) ` $\lozenge$ ` (`q`)` est valide, de type entier, et a pour valeur le résultat de l'évaluation de l'expression **`(char *)`** `(`p`) - (char *) (`q`) ` $\lozenge$ ` 0`.

41. De nombreuses failles de sécurité sont dues à l'emploi mal compris de l'arithmétique des pointeurs.

42. Il s'agit là d'une utilisation de l'acception classique du terme générique : généraliste, à distinguer de celle précisée dans la note 18.

43. En informatique, une entité est polymorphe lorsqu'elle a la capacité de pouvoir être attachée à des objets de divers types.

Les constantes pointeur nul, dont `nullptr` et `NULL`, sont des pointeurs génériques. Lorsqu'elles sont convertie sur un type pointeur non générique, il est garanti que les valeurs résultantes diffèrent de celles de tout pointeur vers un objet.

De manière générale, trois catégories d'opérations qui ont affaire au type pointeur générique peuvent être distinguées :

— celles à qui seules les adresses des objets sont transmises et qui, en conséquence, ne peuvent que se contenter d'actions sur ces seules adresses ;

— celles pour lesquelles la transmission de la taille des objets accompagne celle des adresses : des actions sur les zones pointées peuvent alors avoir lieu ;

— celles enfin correspondant à des tableaux, avec, pour chaque tableau, la transmission de l'adresse de son premier composant, de sa longueur et de la taille commune à tous ses composants : des actions sur les contenus des composants peuvent être entreprises.

Pointeurs et fonctions

Un identificateur de fonction est une constante de type *pointeur de fonction*, de valeur l'adresse de la première instruction de la fonction. Étant assimilée à un pointeur, toute fonction peut être stockée dans une variable ou passée en argument d'une autre fonction. La forme d'une déclaration d'un pointeur de fonction est :

type (∗*identificateur*)(*liste-des-types-des-paramètres*)

la liste pouvant déclarer les identificateurs des paramètres. Le *type d'une fonction* est la liste constituée du type de retour de la fonction et des types de ses paramètres, dans l'ordre dans lequel ils figurent dans les déclarations et la définition de la fonction.

6.2 Portée, niveau, durée

Portée

La *portée* de la déclaration d'un identificateur dans un programme est la partie du programme dans laquelle l'identificateur est utilisable.

La portée débute à l'endroit où est faite la déclaration. Si cet endroit est situé dans un bloc, c'est-à-dire dans une portion de programme délimitée par une accolade ouvrante `{` et une accolade fermante `}`, elle s'étend jusqu'à la fin du bloc. Elle s'étend sinon jusqu'à la fin du programme.

Deux cas particuliers :

— les paramètres formels d'une fonction font partie du bloc de la fonction ;

— la portée d'une variable déclarée au début d'une boucle `for` est limitée à la boucle.

Niveau, localité, globalité

Le *niveau* (ou la *profondeur*) d'une déclaration est le nombre de blocs dans lequel elle est incluse. Toute nouvelle déclaration d'un identificateur masque la précédente. Une nouvelle déclaration de niveau $n \geqslant 1$ masque donc les déclarations des niveaux 0 à $n-1$.

Tout identificateur déclaré au niveau 0 est dit *global*. Tout identificateur de bloc, de niveau strictement positif donc, est dit *local* : sa portée est limitée au bloc. En C standard, les fonctions ne peuvent être déclarées ou définies qu'au niveau 0.

Durée

De manière générale, la *durée (de vie)* d'un objet dépend de son niveau :

— si l'objet est global, il est *permanent* : il existe du début à la fin de l'exécution. Un emplacement mémoire fixe lui est alloué ;

— si l'objet est local, il est *temporaire* : il est créé au début de l'exécution de la fonction qui le déclare et détruit à la fin de cette exécution. Un emplacement mémoire lui est alloué pour la durée de l'exécution sur la pile d'exécution.

6.3 Découpage et compilation séparée

Dès qu'un programme devient important, il convient de le découper en plusieurs fichiers. Ce qui peut donner lieu à la création de *modules*, des couples de fichiers sources *identificateur*`.h` et *identificateur*`.c`, le premier contenant des déclarations, le second, les définitions associées. Quoi qu'il en soit, la production de l'exécutable exige alors la compilation séparée de ces fichiers puis l'édition des liens.

Début et fin typiques d'un fichier en-tête

Afin d'éviter les inclusions multiples[44] du fichier en-tête de nom *identificateur*`.h`, sont typiquement données les directives :

```
#ifndef IDENTIFICATEUR__H fin-de-ligne
#define IDENTIFICATEUR__H fin-de-ligne
```

au début de ce fichier et :

```
#endif fin-de-ligne
```

à la fin du fichier, « *IDENTIFICATEUR* » voulant ici signifier « *identificateur* » écrit avec des lettres capitales. Cela indique au préprocesseur que si la macroconstante *IDENTIFICATEUR*`__H` est de lui inconnue (*if not defined*), la suite des lignes débutant à la ligne suivante et se terminant à la directive `#endif` doit être conservée — il doit donc commencer par définir la macroconstante. Dans le cas contraire, cette suite de lignes est écartée.

Des déclarations, une définition

Si plusieurs déclarations d'un identificateur global existent dans certains des fichiers constituant le programme, une seule d'entre elles doit le définir, les autres n'en sont que des références. Pour une fonction, les références sont les prototypes de la fonction ; quant à *la* définition, elle possède *le* bloc d'instructions.

Les références de variables globales doivent être précédées du mot-clé `extern`. Ce n'est pas une obligation pour les fonctions, mais il s'agit d'un usage courant.

Le mot-clé `static`, placé devant une définition globale, permet de limiter la portée de la définition au seul fichier dans laquelle elle figure.

Les deux spécifications de rangement que sont `extern` et `static` s'excluent mutuellement.

Commande make, fichier makefile et dépendances

La production des fichiers objets et de l'exécutable est classiquement confiée à la commande `make`, laquelle cherche dans le dossier courant le fichier `makefile` (ou `Makefile`). Ce fichier contient notamment des spécifications décrivant les dépendances des différents fichiers nécessaires à la production de l'exécutable.

Donnée sans argument, la commande `make` tente de réaliser la première des spécifications figurant dans le fichier `makefile` en parcourant le graphe des dépendances induit par les spécifications. Donnée avec arguments, la commande `make` tente de réaliser chacune des spécifications nommées.

L'option `-MM` de `gcc` produit en sortie la liste des dépendances pour les fichiers sources figurant sur la ligne de commande sans mentionner les en-têtes standards.

6.4 Exemples

Une procédure de modification de chaines, deuxième version

Suit une deuxième version de la procédure dont la spécification figure page 31 dans laquelle l'accès aux composants de la chaine est réalisé à l'aide d'un simple pointeur avancé au caractère suivant à chaque itération :

44. Une macroconstante ne peut être définie une seconde fois qu'avec la même chaine de substitution.

```
5  void str_all_stars(char *s) {
6    char *p = s;
7    // IB : 0 <= p - s && p - s <= n
8    //   && s == { '*' }^(p - s) . { c_{p - s}, ..., c_{n - 1}, '\0' }
9    // QC : p - s
10   while (*p != '\0') {
11     *p = '*';
12     ++p;
13   }
14 }
```

La procédure de tri polymorphe fournie par la bibliothèque standard

La procédure `qsort`

```
#include <stdlib.h>
void qsort(void *base, size_t nmemb, size_t size,
    int (*compar)(const void *, const void *));
```

trie un tableau dont l'adresse du premier composant est `base`, le nombre de composants (autrement dit sa longueur), `nmemb`, et la taille des composants, `size`. Le contenu du tableau est trié dans l'ordre croissant induit par la fonction de comparaison pointée par `compar`, laquelle doit renvoyer une valeur entière strictement négative, nulle ou strictement positive selon que l'objet pointé par le premier argument est strictement inférieur, égal ou strictement supérieur à celui pointé par le second argument. La procédure garantit que les arguments passés à la fonction pointée par `compar` sont des pointeurs vers des composants du tableau. Autrement dit, si `p` est l'un quelconque des arguments passés à la fonction pointée par `compar`, l'expression suivante est satisfaite :

```
((char *) p - (char *) base) % size == 0
    && (char *) p >= (char *) base
    && (char *) p < (char *) base + nmemb * size
```

Soit à trier un tableau a de longueur n de composants de type T. Si a doit être trié dans l'ordre croissant sur ce type et que le type supporte les opérateurs de comparaison usuels, il suffit de définir la fonction de comparaison nommée arbitrairement `T_compar` comme suit :

```
int T_compar(const T *ptr1, const T *ptr2) {
  return (*ptr1 > *ptr2) - (*ptr1 < *ptr2);
}
```

et d'effectuer l'appel :

```
qsort( a , n , sizeof( T ), (int (*)(const void *, const void *)) T_compar);
```

Si maintenant le type est une structure, que le tableau a doit être trié sur les champs f_1, clé primaire du tri, et f_2, clé secondaire du tri, en croissant sur les deux champs, et que les types des deux champs supportent les opérateurs de comparaison usuels, la fonction de comparaison peut prendre la forme suivante :

```
int T_compar(const T *ptr1, const T *ptr2) {
  int d = (ptr1->f1 > ptr2->f1 ) - (ptr1->f1 < ptr2->f1 );
  if (d == 0) {
    d = (ptr1->f2 > ptr2->f2 ) - (ptr1->f2 < ptr2->f2 );
  }
  return d;
}
```

Utilisation de make

Suivent :

— les listings de sept fichiers sources d'un même programme (d'intérêt nul) ;

— la liste des dépendances telle que donnée par `gcc` pour les fichiers sources `.c` ;

— le listing d'un fichier `makefile` permettant la production des fichiers objets et de l'exécutable ainsi que le nettoyage, le symbole « ⟶ » signifiant une tabulation ;

— la trace d'exécutions consécutives de la commande `make`.

Chacune des lignes de commande rapportées débute par le caractère « `$` » : il s'agit de la séquence d'invite usuelle de l'interprète de commandes `bash`.

y1.h

```
1 #ifndef Y1__H
2 #define Y1__H
3
4 extern void y1();
5
6 #endif
```

y2.h

```
1 #ifndef Y2__H
2 #define Y2__H
3
4 extern void y2();
5
6 #endif
```

z.h

```
1 #ifndef Z__H
2 #define Z__H
3
4 extern void z(int n);
5
6 #endif
```

y1.c

```
1 #include "y1.h"
2 #include "z.h"
3
4 static void y() {
5   z(1);
6 }
7
8 void y1() {
9   y();
10 }
```

z.c

```
1 #include <stdio.h>
2 #include "z.h"
3
4 static int zs = 21;
5
6 void z(int n) {
7   zs += n;
8   printf("%d", zs);
9 }
```

y2.c

```
1 #include "y2.h"
2 #include "z.h"
3
4 static void y() {
5   z(2);
6 }
7
8 void y2() {
9   y();
10 }
```

x.c

```
1 #include <stdlib.h>
2 #include "y1.h"
3 #include "y2.h"
4
5 int main() {
6   y1();
7   y2();
8   return EXIT_SUCCESS;
9 }
```

```
$ gcc -MM x.c y1.c y2.c z.c
x.o: x.c y1.h y2.h
y1.o: y1.c y1.h z.h
y2.o: y2.c y2.h z.h
z.o: z.c z.h
```

makefile

```
1 CC = gcc
2 CFLAGS = -std=c2x \
3   -Wall -Wconversion -Werror -Wextra -Wpedantic -Wwrite-strings -O2
4 LDFLAGS =
5 objects = x.o y1.o y2.o z.o
6 executable = x
7
8 all: $(executable)
9
10 clean:
11 ⟶$(RM) $(objects) $(executable)
```

```
12
13 $(executable): $(objects)
14 	$(CC) $(objects) $(LDFLAGS) -o $(executable)
15
16 x.o: x.c y1.h y2.h
17 y1.o: y1.c y1.h z.h
18 y2.o: y2.c y2.h z.h
19 z.o: z.c z.h
```

```
$ make clean
rm -f x.o y1.o y2.o z.o x
$ make y1.o
gcc -std=c2x -Wall -Wconversion -Werror -Wextra -Wpedantic -Wwrite-strings -O2
 -c -o y1.o y1.c
$ make y1.o
make: « y1.o » est à jour.
$ make
gcc -std=c2x -Wall -Wconversion -Werror -Wextra -Wpedantic -Wwrite-strings -O2
 -c -o x.o x.c
gcc -std=c2x -Wall -Wconversion -Werror -Wextra -Wpedantic -Wwrite-strings -O2
 -c -o y2.o y2.c
gcc -std=c2x -Wall -Wconversion -Werror -Wextra -Wpedantic -Wwrite-strings -O2
 -c -o z.o z.c
gcc x.o y1.o y2.o z.o  -o x
```

Sont à remarquer dans le fichier `makefile` :

— la définition de variables, « *v* `=` ... », celle de spécifications, « *s*`:` ... », l'utilisation de variables, « `$(`*v*`)` » ;

— les (re)définitions des variables prédéfinies : `CC`, nom de la commande de compilation ; `CFLAGS`, options de cette commande ; `LDFLAGS`, options pour l'éditeur de liens (ici vide) ;

— les définitions de nouvelles variables : `objects`, liste des fichiers objets du programme ; `executable`, nom du fichier exécutable à produire ;

— les identificateurs traditionnellement utilisés pour les spécifications : `all`, pour tout produire et, typiquement, première des spécifications mentionnées ; `clean`, pour supprimer tout ce qui a pu être produit par `all` ;

— les spécifications relatant une relation de dépendance ;

— les spécifications explicitant sur la ligne située au dessous, après une tabulation obligatoire, l'action à effectuer ;

— l'utilisation de la variable prédéfinie `RM`, de valeur implicite « `rm -f` », commandant la suppression de fichiers ;

— l'utilisation du caractère barre oblique inverse pour améliorer la lisibilité et poursuivre la définition d'une variable sur la ligne du dessous.

À remarquer à l'exécution de `make` :

— l'utilisation pour les spécifications d'identificateur en « `.o` » de la règle implicite

```
$(CC) $(CPPFLAGS) $(CFLAGS) -c
```

`CPPFLAGS` étant une variable prédéfinie signifiant les options passées au préprocesseur et vide par défaut, l'option `-c` signifiant qu'il ne doit pas être fait appel à l'éditeur de lien ;

— l'exécution d'un minimum d'actions, dont le calcul est basé sur la date et l'heure de mise à jour des fichiers, sources comme objets.

Exercices

▶ É 6.1

Soit `a` un tableau de longueur `n` dont les composants sont d'un type nommé `element`.

1) Exprimez l'adresse du tableau sur chacun des types `element *`, `void *` et `char *`.

2) Même chose pour l'adresse qui suit immédiatement la fin du tableau.

3) Même chose pour l'adresse du composant d'indice `k` du tableau.

4) Soit `p` l'adresse d'un composant du tableau. Quel est l'indice du composant lorsque `p` est de type `element *` ? de type `void *` ? de type `char *` ?

▶ É 6.2

Donnez corps à la fonction de prototype

```
int fcopy_charcond(const char *destfn, const char *srcfn, int (*charcond)(int));
```

qui tente de copier vers le fichier de nom la chaine pointée par `destfn` les caractères du fichier de nom la chaine pointée par `srcfn` qui satisfont la condition `charcond` (le paramètre `charcond` pourrait par exemple être l'une des fonctions de la famille `is...` de l'en-tête standard `<ctype.h>`). La fonction renvoie zéro en cas de succès, une valeur non nulle en cas d'échec.

▶ É 6.3

À l'aide de la procédure `qsort`, faites trier un tableau de `fellow` (voir exercice 4.3) :

1) dans l'ordre croissant des dates de naissance ;

2) dans l'ordre croissant des noms, clé primaire, et des prénoms, clé secondaire, en utilisant la fonction `strcmp`.

▶ É 6.4

Dans le problème qui consiste à trier dans l'ordre croissant un tableau a de longueur n sur un type T supportant les opérateurs de comparaison usuels à l'aide de la procédure `qsort`, on lit souvent dans la mauvaise littérature que le corps de la fonction de comparaison T`_compar` se résume à `return *ptr1 - *ptr2;`. Quels sont les seuls types de base pour lesquels cela est vrai ?

▶ É 6.5

Considérez le programme suivant :

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  static int n = 7;
5
6  static void proc1() {
7    ++n;
8    printf("%d", n);
9  }
10
11 static void proc2(int *n) {
12   *n += 5;
13   printf("%d", *n);
14 }
15
16 int main() {
17   int n = 0;
18   proc1();
```

```
19   printf("%d", n);
20   proc2(&n);
21   {
22     int n = 1;
23     proc2(&n);
24     printf("%d", n);
25   }
26   proc2(&n);
27   proc1();
28   return EXIT_SUCCESS;
29 }
```

1) À quelle déclaration de `n` se rapporte chacun de ses usages ?
2) Quel est le niveau de chacune des déclarations de `n` ?
3) Qu'affiche le programme ?

► **B 6.6**

Donnez corps aux fonctions standards suivantes avec, pour chacune d'elles, au moins une version à variable de contrôle de boucle du type pointeur de caractère :

```
#include <string.h>
int strcmp(const char *s1, const char *s2);
int strncmp(const char *s1, const char *s2, size_t n);
size_t strlen(const char *s);
```

— `strcmp` compare les chaines de caractères pointées par `s1` et `s2` et renvoie une valeur strictement négative, nulle ou strictement positive selon que la première chaine est strictement inférieure, égale ou strictement supérieure à la seconde pour l'ordre lexicographique. Pour ce faire, la fonction parcourt en parallèle les deux chaines. Si leurs caractères sont identiques jusqu'à la marque de fin de chaine `'\0'`, elle renvoie zéro. Dans le cas contraire, le parcours prend fin sur un couple de caractères (c_1, c_2) tel que $c_1 \neq c_2$; la fonction renvoie alors un entier strictement négatif si la valeur *non signée* de c_1 est inférieure à celle de c_2, et un entier strictement positif sinon ;
— même chose pour `strncmp` que pour `strcmp`, mais limité à `n` couples de caractères ;
— `strlen` renvoie la longueur de la chaine de caractères pointée par `s`.

► **MD 6.7**

Passez au polymorphisme en y plongeant les exercices 4.4 et 5.7. Pour les prototypes, reprenez les précédents : dans le cas des tableaux, remplacez les occurrences de `const element *` par `const void *` ; dans les deux cas, ajoutez

```
, size_t size, int (*compar)(const void *, const void *)
```

en fin de liste des paramètres.

7 Analyse des algorithmes

L'activité qui consiste à chercher des critères de performance pour des algorithmes, à en exprimer les couts et à comparer ces couts est appelée *analyse d'algorithmes*.

7.1 Complexité

Définitions : complexités temporelle et spatiale

Par *complexités d'un algorithme*, on entend :
— sa *complexité temporelle* (ou *en temps*) : son temps de calcul ;
— sa *complexité spatiale* (ou *en espace*) : l'espace mémoire requis par le calcul.

Définitions : complexités pratique et théorique

— La *complexité pratique* est une mesure précise des complexités temporelles et spatiales pour un *modèle de machine donné*.

— La *complexité (théorique)* est un *ordre de grandeur* de ces couts, exprimé de manière la plus *indépendante* possible des conditions pratiques d'exécution.

7.2 Opérations représentatives et taille des données

Pour quantité de problèmes, on peut mettre en évidence une ou plusieurs opérations que l'on qualifie de *représentatives* au sens où la complexité, temporelle ou spatiale, d'un algorithme solution du problème étudié s'exprime en fonction du nombre de ces opérations. Il peut s'agir, selon les cas, d'une opération arithmétique particulière, d'une opération de comparaison, de transfert ou d'échange d'éléments d'un type donné, d'une opération de contrôle du flux d'instructions (branchements conditionnel et inconditionnel, appels, retour à l'appelant).

Il est nécessaire d'adopter simultanément une mesure de la *taille des données* du problème. Selon les cas, il peut s'agir de la longueur d'un tableau ou d'un fichier, de la longueur de la décomposition d'un nombre dans une base particulière, de deux nombres, le premier spécifiant un nombre de lignes, le second, un nombre de colonnes, etc.

Le dénombrement de telles opérations en fonction de la taille des données permet d'en déduire la complexité sans avoir à passer par les associations opération-temps ou opération-espace et de comparer les performances d'algorithmes solutions d'un même problème.

7.3 Cas extrêmes et cas moyen

Définitions : complexités maximale, minimale et moyenne

Trois mesures de la complexité (temporelle ou spatiale), fonction de la taille n des données, sont généralement considérées dans les analyses d'algorithmes :

— la *complexité maximale*, obtenue dans le *pire des cas*, pour des données de taille n pour lesquelles la mesure est maximale ;

— la *complexité minimale*, obtenue dans le *meilleur des cas*, pour des données de taille n pour lesquelles la mesure est minimale ;

— la *complexité moyenne*, moyenne calculée sur l'ensemble de données de taille n. Il s'agit du *cas moyen*, lequel n'a de sens que si des *hypothèses probabilistes* sur la répartition des données peuvent être faites.

7.4 Notations asymptotiques

Dans la suite, on entend par « fonction positive » une fonction de l'ensemble des naturels dans l'ensemble des réels positifs.

Définitions : notations asymptotiques[45] *des ordres de grandeur*

Soit g une fonction positive :

— $O(g)$ est l'ensemble des fonctions positives f pour lesquelles il existe une constante positive α et un rang n_0 tels que, pour tout $n \geqslant n_0$, $f(n) \leqslant \alpha g(n)$;

— $\Omega(g)$ est l'ensemble des fonctions positives f pour lesquelles il existe une constante positive α et un rang n_0 tels que, pour tout $n \geqslant n_0$, $f(n) \geqslant \alpha g(n)$;

— $\Theta(g) = O(g) \cap \Omega(g)$.

45. La notation O a été introduite en 1894 par le mathématicien allemand Paul Bachman (1837-1920). Elle a été popularisée par la suite, sous la forme « $= O$ », par son compatriote Edmund Landau (1877-1938), avec une confusion entre égalité et inclusion. La notation Ω a été introduite en 1914 par les mathématiciens britanniques Godfrey Harold Hardy (1877-1947) et John Edensor Littlewood (1885-1977). La notation Θ date de 1976 ; elle est due à l'informaticien états-unien Donald Erwin Knuth.

La dernière définition peut s'exprimer autrement : $\Theta(g)$ est l'ensemble des fonctions positives f pour lesquelles il existe deux constantes positives α et α' ainsi qu'un rang n_0 tels que, pour tout $n \geqslant n_0$, $\alpha g(n) \leqslant f(n) \leqslant \alpha' g(n)$.

O, Ω et Θ se prononcent « grand oh », « grand oméga » et « grand thêta ». Ces notations se généralisent au cas de fonctions de plusieurs variables.

Par commodité, on emploie le plus souvent les expressions « image » dans les notations, « $f(n) \in X(g(n))$ » où X vaut O, Ω ou Θ, plutôt que les seuls symboles des fonctions, « $f \in X(g)$ ». Par commodité toujours, on écrit ou dit souvent « est (en) » plutôt que « $\in$ » ou « appartient à ».

Définitions : désignations des classes de complexités courantes

À chaque classe de complexité Θ est associée sa désignation courante :

notation	*désignation*	*notation*	*désignation*
$\Theta(1)$	constante	$\Theta(n^2)$	quadratique
$\Theta(\ln n)$	logarithmique	$\Theta(n^3)$	cubique
$\Theta(\sqrt{n})$	racinaire	$\Theta(n^k)$, $k \in \mathbb{N}, k \geqslant 2$	polynomiale
$\Theta(n)$	linéaire	$\Theta(a^n)$, $a > 1$	exponentielle
$\Theta(n \ln n)$	quasi-linéaire	$\Theta(n!)$	factorielle

Pour les classes O et Ω, les désignations sont précédées de « au plus » et « au moins ».

7.5 Exemples

Problème des occurrences répétées

Le problème des occurrences répétées consiste à déterminer si un mot arbitraire contient au moins deux fois la même lettre ou non.

Sans autre précision, l'opération représentative est la comparaison de (deux) lettres, « égal à » ou « différent de », et la taille des données, la longueur n du mot.

Maintenant, on peut remarquer qu'il suffit de parcourir le mot à partir de sa deuxième lettre à la recherche d'une lettre identique à sa première lettre, puis, si la recherche n'est pas fructueuse, à partir de sa troisième à la recherche d'une lettre identique à sa deuxième, etc., jusqu'à partir de la dernière lettre. Si, lors d'un parcours, la lettre cherchée est trouvée, l'algorithme prend fin immédiatement et fournit un résultat positif. Dans le cas contraire, c'est un résultat négatif qui doit être fourni.

Dans le meilleur des cas et pour $n \geqslant 2$, une seule comparaison est effectuée : les deux premières lettres sont identiques. Dans le pire des cas, $n-1$ comparaisons sont effectuées lors du premier parcours, $n-2$ lors du deuxième, etc., 1 pour le $(n-1)$-ième et 0 pour le dernier ; soit au total $n(n-1)/2$ comparaisons.

Il vient au final que l'algorithme qui vient d'être esquissé, puis, que le problème lui-même, a une complexité temporelle à la fois $\Omega(1)$ et $O(n^2)$.

Tyrannie de la complexité

Aucun progrès technologique (modèle à un seul processeur) ne permet à un algorithme de changer de classe de complexité. C'est ce qu'illustre le tableau suivant, où sont indiqués les effets de la multiplication de la puissance d'une machine par 10, 100 et 1 000, vitesse ou mémoire, quant à la taille maximale N des problèmes que peuvent traiter des algorithmes de complexité donnée, temps ou espace imparti :

complexité	$\times 10$	$\times 10^2$	$\times 10^3$
$\Theta(\ln n)$	N^{10}	N^{10^2}	N^{10^3}
$\Theta(\sqrt{n})$	$10^2\,N$	$10^4\,N$	$10^6\,N$
$\Theta(n)$	$10\,N$	$10^2\,N$	$10^3\,N$

complexité	$\times 10$	$\times 10^2$	$\times 10^3$
$\Theta(n^2)$	$\simeq 3\,N$	$10\,N$	$\simeq 32\,N$
$\Theta(n^3)$	$\simeq 2\,N$	$\simeq 5\,N$	$10\,N$
$\Theta(2^n)$	$\simeq N+3$	$\simeq N+7$	$\simeq N+10$

Exercices

▶ **É 7.1**

1) Qu'est-ce que la complexité temporelle d'un algorithme ? sa complexité spatiale ? sa complexité pratique ? sa complexité théorique ?

2) Qu'est-ce que la complexité maximale ? minimale ? moyenne ?

3) Que signifie-t-on lorsque l'on écrit $f \in O(g)$? $f \in \Omega(g)$? $f \in \Theta(g)$?

4) Que signifie-t-on d'une complexité si on la qualifie de constante ? logarithmique ? au moins linéaire ? au plus quasi-linéaire ? quadratique ?

▶ **É 7.2**

Validez, invalidez ou amendez les propositions suivantes :

1) $10^3\, n^2 \in O(10^{-5}\, n^3)$;

2) $144\, n^4 - 236\, n^3 + 100\, n^2 \in O(n^4)$;

3) $2^{n+64} \in O(2^n)$.

▶ **É 7.3**

Montrez que H_n est $\Theta(\ln n)$ et que $\ln n!$ est $\Theta(n \ln n)$.

▶ **É 7.4**

Donnez, si cela est possible, les relations d'inclusion des ensembles suivants :

1) $O(\ln n)$, $O(n^3)$, $O(n)$, $O(1)$, $O(2^n)$, $O(n^2)$, $O(\sqrt{n})$, $O(n \ln n)$.

2) Même chose, mais en remplaçant O par Ω.

3) Même chose, avec cette fois Θ.

▶ **É 7.5**

Trois algorithmes répondent à un même problème pour une donnée de taille n. Pour essayer de les départager, on mesure un certain cout. Là où le premier titre $\lfloor n^2/4 \rfloor$, le deuxième affiche $2n\lfloor \mathrm{lb}(n+1) \rfloor + n$ et le dernier $25n$. Quel est le plus performant des trois ?

▶ **B 7.6**

1) Exprimez sous la forme d'un Θ l'ordre de grandeur de $\sum_{k=1}^{n} \lfloor \mathrm{lb}\, k \rfloor$.

2) Même chose pour $\sum_{k=1}^{n} \lceil \mathrm{lb}\, k \rceil$.

▶ **B 7.7**

Un nombre *parfait* est un naturel égal à la somme de ses diviseurs stricts (les naturels qui le divisent et qui lui sont strictement inférieurs). Deux nombres *amicaux* sont deux naturels tels que la somme des diviseurs stricts de l'un égale l'autre.

1) Montrez que le problème de savoir si un naturel n est parfait ou non possède une solution triviale de complexité $\Theta(n)$.

2) Montrez qu'il en possède une autre, un peu moins triviale, de complexité $\Theta(\sqrt{n})$.

3) Déduisez-en que le problème de savoir si deux naturels m et n sont amicaux ou non peut se résoudre avec une complexité $O(\sqrt{\max\{m, n\}})$.

▶ **D 7.8**

1) Sauf à ce que vous ne l'ayez déjà fait dans l'exercice 7.6 (ce qui n'était pas nécessaire), donnez de l'expression $\sum_{k=1}^{n} \lfloor \mathrm{lb}\, k \rfloor$ une forme close en la débarrassant du symbole somme.

2) Même chose pour $\sum_{k=1}^{n} \lceil \mathrm{lb}\, k \rceil$.

▶ **D 7.9**

Supposons que l'opération « dire l'élément d'un ensemble » coute un temps constant et que le seul cout retenu pour soit celui-là. Il vient dès lors immédiatement que le problème de « dire » tous les éléments de chacune des 2^n parties d'un ensemble à n éléments possède un cout à la fois $\Omega(2^n)$ et $O(n2^n)$. Montrez que ce cout est en fait $\Theta(n2^n)$.

Partie II. Sélection d'algorithmes fondamentaux

Les algorithmes considèrent classiquement les types entier et flottant. Il peuvent considérer tout autre type élémentaire : des éléments d'un ensemble quelconque ou aux propriétés précisées. Il peuvent également considérer le type mot sur ces types élémentaires.

Les mots sont un *modèle*. Il leur correspond — par exemple et pour ce qui peut être mis en œuvre ici — des implantations sous la forme de tableaux accompagnés d'une longueur constante ou variable ou de fichiers homogènes, parcourus séquentiellement, à rebours ou dans un autre ordre encore, en totalité ou partiellement.

Pour à la fois simplifier et coller au langage C, les algorithmes sont exprimés au sein d'un modèle dans lequel les expressions booléennes sont évaluées de gauche à droite de manière paresseuse.

8 Algorithmes numériques

9 Algorithmes de recherche

10 Algorithmes de tri

LA SUITE N'EST PAS ENCORE À JOUR