

Chapitre 2

MÉMOIRE

Notions	Commentaires
Représentation des flottants. Problèmes de précision des calculs flottants.	On illustre l'impact de la représentation par des exemples de divergence entre le calcul théorique d'un algorithme et les valeurs calculées par un programme. Les comparaisons entre flottants prennent en compte la précision.


Extrait de la section 1.1 du programme officiel de MP2I : « Algorithmes et programmes ».

Notions	Commentaires
Utilisation de la pile et du tas par un programme compilé.	On présente l'allocation des variables globales, le bloc d'activation d'un appel.
Notion de portée syntaxique et durée de vie d'une variable. Allocation des variables locales et paramètres sur la pile.	On indique la répartition selon la nature des variables : globales, locales, paramètres.
Allocation dynamique.	On présente les notions en lien avec le langage C : malloc et free [...]

Extrait de la section 5.1 du programme officiel de MP2I : « Gestion de la mémoire d'un programme ».

SOMMAIRE

0. Représentation des types de base	30
0. Notion de données	30
1. Booléens	30
2. Différents types d'entiers	31
<i>Entiers non-signés (p. 31). Caractères (p. 33). Entiers signés (p. 33). Dépassements de capacité (p. 35).</i>	
3. Nombres à virgule (flottante)	36
<i>Notation mantisse-exposant pour l'écriture scientifique (p. 36). Erreurs d'approximation et autres limites (p. 37).</i>	
4. Tableaux	38
5. Chaînes de caractères	39
1. Compléments très brefs de programmation.....	40
0. Notion d'adresse mémoire	40
1. Portée syntaxique et durée de vie	40
<i>Définition et premiers exemples (p. 40). Exemples avancés (p. 43).</i>	
2. Organisation de la mémoire d'un programme	44
0. Bloc d'activation	44
1. Organisation de la mémoire virtuelle d'un programme	45
<i>Pile mémoire (p. 46). Tas mémoire (p. 46). À savoir faire (p. 47).</i>	

 Ce chapitre, surtout sa dernière section, est rempli de simplifications.

0 Représentation des types de base

Dans cette section, nous allons voir comment les données (entiers, nombres à virgule, etc) sont représentées en mémoire. Nous n'allons *pas* voir où ces représentations sont rangées en mémoire.

Ce qui est vu dans cette section est valable en C et en OCaml.

0.0 Notion de données

Définition 1 (Donnée informatique).

Une donnée informatique est un élément fini d'information que l'on peut lire, stocker transmettre.

Définition 2.

- L'**unité élémentaire** de stockage des données est le **chiffre binaire (bit en anglais)**. Il peut prendre deux valeurs : 0 et 1.
- Un groupe de 8 bits est appelé un **octet**. Les ordinateurs modernes ne stockent pas les bits uns par uns dans la mémoire mais des octets : on dit que c'est le **plus petit adressable (byte en anglais)**.
- Les processeurs modernes, pour aller plus vite, ne font pas leurs calculs sur des octets mais sur un des groupes plus gros. On parle de **mot machine**, et ils font aujourd'hui souvent 8 octets / 64 bits

Remarque. Le terme du « plus petit adressable » provient du fait que puisque les bits sont stockés 8 par 8, on ne peut pas accéder à un des bits de l'octet sans avoir lu en même temps les autres. C'est une question de câblage électronique : les « fils » ne ciblent pas chaque bit individuellement mais des groupes de 8 bits adjacents.

Remarque. Des (suites d')octets en eux-mêmes ne veulent rien dire. Il faut connaître la façon de les déchiffrer. C'est le rôle d'un **type de données** : c'est une description de comment encoder et décoder des données d'une certaine nature. Vous connaissez déjà des types : `int` , `bool` .

0.1 Booléens

Le type booléen permet de stocker les deux valeurs logiques : Vrai (« true ») et Faux (« false »).

Proposition 3 (Encodage des booléens).

Il suffit en théorie d'1 bit pour encoder un booléen.

En C, on utilise un plus petit adressable (1 octet).

En OCaml, un mot machine (8 octets).

Remarque.

- En OCaml, *tout* est manipulé à travers un mot machine, dont les booléens.
- En C comme en OCaml, il existe des astuces pour stocker un booléen dans chacun des bits d'un octet / mot machine.

Définition 4 (Opérateurs logiques).

Voici les opérations usuelles des booléens :

x	false	true
Non(x)	true	false

(a) NON(x)

x \ y	false	true
false	false	false
true	false	true

(b) ET(x,y)

x \ y	false	true
false	false	true
true	true	true

(c) OU(x,y)

x \ y	false	true
false	false	true
true	true	false

(d) XOR(x,y)

FIGURE 2.1 – Tables (à double entrée sauf pour la négation) des opérations logiques usuelles.

Pour permettre d'alléger les écritures de formules logiques, on se donne des règles de priorité : NON est plus prioritaire que ET qui est plus prioritaire que XOR qui est plus prioritaire que OU.

Exemple. « NON false ET false OU true » s'évalue à true, car cette expression se parenthèse ainsi d'après les règles : « ((NON false) ET false) OU true »

Exercice. Évaluez NON (NON false ET NON true OU NON false ET true).

0.2 Différents types d'entiers

Il faut distinguer deux grandes familles de types d'entiers :

- les **entiers non-signés** sont des types qui permettent de manipuler uniquement des entiers positifs.
- les **entiers signés** sont des types qui permettent de manipuler des entiers relatifs.

0.2.0 Entiers non-signés

Théorème 5 (Écrire en base b).

Soit b un entier supérieur ou égal à 2 appelé **base**. Pour tout $x \in \mathbb{N}$, il existe une unique décomposition de x de la forme :

$$x = \sum_{i=0}^{N-1} c_i \cdot b^i$$

qui vérifie :

- pour tout i , $c_i \in \llbracket 0; b \rrbracket$. Les c_i sont appelés les **chiffres**.
- $a_{N-1} \neq 0$.

On appelle la donnée des c_i l'**écriture en base b** de x . Pour indiquer la base, on note généralement : ${}_b \overline{c_{N-1}c_{N-2}\dots c_1c_0}$

On dit que c_{N-1} est le **chiffre** de poids fort et que c_0 est le **chiffre de poids faible**.

Convention 6 (Base 10 implicite).

Lorsque l'on ne précise pas la base, on utilise implicitement la base 10.

Remarque.

- On parle de **binaire** pour la base 2, **décimal** pour la base 10, **hexadécimal** pour la base 16.
- En base 16, les chiffres peuvent valoir plus que 9 : on note a le chiffre correspondant au nombre décimal 10, b pour 12, ..., f pour 16.
- L'écriture $\overline{c_{N-1}c_{N-2}\dots c_1c_0}^b$ existe aussi.
- Vous êtes habitué-es à l'écriture en base 10 : $2048 = 2 \cdot 10^3 + 0 \cdot 10^2 + 4 \cdot 10^1 + 8 \cdot 10^0$.

Exemple.

$$\begin{aligned} {}^2\overline{1100\ 0010} &= 1 \cdot 2^7 + 1 \cdot 2^6 + 0 \cdot 2^5 + 0 \cdot 2^4 + 0 \cdot 2^3 + 0 \cdot 2^2 + 1 \cdot 2^1 + 0 \cdot 2^0 \\ &= 128 + 64 + 2 \\ &= 194 \end{aligned}$$

Définition 7 (Représentation des entiers non-signés).

Pour représenter un entier non-signé, on stocke son écriture en base b dans un ou plusieurs octets. le bit de poids faible est tout à droite. Au besoin, on complète à gauche par des 0 :

1	1	0	0	0	0	1	0
---	---	---	---	---	---	---	---

Le nombre d'octets utilisés peut être fixé par le type (4 octets pour `unsigned int` en C, 8 octets en OCaml).

Exemple. Voici l'entier 3452 représenté sur deux octets :

0	0	0	0	1	1	0	1	0	1	1	1	1	1	0	0
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

Remarque.

- Les 32 bits du type `unsigned int` C peuvent dépendre de la machine, mais 32 est la valeur la plus commune.
- En incluant `stdint.h`, on a accès en C à des types dont le nombre de bits est garanti : `uint8_t`, `uint16_t`, `uint32_t`, `uint64_t`.

Proposition 8 (Capacité des types non-signés).

Sur k bits, on peut représenter de cette façon exactement les entiers de $\llbracket 0; 2^k \rrbracket$.

⚠ Conséquence importante : on ne peut pas représenter *tous* les entiers positifs en C ni en OCaml!

Remarque. C'est long d'écrire en binaire. Pour raccourcir, on utilise souvent la base 16. En effet, 4 chiffres consécutifs en base 4 correspondent à 1 chiffre en base 16 : la traduction est donc facile¹. Convertissons par exemple ${}^2\overline{0000\ 1101\ 0111\ 1100}$

$$\begin{array}{cccc} \overline{0000} & \overline{1101} & \overline{0111} & \overline{1100} \\ \text{0} & \text{d} & \text{7} & \text{c} \end{array}$$

$$\text{Donc } {}^2\overline{0000\ 1101\ 0111\ 1100} = {}^{16}\overline{0d7c}.$$

Exercice. Réciproquement, convertissez ${}^{16}\overline{9c}$ en binaire.

1. Beaucoup plus qu'entre la base 10 et la base 2 (ou la base 16).

0.2.1 Caractères

Le type des **caractères** permet de stocker des « lettres » : une lettre de l'alphabet, un chiffre décimal, une espace, etc. Dans l'idée, il permet de stocker « une touche du clavier ».

Attention, je parle bien de lettres uniques et non de suites de lettres : ce second cas est plus compliqué, et on parle pour lui de chaînes de caractères (strings).

Définition 9 (Représentation des caractères).

Pour représenter un caractère, on le fait correspondre à un entier non-signé et on encode ensuite cet entier non-signé.

La façon la plus commune actuellement de réaliser cette correspondance est la table ASCII. Elle donne une correspondance entre 128 caractères et les 128 entiers 7-bits.

En C, le type `char` est le type des correspondances des caractères ASCII. C'est un type d'entiers non-signés, avec lequel on peut faire des calculs.

Remarque.

- ASCII signifie **A**merican **S**tandard **C**ode for **I**nformation **I**nterchange. En conséquence, les caractères qui n'intéressent pas les américains (caractères accentués par exemple) n'apparaissent pas dedans.
- Aujourd'hui, on utilise beaucoup plus de caractères que l'ASCII : nos caractères sont maintenant encodés par l'Unicode (UTF-8), qui donne des correspondances entre *beaucoup* plus de caractères et plus d'octets.
- Dans la table ASCII, les lettres de l'alphabet sont les unes après les autres (les majuscules vont de 'A'=65 à 'Z'=90, et les minuscules de 'a'=97 à 'z'=122).

Exemple. Comme les `char` sont des entiers et que les lettres se suivent, on peut écrire :

```
1 char lettre = 'a';
2 while (lettre <= 'z') {
3     fairedetrucs;
4
5     // passer à la lettre suivante
6     lettre = lettre +1;
7 }
```



0.2.2 Entiers signés

Pour représenter des entiers signés, on utilise le **complément à 2**. Pour simplifier les explications, commençons par présenter le **complément à 10**, et je noterai entre guillemets les codes (la façon de représenter les nombres) :

- 0) On fixe le nombre de chiffres que la représentation utilisera. Dans cet exemple, je travaillerai avec 3 chiffres décimaux.
- 1) La première moitié des codes possibles encodent les positifs de manière transparente :

Code	Entier représenté	Code	Entier représenté
"000"	0
"001"	1	"497"	497
"002"	2	"498"	498
...	...	"499"	499

- 2) La seconde moitié des codes encode les nombres négatifs. Pour encoder $-|x|$, on calcule $10^3 - |x|$ et on utilise le résultat comme code :

Code	Entier représenté	Code	Entier représenté
"500"	-500
"501"	-499	"997"	-3
"503"	-498	"998"	-2
...	...	"999"	-1

3) Fin.

Remarque.

- En complément à 10 sur 3 chiffres, on peut donc représenter $\llbracket -500; 500 \rrbracket$. Plus généralement, en complément à 10 sur k chiffres on peut représenter $\llbracket -10^k/2; 10^k/2 \rrbracket$.
- Le premier chiffre n'est pas un chiffre de signe au sens où il ne stocke pas le signe. Par contre, on peut en déduire le signe.
- Si l'on dispose d'un code "xyz", pour savoir s'il encode un entier positif ou strictement négatif il suffit de savoir si ce code fait partie de la première moitié des codes ou de la seconde... et donc il suffit de regarder le premier chiffre !

Et le complément à 2 ? C'est pareil mais en base 2 ! Par exemple sur 1 octet, on représente :

Code	Entier représenté	Code	Entier représenté
"0000 0000"	0	"1000 0000"	-128
"0000 0001"	1	"1000 0001"	-127
...
"0111 1110"	126	"1111 1110"	-2
"0111 1111"	127	"1111 1111"	-1

On représente souvent les compléments sous forme visuelle :

(a) Complément à 10 sur 3 chiffres

(b) Complément à 2 sur 8 chiffres

FIGURE 2.2 – Représentation visuelle des compléments

0.2.3 Dépassements de capacité

Sur des types utilisant un nombre fixé d'octets (comme en C ou en OCaml²), on ne peut donc utiliser qu'un nombre fini d'octets. Mais que se passe-t-il si, par exemple, on additionne deux nombres du type et que le résultat de l'addition est trop grand pour le type ?

Par exemple (avec des `uint8_t`), on peut poser l'addition :

$$\begin{array}{r} 1101 \quad 0011 \\ + \quad 0011 \quad 0101 \\ \hline 1 \quad 0100 \quad 1000 \end{array}$$

Le résultat de cette addition ne tient pas sur un octet. Sur la plupart des machines et des langages (dont C et OCaml), le résultat serait simplement tronqué en « enlevant ce qui déborde » et on obtiendrait ²0100 1000.

Définition 10 (Dépassement de capacité).

On parle de **dépassement de capacité (overflow)** lorsque le résultat attendu d'une opération sort de l'ensemble des valeurs représentables par son type.

On parle parfois de **surpassement** pour qualifier un dépassement « par le haut » et de **sous-passement** « par le bas ».

Proposition 11 (Gestion des dépassements entiers en C/OCaml).

Dans les deux langages au programme, les dépassements d'entiers sont gérés ainsi :

- types d'entiers non-signés : sur un type d'entiers k bits, les calculs sont effectués modulo 2^k . Cela revient à « ignorer ce qui déborde ».
- types d'entiers signés : les *encodages* sont calculés en « ignorant ce qui déborde ». Cela donne un caractère circulaire au type (cf TD).

NB : en C, cette façon de gérer n'est pas obligatoire. Elle est là sur la plupart des machines mais il est dangereux de faire un programme qui repose dessus.

⚠ Les dépassements de capacité sont une source **très** commune d'erreur. Il faut y prêter attention !

Exemple. Voici une liste d'exemples de bogues de dépassements de capacité³ :

- Explosion du vol 501 de la fusée Ariane 5, le 4 juin 1995 (à cause d'un écrêtage dans une conversion 64bits -> 16 bits et d'un mauvais choix dans le code de comment réagir à une erreur imprévue) : https://fr.wikipedia.org/wiki/Vol_501_d%27Ariane_5#Conclusions.
- La machine de radiothérapie Therac-25 a causé la mort de cinq patients par violent surdosage de radiation. Parmi les causes, une variable entière qui débordait et passait sous le seuil qui déclenchait des tests. Il y a beaucoup d'autres causes, dont et surtout la façon dont le logiciel a été conçu, relu, documenté, etc : <https://fr.wikipedia.org/wiki/Therac-25#Causes>.
- Bug de l'an 2038 : dans un ordinateur, la date est stockée en nombre de secondes depuis le 1er janvier 1970. Sur les machines où cette date est stockée en complément à 2 sur 32bits (comme `int` en C...), il y aura un débordement le 19 janvier 2038 à 3h14min8s. La date deviendra alors (débordement « circulaire ») le 13 décembre 1901, 20h45 et 52s. Cf https://fr.wikipedia.org/wiki/Bug_de_l%27an_2038.
Notez que la norme demande d'utiliser un entier non signé, ce qui sur 32 bits repousse le bug à 2106.
- Et beaucoup d'autres.

2. Mais pas comme en Python

3. En réalité, un bogue n'est presque jamais uniquement dû à un dépassement de capacité. Il se combine avec d'autres manquements, que ce soit dans le code (une fonction critique ne s'assure pas du respect de ses pré-conditions), dans le processus de validation du code (le code n'a pas été assez bien testé, ni assez prouvé, ni assez bien relu) et/ou dans la gestion de l'équipe de développement.

0.3 Nombres à virgule (flottante)

On veut représenter des nombres à virgule. Remarquons tout d'abord que dans tout intervalle non-vide, il y a une quantité infinie de nombres rationnels (sans parler des réels). On ne pourra donc pas représenter tous les nombres d'un intervalle, aussi "petit" soit-il.

0.3.0 Notation mantisse-exposant pour l'écriture scientifique

Commençons à nouveau par expliquer en base 10. Un nombre $x \in \mathbb{R}$ est dit **nombre décimal** s'il existe un $p \in \mathbb{N}$ tel que $x * 10^p \in \mathbb{Z}$. Autrement dit, si il a un nombre fini de chiffres après la virgule.

Exemple.

- Tous les entiers sont décimaux.
- 0.1 est décimal car $0.1 * 10^1 = 1 \in \mathbb{Z}$.
- 0.5 est décimal car $0.5 * 10^1 = 5 \in \mathbb{Z}$.
- $\frac{1}{3}$ n'est pas décimal même si il est rationnel.
- $\sqrt{2}$ n'est pas décimal même si il est algébrique (racine d'un polynôme à coefficients entiers).
- π et $\exp(1)$ ne sont pas décimaux.

Définition 12 (Écriture scientifique en base 10).

Pour tout x décimal, il existe une unique écriture de x de la forme $x = \pm m * 10^e$ avec :

- $m \in [1; 10[$ (et est décimal).
- $e \in \mathbb{Z}$.

On appelle m la mantisse et e l'exposant.

On parle parfois de **virgule flottante** pour désigner le fait qu'on « déplace » la virgule lorsqu'on passe un nombre en écriture scientifique.

Exemple.

- $243 = 2.43 * 10^2$
- $0.00759 = 7.59 * 10^{-2}$

Pour représenter un nombre décimal, on peut donc donner : le signe, la mantisse et l'exposant.

Revenons à la base 2. On va appliquer exactement cette idée : écrire en écriture scientifique, et communiquer mantisse, signe, exposant.

Définition 13 (Nombre dyadique).

Un nombre réel x est dit dyadique s'il existe un $p \in \mathbb{Z}$ tel que $x * 2^p \in \mathbb{Z}$. Autrement dit, s'il a un nombre fini de chiffres après la virgule en base 2.

Exemple.

- Tous les entiers sont dyadiques.
- 0.1 n'est **pas** dyadique.⁴
- 0.5 est dyadique car c'est 2^{-1} .
- $\frac{1}{3}, \sqrt{2}, \pi, \exp(1)$ ne sont pas dyadiques.

4. Le prouver est un petit exo de maths sympa. On peut par contre prouver que si un nombre est dyadique alors il est décimal : c'est la deuxième partie d'un petit exo sympa.

Définition 14 (Écriture scientifique en base 2).

Pour tout x dyadique, il existe une unique écriture de x de la forme $x = \pm m * 2^e$ avec :

- $m \in [1; 2[$ (et est dyadique).
- $e \in \mathbb{Z}$.

On appelle m la mantisse et e l'exposant.

L'idée est alors la suivante. On va stocker des nombres dyadiques sur 64 bits, ainsi :

- Le premier bit est un bit de signe 0 pour positif, 1 pour négatif.
- Ensuite l'exposant. On le stockera sous une forme particulière qui simplifie certains calculs.
- Enfin la mantisse. Toutefois, il est inutile de stocker le bit de poids fort de la mantisse : on sait qu'il vaut 1 puisque $m \in [1; 2[$. On stocke donc uniquement la partie après la virgule de la mantisse !⁵

On obtient le format suivant (aussi appelé **norme IEEE 754**) :

Définition 15 (Représentation des nombres dyadiques).

On encode des nombres dyadiques sur 64 bits de la façon suivante :

	signe s	exposant E	mantisse M
64 bits :	1 bit	11 bits	52 bits

Qui représente : $\pm m * 2^e$ avec :

- + si $s = 0$, - sinon.
- E code un entier non-signé sur 11 bits, et $E = e + 1023$. On représente donc les puissances $e \in \llbracket -1023; 1024 \rrbracket$. 1023 est appelé la **constante d'excentrement**.
- $m = 1, M$. C'est à dire que la mantisse stockée ne stocke que ce qu'il y a après la virgule. On a donc en réalité accès à 53 chiffres significatifs.

On parle de représentation en **double précision**, car autrefois on utilisait moitié moins de bits (32). Le double correspondant en C est **double**.

Définition 16 (Valeurs réservées).

Certaines paires de valeur (M, E) sont réservées :

- $M = 0...0$ et $E = 0...0$ encodent ± 0 . Par conséquent, $\pm \overline{1, 0...0} * 2^{-1023}$ n'est pas représentable.
- $M = 0...0$ et $E = 1...1$ encodent $\pm \infty$. Par conséquent, $\pm \overline{1, 1...1} * 2^{1024}$ n'est pas représentable.
- un autre M avec $E = 1...1$ encode NaN (Not a Number).

0.3.1 Erreurs d'approximation et autres limites

Cette représentation double précision a des limites. Elles proviennent du fait que puisque la mantisse est de taille finie, on manipule des approximations⁶.

- Propagation d'erreurs : toutes les erreurs listées ci-dessous se propagent lors des calculs, comme les approximations en physique.
- Beaucoup de nombres sont approximés et non exacts, dont tous les nombres non-dyadiques. Rappelons que 0.1 n'est pas dyadique : on en manipule donc un arrondi, ce qui mène à des erreurs. Si possible, il vaut lui préférer 0.125 qui est lui exact.
- Faux négatif sur les comparaisons : parfois, des comparaisons comme « $a+b=c$ » s'évaluent à **false** alors qu'elles sont mathématiquement justes. Un exemple classique est $0.1 + 0.2 == 0.3$.⁷

5. Ça a l'air de rien, mais on gagne ainsi 1 bit dans la mantisse, c'est à dire un chiffre significatif stockable de plus.

6. Comme en physique : si on a trop peu de chiffres significatifs, le résultat est peu précis.

7. En même temps, **aucun** de ces 3 nombres n'est dyadique. Partant de là, l'égalité eut été un coup de chance incroyable.

- Faux positif sur les comparaisons : de même.
- La somme de deux nombres représentables ne l'est pas forcément. Par exemple, 2^{30} l'est, 2^{-30} l'est, mais il faut un M de 60 chiffres pour stocker leur somme. En fait, il se passe un débordement de la mantisse : un flottant déborde en perdant de la précision.
- Non associativité : on n'a pas forcément $(a+b)+c == a+(b+c)$. Par exemple $(2^{-30} + 2^{30}) - 2^{30}$ ne renvoie pas le même résultat que $2^{-30} + (2^{30} - 2^{30})$.

Corollaire 17.

Quand on compare des flottants, on le fait toujours à epsilon près, avec epsilon la précision voulue!

Par exemple, au lieu de tester $a \neq b$, on teste $|a-b| > \epsilon$.

0.4 Tableaux

Définition 18 (Représentation des tableaux).

Soit τ un type qui se représente sur r bits. Un tableau T de longueur L dont les cases sont de type τ est représenté en mémoire comme :

Représentation de $T[0]$	Représentation de $T[1]$...	Représentation de $T[L-1]$
-----------------------------	-----------------------------	-----	-------------------------------

FIGURE 2.3 – Représentation du tableau T

Autrement dit, pour représenter T , on utilise $L \times r$ bits. Les r premiers bits stockent $T[0]$, les r suivants $T[1]$, etc. Les cases sont contiguës en mémoire.

Remarque.

- Si l'on connaît le type τ , on connaît sa taille r . Si l'on a accès à l'adresse mémoire du début D du tableau, on peut alors aisément calculer :
 - l'adresse de $T[0]$: c'est D .
 - l'adresse de $T[1]$: c'est $D + r$.
 - l'adresse de $T[2]$: c'est $D + 2r$.
 - Ainsi de suite. $T[i]$ a pour adresse $D + ir$.

C'est pour cela que les tableaux sont une structure de donnée aussi efficace : il est très rapide d'accès au i -*me* élément, car cet accès demande une unique calcul à partir de l'adresse du début et de l'indice.

Notez que pour que cette propriété soit vraie, il *faut* que toutes les cases du tableau aient la même taille! C'est pour cela que l'on autorise pas les mélanges de type dans un tableau.

- **Lorsque l'on passe un tableau à une fonction, tout se comporte comme si le contenu du tableau était passé par référence.**

En fait, le tableau est affaibli en pointeur : au lieu de passer le tableau par valeur, on passe à la place l'adresse de sa première case. D'après le point précédent, c'est exactement ce dont on a besoin. Mais puisqu'on a fourni à la fonction l'emplacement mémoire du tableau initial, les cases qu'elle modifie sont celles du tableau.

- Cette façon de manipuler des tableaux est celle de C. Elle a deux inconvénients : la longueur du tableau n'est pas stockée, et le tableau n'est pas redimensionnable (si ça se trouve, les cases mémoire qui suivent la fin du tableau sont déjà prises et on ne peut donc pas agrandir el tableau). OCaml (et Python) utilisent une structure qui permet de stocker la longueur du tableau en plus de ses valeurs⁸. Nous verrons avec les tableaux dynamiques comment crée des tableaux redimensionnables (comme en Python).

8. Dans l'idée, on crée une case fictive au début du tableau qui sert à stocker la longueur.

- Il faut que tous les éléments d'un tableau aient la même taille. En conséquence, si on veut faire un tableau de tableaux, il y a deux grandes façons de faire :
 - Imposer que tous les sous-tableaux aient la même longueur et le même type.
 - Ne pas stocker les sous-tableaux dans les cases, mais l'adresse de leur début (toutes les adresses ont la même taille). Il faut alors stocker les sous-tableaux ailleurs.

(a) Tableaux imbriqués

(b) Tableaux d'adresses de tableaux

FIGURE 2.4 – Les deux grandes façons de faire un tableau de tableaux.

0.5 Chaînes de caractères

Définition 19 (Chaînes de caractères).

Une chaîne de caractère est une succession de caractères, c'est à dire un texte.

On les représente généralement comme une succession de caractères contigus en mémoire. On utilise un caractère particulier, `\0` qui marque la fin (et ne veut rien dire d'autre) :

Premier caractère	Second caractère	...	Dernier caractère	<code>\0</code>
-------------------	------------------	-----	-------------------	-----------------

Remarque. Ceci est la façon de faire de C (qui a l'avantage de la simplicité). Là aussi, on peut stocker la longueur de la chaîne en dur, et/ou faire une représentation plus compliquée qui permet plus d'opérations.

1 Compléments très brefs de programmation

1.0 Notion d'adresse mémoire

La mémoire d'un ordinateur peut être pensée comme un « très gros tableau » : c'est une succession de cases mémoires (dont la taille est un plus petit adressable), ayant chacune un indice. Celui-ci est appelé une **adresse**.

Autrement dit, une adresse indique où est rangée une donnée en mémoire.

Définition 20 (Pointeur).

Une variable qui stocke une adresse est appelée un **pointeur**.

En C, si le contenu de la case mémoire pointée est de type `type`, le type du pointeur sera `type*`.

Nous reparlerons de cette notion en TP. Tout ce qu'il faut retenir pour l'instant, c'est qu'en plus d'utiliser des variables, on peut manipuler des adresses mémoire explicites.

On parle alors d'**objet mémoire** pour désigner quelque chose qui a un contenu et une adresse.

1.1 Portée syntaxique et durée de vie

On a déjà vu comment les fonctions, if-then-else et boucles structurent un code source en différents corps (le corps de la fonction, le corps du if, etc). les lignes qui appartiennent exactement aux mêmes corps sont appelés un **bloc syntaxique**.

Remarque.

- Cette appellation provient de la *syntaxe* : un corps est délimité par quelque chose qui indique son début et quelque chose qui indique sa fin. En C, ces indicateurs sont les `{` et `}`.
- Pour les fonctions, il y a une subtilité : il faut considérer que la déclaration des arguments de la fonction est aussi dans le bloc syntaxique.
- Il y a une notion d'imbrications : par exemple, si une boucle est dans une fonction, on dira que le corps de la boucle est imbriqué dans le bloc de la fonction.

1.1.0 Définition et premiers exemples

Définition 21 (Portée).

La **portée** d'un identifiant (c'est à dire d'un nom de variable ou de fonction) est l'ensemble des endroits où cet identifiant est callable.

Proposition 22 (Portée locale vs globale).

Il y a deux types de portées :

- **locale** : si un identifiant est créé dans un bloc syntaxique, il n'est utilisable qu'à partir de sa création jusqu'à la fin de son bloc syntaxique. On peut l'utiliser dans des blocs qui sont imbriqués dans son corps.
- **globale** : si un identifiant est créé en dehors de tout bloc syntaxique (c'est à dire en dehors de toute fonction), il est utilisable de sa déclaration à la fin du fichier (peu importe les blocs).

Exemple.



```

1  #include <stdio.h>
2  #include <stdlib.h>
3
4
5  int somme = -10;
6  int Lmax = 100000;
7
8
9  int somme_tableau(int T[], int len) {
10     if (len > Lmax) {
11         // faire planter le programme
12         exit(EXIT_FAILURE);
13     }
14
15     int somme = 0;
16     int indice = 0;
17     while (indice < len) {
18         int tmp = 666;
19         somme = somme + T[indice];
20         indice = indice + 1;
21     }
22
23     return somme;
24 }
25
26
27 int main(void) {
28     int T[] = {5, 85, -20};
29     int sortie_somme = somme_tableau(T, 3);
30     printf("variable globale somme : %d\n", somme);
31     printf("sortie de somme_tableau : %d\n", sortie_somme);
32
33     return EXIT_SUCCESS;
34 }

```

Dans l'exemple ci-dessus :

- Dans l'exemple ci-dessus, le bloc syntaxique du `while` de `somme_tableau` commence ligne 18 et se finit ligne 20. Ainsi, la variable `foo` n'est utilisable que entre ces lignes : c'est sa portée.
- Celui de la fonction `somme_tableau` commence ligne 9 (déclaration des arguments) et finit ligne 24. La portée de `indice` est donc de la ligne 16 à la ligne 24. On peut bien utiliser cette variable dans le bloc du `while` puisque celui-ci est imbriqué dans le bloc de la fonction.
- La variable `Lmax` a une portée globale (ligne 6 jusqu'à la fin). On peut donc l'utiliser ligne 10 (dans le bloc de la fonction).

Définition 23 (Masquage).

On parle de **masquage** lorsqu'au sein de la portée d'un identifiant, on redéfinit une nouvelle variable/fonction qui a le même identifiant. L'identifiant d'origine n'est alors plus accessible tant que l'on est dans la portée du nouveau : seul le plus « récent » est utilisable.

Exemple. Dans l'exemple précédent, il y a un masquage du `somme` global (ligne 5) par le `somme` local (ligne 15). On ne peut pas accéder au premier tant que l'on est dans la portée du second (ligne 15-24).

Définition 24 (Durée de vie).

La **durée de vie** d'un objet mémoire (par exemple une variable) est l'ensemble des endroits où cet objet existe et a une valeur définie en mémoire. Cela ne coïncide pas nécessairement avec la portée.

Exemple.

- Dans l'exemple précédent, durant le masquage, on est bien dans la durée de vie du `somme` global (il a toujours sa même valeur en mémoire, il n'a pas été supprimé). Par contre, on est pas dans sa portée à cause du masquage.
- En C, lorsque l'on crée un tableau mais que l'on ne l'a pas encore initialisé (par exemple : `double T[10]`; crée un tableau de 10 nombres à virgule sans l'initialiser), on est dans sa portée puisque l'identifiant a été créé, mais pas dans sa durée de vie puisque le contenu du tableau n'a pas de valeur définie en mémoire.

Proposition 25 (Durée de vie statique, automatique et allouée).

Il y a trois durées de vie possibles :

- **statique** : l'objet est toujours défini en mémoire. C'est le cas des variables globales. Leur valeur est stockée dans une zone particulière de la mémoire, où elles existent dès le début du fichier (avant le début de leur portée, donc).
- **automatique** : l'objet n'existe en mémoire que pendant un bloc. C'est le cas des variables locales : elles "naissent" au début de leur bloc syntaxique et "meurt" automatiquement à la fin de celui-ci.
- **allouée** : on contrôle à la main la "naissance" et la "mort" d'un objet. Cela se fait en C via les fonctions `malloc` et `free`.

Exercice. Dans l'exemple `portee.c` précédent, indiquer des variables statiques et des variables automatiques.

Remarque.

- Si on oublie de libérer la mémoire d'un objet alloué, on provoque une fuite de mémoire : la zone restera allouée jusqu'à la fin du programme, même si on ne s'en sert plus.⁹
- Certains langages de programmation ne proposent pas de contrôler à la main la "mort" des objets de durée de vie allouée. À la place, ils essaient de détecter le moment à partir duquel ils ne seront plus utilisés (et les suppriment alors). On parle de **ramasse-miette**. C'est ce que font OCaml et Python. Cela a l'avantage d'éviter les erreurs du programmeur liée à la mort des objets, mais a l'inconvénient de ralentir le programme (car il faut détecter si une variable servira encore ou non).

Proposition 26.

Les objets à durée de vie automatique ont une propriété de type LIFO (*Last In First Out*) : ce sont ceux du dernier bloc ouvert qui seront supprimés en premier.

Cette propriété provient du fait que les blocs eux-mêmes vérifient cette propriété : le bloc qui a été ouvert le plus récemment est le prochain à fermer. Vous pouvez y penser comme à des parenthèses : quand vous croisez une parenthèse fermante, elle ferme la parenthèse encore ouverte la plus récente.

9. Pire encore : même si on ne peut plus libérer la mémoire, car cette libération avait besoin d'une variable locale à laquelle nous n'avons plus accès.

1.1.1 Exemples avancés

Avec les objets alloués, on peut créer d'autres exemples de désynchronisation entre la durée de vie et capacité à accéder au contenu :

```
1 int* ptr = (int*)
  ↪ malloc(<taille>);
2 free(p);
3 int x = p[0];
```

Lors de la ligne 4 de ce code, l'objet alloué n'existe plus mais on peut encore y accéder via `p`.

⚠ Attention, c'est l'objet pointé par `p` qui est mort. `p`, lui, va très bien et contient toujours la même adresse (sauf qu'il n'y a plus rien à cette adresse).

```
1 int* zombi(void) {
2     int var = 42;
3     return &var;
4 }
```

Cette fonction renvoie l'adresse d'une variable locale. Or celle-ci meurt à la fin de la fonction : on récupère donc un pointeur sur de la mémoire morte.

```
1 void fuite(unsigned int n) {
2     malloc(n*sizeof(char));
3     return;
4 }
```

Après un appel à cette fonction, l'objet mémoire créé existe toujours mais on n'a plus aucun moyen d'y accéder. Il est même impossible de libérer cette mémoire car on n'a pas mémorisé le pointeur qui y mène...

```
1 double* zombieRetour(void) {
2     double tab[5] = {3.14, -4.2,
  ↪     2.71, 5.0};
3     return tab;
4 }
```

Idem qu'à gauche, sauf qu'ici ça se voit moins (surtout si on a fait du Python). Le tableau est affaibli en pointeur lorsqu'on le renvoie, c'est donc exactement la même situation.

Pour enfoncer le clou dans le cadavre des zombies, considérons le code ci-dessous et demandons au compilateur son avis :

```
1 int main(void) {
2     double* tab = zombieRetour();
3     return EXIT_SUCCESS;
4 }
```

Ligne de compilation et résultat :

```
1 MP2I/Cours/Memoire: gcc error.c -Wall -Wextra
2 error.c: Dans la fonction « zombieRetour »:
3 error.c:5:10: attention: la fonction retourne l'adresse d'une variable
  ↪ locale [-Wreturn-local-addr]
4     5 |     return tab;
5       |           ^~~
```

Voilà. Tout est dit.

2 Organisation de la mémoire d'un programme

Dans cette section, nous allons voir comment est « rangée » la mémoire dans le langage C : où vont les octets des différentes variables d'un programme.

Pour OCaml, c'est plus compliqué : la notion de pile d'appels de fonction reste valide, mais le reste est plus différents (pour faire fonctionner le ramasse-miette).

2.0 Bloc d'activation

Vous vous encourage très fortement à aller jouer avec le site ci-dessous pour visualiser la mémoire et les blocs :

<https://pythontutor.com/c.html#mode=edit>

Définition 27 (Bloc d'activation).

Les variables créées par un même bloc syntaxique sont stockées les unes après les autres en mémoire. L'espace mémoire servant à stocker ces variables est appelé **bloc d'activation**. Il contient donc les variables créées dans ce bloc syntaxique (mais pas dans ses blocs imbriqués). Rappelons que si le bloc syntaxique est un bloc de fonction, les arguments en font partie.

Exemple.

```

1  int f(int a, int b) {
2      int c = 10;
3      int i = 0;
4      while (i < c) {
5          int d = 42;
6          int c = 50;
7          i = i+1;
8      }
9      int e = 2;
10     return c;
11 }
```

 bloc.c

Le bloc d'activation de la fonction contient un sous-bloc d'activation (boucle for) qui s'étend des lignes 3 à 6.

On peut représenter ces deux blocs sur un schéma où on les « empile » :

FIGURE 2.5 – Blocs d'activation de `f`

Remarque.

- Un appel de fonction ouvre un nouveau bloc (celui du "code" de la fonction appelée).
- Beaucoup de langages ne font pas un bloc d'activation par bloc syntaxique mais simplement un gros bloc d'activation pour toute la fonction.

2.1 Organisation de la mémoire virtuelle d'un programme

On parle de **mémoire virtuelle** (abrégié VRAM) pour désigner la mémoire telle que le système d'exploitation la présente à un programme. C'est une version « mieux rangée » de la mémoire réelle, que le système d'exploitation s'occupe de faire correspondre avec la mémoire réelle.¹⁰

Tout comme la mémoire réelle, la mémoire virtuelle est un « tableau géant » indicé par des adresses. Du point de vue d'un programme, la mémoire virtuelle ressemble à ceci :

FIGURE 2.6 – Mémoire virtuelle vue par un processus.

Légende :

- *Code* : c'est là où les instructions du programme (son... code) sont stockées.
- *Données statiques initialisées* : une donnée statique est une donnée qui est connue à la compilation. Les données statiques initialisées sont les variables globales, ainsi que le contenu des chaînes de caractères¹¹
- *Données statiques initialisées* : hors-programme.
- *Pile mémoire* : c'est là que sont stockés les différents blocs d'activation en cours, empilés les uns après les autres.
- *Tas mémoire* : là où on stocke les objets à durée de vie allouée.

Ces deux dernières zones évoluent durant le programme, au fur et à mesure que des variables sont créées ou détruites.

10. C'est la différence entre la description que je fais à autrui de mes brouillons, et l'état réel de mes brouillons. Tel un système d'exploitation, je suis capable de lire mes brouillons fort mal organisés et de les faire correspondre en direct à quelque chose de plus présentable.

11. Une chaîne de caractère est un `char*`, c'est à dire un pointeur vers une zone où les caractères sont rangés les uns à côté des autres. Cette zone se trouve dans les données statiques initialisées.

2.1.0 Pile mémoire

Quand on entre dans un nouveau bloc syntaxique, on crée son bloc d'activation associé et on le met au bout de la pile mémoire. Quand on quitte un bloc syntaxique, on supprime son bloc d'activation.

FIGURE 2.7 – Schéma de l'évolution de la pile mémoire

Exercice. Représenter de même l'évolution de la pile mémoire lors de l'exécution du programme dont le code C est portée. c.

Proposition 28 (Organisation de la pile mémoire).

Les blocs d'activation sont stockés sur la pile mémoire suivant le principe LIFO. La pile est donc toujours remplie de manière contigu : elle n'a pas de « trou ».
La taille de la pile évolue durant l'exécution.

Remarque. La pile mémoire se comporte comme la structure de donnée pile (que nous verrons plus tard).

2.1.1 Tas mémoire

Les objets à durée de vie allouée sont stockés dans le tas. Quand le programme demande à allouer (« réserver », « faire naître ») x octets, on trouve x octets dans le tas et on les fournit au programme. Ils ne seront pas réutilisables tant que le programme n'aura pas explicitement libéré cette mémoire.

Exemple. Considérons le programme suivant, et montrons un exemple possible d'évolution du tas pour ce programme :

- 1 $x \leftarrow$ allouer 1 cases
- 2 $y \leftarrow$ allouer 3 cases
- 3 $z \leftarrow$ allouer 2 cases
- 4 Désallouer y

Proposition 29 (Non-contigüité du tas mémoire).

Le tas mémoire n'est pas rempli contigüement.

Remarque.

- Cette différence avec la pile mémoire provient de fait que l'on a pas accès à une bonne propriété comme LIFO : les objets alloués peuvent être libérés dans n'importe quel ordre.
- Il existe des stratégies pour essayer de limiter la création de « petits trous » dans le tas. On parle de stratégie d'allocation.
- Le tas mémoire n'a rien à voir avec la structure de donnée tas.

2.1.2 À savoir faire

L'objectif premier de cette section sur la mémoire d'un programme est que vous soyez capable de faire des schémas montrant l'évolution de la pile mémoire et une évolution possible du tas mémoire durant l'exécution d'un programme.¹²

Ce sont des notions importantes à avoir en tête lorsque l'on programme en C avec des pointeurs. En fait, **pour comprendre ce que l'on fait en C, il faut être capable de visualiser la mémoire.**

12. Notez que tout ce qui n'est pas marqué hors-programme est au programme. En particulier, j'ai déjà vu un sujet de concours MPI poser la question « dans quelle zone mémoire sont stockées les variables globales ? ».

