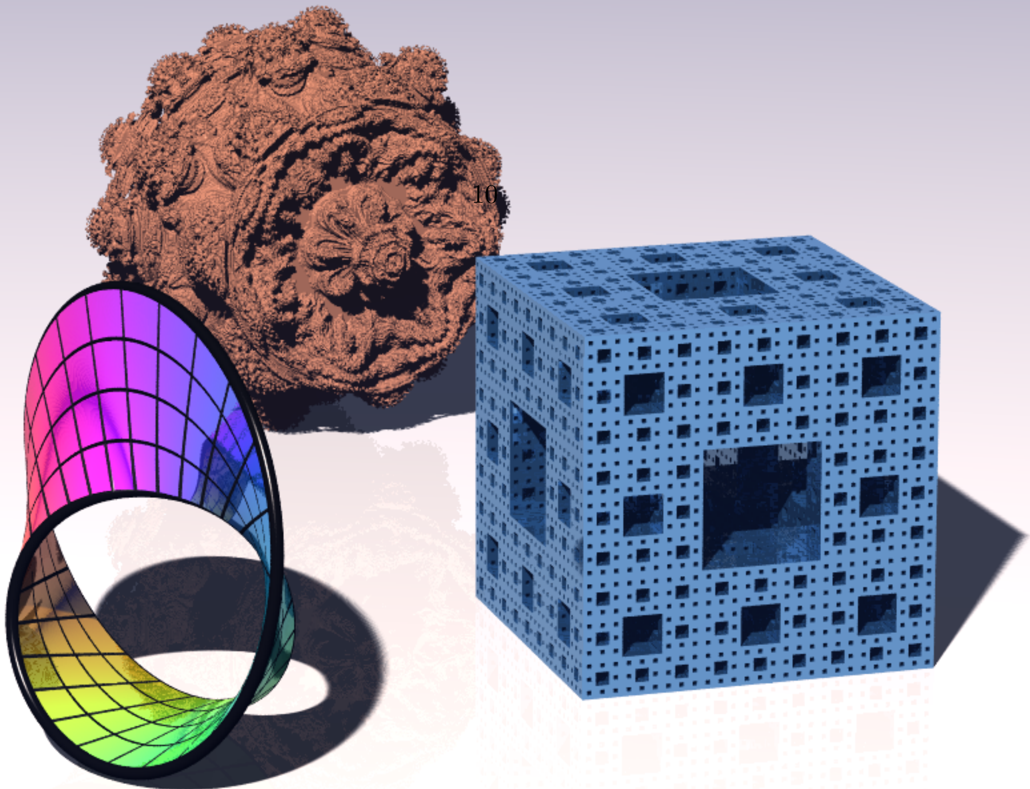
# Informatique

## MP2I




**Guillaume Dewaele**
**Lycée Louis-le-Grand**

# Introduction au langage C

« Computers are like Old Testament gods – lots of rules and no mercy. »

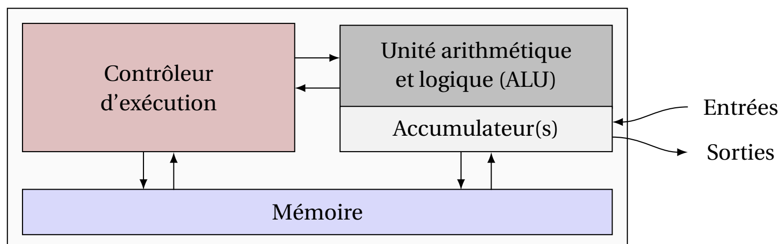
— *Joseph Campbell*

## 1 Du programme au processeur

### 1.1 Des instructions pour le processeur

Si l'informatique, comme nous le verrons, est une science à part entière qui va bien au-delà de la seule programmation (Edsger W. Dijkstra disait que l'ordinateur n'est guère à l'informatique que ce que le télescope est à l'astronomie), cette dernière en reste un aspect majeur et concret.

En un peu plus d'un demi-siècle, on a vu fleurir des centaines de *langages de programmation*. Afin de mieux saisir l'intérêt que ceux-ci présentent, penchons tout d'abord brièvement sur le fonctionnement d'un ordinateur. Dans le modèle d'architecture développé par John Von Neumann, qui décrit encore fort bien les machines actuelles, le processeur extrait de la mémoire de la machine la séquence d'instructions (le *programme*) qu'il devra exécuter. Ces instructions auront pour effet d'altérer les valeurs mémorisées dans cette même mémoire, et d'échanger des informations avec l'utilisateur via des entrées/sorties.




Les données en mémoire sont usuellement dans un format binaire [1], et cela inclue la séquence d'instructions à exécuter, sous une forme directement compréhensible par les circuits électriques ou électroniques du processeur. Ainsi, le calcul de la factorielle d'un entier pourrait, pour un processeur intel i86, être stocké (et ultérieurement lu et exécuté par processeur) sous la forme de la séquence ci-dessous, retranscrite ici sous une forme hexadécimale, afin d'en faciliter la lecture.

```
b8 01 00 00 00 83 f9 01 7e 07 0f af c1 ff c9 eb f4 c3
```

Comme il serait extrêmement malcommode pour un humain de manipuler directement des codes numériques, qu'ils soient binaires ou hexadécimaux, une première amélioration a consisté à concevoir une alternative humainement lisible aux séquences de bits constituant les instructions destinées au processeur. On parle de *langage assembleur* ou *langage d'assemblage*.

Les programmes en langage assembleur consistent donc en une suite de *mnémoniques*, symboles plus aisés à retenir et manipuler, représentant les opérations à effectuer par le processeur, avec d'éventuels arguments. Notre fonction factorielle devient, en langage assembleur :

```
       mov    $0x1, %eax
 loop: cmp    $0x1, %ecx
       jle    end
       imul   %ecx, %eax
       dec    %ecx
       jmp    loop
  end: ret
```

Les mnémoniques ne sont qu'une traduction *directe* des codes binaires utilisés par le processeur. Ainsi, l'instruction « `mov $0x1, %eax` », qui place la valeur 1 dans le registre processeur nommé EAX, correspond très exactement à la séquence hexadécimale `b8 01 00 00 00`. De même, l'instruction « `imul %ecx, %eax` » qui réalise une multiplication entière entre deux registres à la séquence `0f af c1`.

Le processeur peut ainsi effecuter des opérations de différentes natures : opérations arithmétiques et logiques (multiplications, décrémentations, comparaisons...), manipulation de la mémoire, contrôle du flux d'exécution (sauts conditionnels ou non...)

### 1.2 Documenter le code pour les humains

Il est tout à fait possible d'écrire des programmes complexes en langage assembleur, cela a même été la norme pendant fort longtemps dans certains domaines, et c'est encore occasionnellement le cas. Il est toutefois rapidement difficile de s'y retrouver si le programme est long.

Une solution partielle a consisté à ajouter des commentaires explicitant le but de chaque ligne de code, lesquels sont ignorés lorsque le langage assembleur est traduit en séquences

1. C'est à dire une séquence constituée de 0 et de 1, nous reviendrons ultérieurement sur les avantages concret que cela représente.

hexadécimales pour le processeur, mais *très* précieux pour les programmeurs, comme dans le morceau du code utilisé dans les missions Apollo [2], reproduit ci-dessous :

```
P63SPOT3 CA       BIT6       # IS THE LR ANTENNA IN POSITION 1 YET
         EXTEND
         RAND     CHAN33
         EXTEND
         BZF      P63SPOT4   # BRANCH IF ANTENNA ALREADY IN POSITION 1

         CAF      CODE500    # ASTRONAUT:    PLEASE CRANK THE
         TC       BANKCALL   #               SILLY THING AROUND
         CADR     GOPERF1
         TCF      GOTOP00H   # TERMINATE
         TCF      P63SPOT3   # PROCEED       SEE IF HE'S LYING

P63SPOT4 TC       BANKCALL   # ENTER         INITIALIZE LANDING RADAR
         CADR     SETPOS1

         TC       POSTJUMP   # OFF TO SEE THE WIZARD ...
         CADR     BURNBABY
```

On remarquera aisément que le langage assembleur ci-dessous, bien que partageant des points communs avec le langage assembleur i86 vu tantôt, est fort différent. C'est en partie dû à la nature particulière de l'AGC. Produit de son époque, et d'abord développé pour être extrêmement fiable et gérant de nombreuses entrées (capteurs), ses possibilités sont limitées… il ne peut nativement effectuer que des additions, et aucune opération logique, même si bien entendu il est possible d'y parvenir indirectement. Mais de façon générale, un langage assembleur est profondément lié au processeur auquel il est destiné.

## 1.3 Langages de programmation

Le caractère spécifique des langages d'assemblage est rapidement un handicap. En effet, lorsque l'on change le processeur, le programme entier est à réécrire, ce qui n'a rien de très séduisant. En outre, de part les possibilités relativement limitées de ces langages en terme d'expressivité, écrire des programmes en langage assembleur nécessite des trésors de patience et d'organisation. Cela n'a rien d'un modèle d'efficacité.

Pour pallier ces difficultés, on a donc élaboré des *langages de programmation* qui permettent de décrire, de façon claire et lisible pour un être humain, les structures de données à manipuler ainsi que les opérations qui doivent être effectuées sur ces données, tout en s'affranchissant des spécificités techniques de la machine sur laquelle ces opérations seront effectuées.

2. L'intégralité du code du dipositif de contrôle (AGC) est disponible sur GitHub, et la lecture en est fascinante, avec beaucoup de commentaires amusants ou parfois inquiétants (lorsque l'on trouve des morceaux de code dans la version définitive où l'auteur a indiqué « *TEMPORARY, I HOPE HOPE HOPE* »).

Le langage C, qui nous occupera pour le moment, est un tel langage développé de manière à être raisonnablement portable d'une architecture à une autre. Nous verrons que notre calcul de la factorielle d'un entier $n$ pourrait par exemple s'écrire, dans ce langage, de la sorte :

```c
int fact(int n) {
    int acc = 1;
    while (n>1) {
        acc = acc*n;
        n = n-1;
    }
    return acc;
}
```

Les opérations spécifiques à un processeur ont ici disparu, et avec tantinet d'habitude, le sens du programme est également plus clair. En outre, les règles régissant la syntaxe d'un tel langage de programmation permettent d'identifier de manière automatique des incohérences dans la rédaction, et donc repérer d'éventuelles erreurs.

De très nombreux langages de programmation sont donc apparus en même temps que les premiers ordinateurs programmables. Parmi les plus anciens de ces langages, on peut citer notamment le langage FORTRAN (FORmula TRANslation) créé en 1954, les langages LISP et ALGOL (ALGOrithmic Language) apparus en 1958, ou encore le langage COBOL (COmmon Business Oriented Language) datant de 1959. Ces précurseurs ont posé les premières bases des langages informatiques, et il est intéressant de voir que, bien que leur syntaxe ait beaucoup évolué en un peu plus d'un demi-siècle, et que de nombreux autres langages sont apparus depuis, ces langages originaux restent pour nombre d'entre eux toujours utilisés à l'heure actuelle.

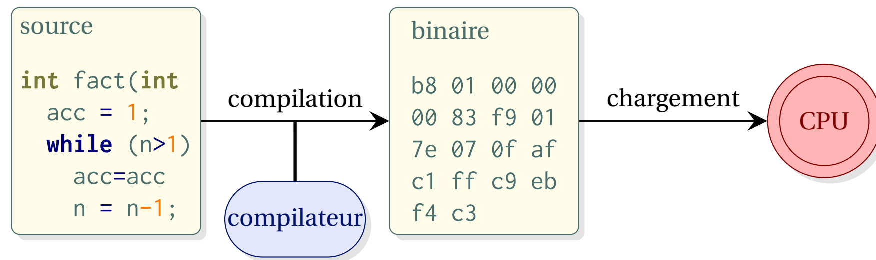
Le morceau de programme précédent est qualifié de *code source*. Il s'agit d'un texte qui présente les instructions composant un programme sous une forme lisible pour un être humain. Ce texte étant généralement mémorisé dans un fichier, on parle naturellement également de *fichier source*.

## 1.4 Compilation et interprétation

Bien évidemment, le processeur, lui, ne comprend que les instructions en binaire. Il doit donc y avoir un travail de traduction pour transformer le code source en la série de bits qui pourra être transmise au processeur.

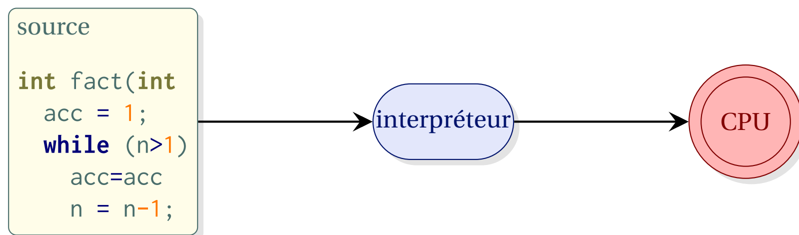
La première solution, pour ce faire, est la *compilation*. Il s'agit de fournir le code source à un programme spécialisé, appelé *compilateur*, qui analyse et vérifie sa syntaxe, puis le transforme en une séquence binaire compréhensible par le processeur et enregistre le

résultat dans un fichier, contenant le programme compilé. À chaque fois que l'on voudra exécuter le programme, il suffira donc d'envoyer la séquence binaire au processeur.




C'est la méthode généralement utilisée pour le langage C, pour lequel il existe de très nombreux compilateurs : cc, icc, gcc, Clang, MinGW, visual-C pour n'en citer que quelques-uns... Les langages basés sur ce principe sont qualifiés de *langages compilés*. C'est ainsi que fonctionnent les langages FORTRAN ou OCaml. Mais il ne s'agit pas de la seule possibité.

Une autre solution consiste à disposer d'un *interpréteur*, programme actif lorsque l'on veut exécuter notre programme qui va traduire, à la volée et au tout dernier moment, les instructions de notre code source en instructions binaires adatées au processeur.




On parle, naturellement, de *langages interprétés*. C'est (à peu près) sur ce principe que fonctionne le langage Python, ou que fonctionnait le langage LISP initialement.

Les deux approches ont évidemment leurs avantages et leurs inconvénients. La compilation permet d'effectuer la traduction une seule fois, et pusque l'on n'a pas de contraintes de temps et que l'on a une vision globale du programme, il est possible d'obtenir des optimisations intéressantes. L'interprétation dispense de l'étape de compilation et permet donc un déploiement plus simple, mais l'interprétation a généralement un coût en terme de vitesse d'exécution (et on effectue la traduction de nombreuses fois).

Dans la pratique, les deux approches peuvent se mélanger. Ainsi, beaucoup de langages qui semblent interprétés effectuent une compilation dite « *just in time* » et ont un mécanisme de « cache ». Parfois les solutions sont encore plus complexes, comme le langage Java qui possède une étape de compilation, laquelle produit un fichier binaire contenant des commandes pour un processeur « virtuel [3] », et lors de l'exécution, ce code binaire est retraduit au dernier moment pour s'adapter au processeur réel. Un des avantages est que le binaire intermédiaire peut être utilisé sur des architectures différentes.

Enfin, précisons que le langage n'impose pas la façon dont les programmes seront transmis au processeur. On peut donc très souvent trouver à la fois des compilateur et des interpréteurs pour un même langage (ce qui peut être utile, les interpréteurs permettant une mise au point plus rapide, et les compilateurs souvent une meilleure efficacité).

## 1.5 Naissance du C, forces et faiblesses

Entre 1969 et 1973, plusieurs personnes travaillaient dans les laboratoires Bell sous la direction de Kenneth Thompson à la mise au point du système d'exploitation [4] Unix. Comme il eût été difficile de programmer celui-ci entièrement en assembleur, le langage de programmation C a été élaboré en parallèle, notamment par Dennis Ritchie, afin d'avoir des sources plus lisibles et plus aisément transposables sur différentes architectures.

Le C s'est démarqué de ses concurrents de l'époque par ses capacités à intéragir très efficacement avec le matériel, un besoin impérieux puisqu'il était développé pour servir au développement du système Unix, tout en restant aussi portable que possible. Cette spécificité a rendu le C pratiquement incontournable dans le développement de systèmes d'exploitations et d'applications dites « bas niveau » travaillant de très près avec le matériel.

Le langage C a considérablement évolué avec les années. Il à commencé à se fixer avec l'écriture de l'ouvrage « *the C programming language* » par Dennis Ritchie et Brian Kernighan qui en présentait les bases. Toutefois, les différents compilateurs de l'époque implémentaient chacun des variantes du langage, ce qui gênait la portabilité. Il a donc été normalisé en 1989 par l'American National Standard Institute (on parle de C ANSI) puis en 1990 a fait l'objet d'une norme ISO (on parle de C90). Diverses améliorations et corrections ont été ensuite apportées pour arriver à la fin du siècle dernier à la norme C99.

Depuis, d'autres améliorations, plus modestes, ont été apportées au langage, avec des mises à jour en 2011 (C11), 2018 (C17) et 2024 (C23), et une prochaine attendue d'ici la fin de la décennie (C2Y). Dans le cadre de ce cours, nous nous baserons sur la norme C99 qui est largement répandue et supportée (même si certains programmes et certaines architectures utilisent encore actuellement les versions de 1989 ou 1990).

Le C est un langage relativement restreint, et dont la philosophie reste très proche d'une architecture réelle (certes un peu idéalisée). Cela permet d'effectuer des optimisations très aggressives, de sorte que le code binaire produit est en général très efficace en terme d'utilisation mémoire et vitesse d'exécution. Cette recherche de l'efficacité a gouverné les

3. L'interpréteur Python usuel effectue également une compilation vers un langage d'assemblage pour un processeur virtuel, qui est ensuite interprété.
4. Un programme exécuté dans un ordinateur et permettant de gérer ses ressources et de permettre l'exécution d'autres programmes, notamment des logiciels applicatifs.

choix du langage, au détriment de mécanismes de sécurité ou de confort de programmation. Il faut donc être très prudent si l'on veut obtenir du code qui fonctionne comme on le souhaite[5].

Précisons que, si le langage C a été choisi pour l'enseignement de MP2I, ce n'est pas pour ses qualités intrinsèques, ni parce qu'il demeure encore, à l'heure actuelle, omniprésent[6]. Il ne faut pas voir ce choix comme une incitation à l'utiliser dans un projet, il y a très souvent quantité de langages bien mieux adaptés (même pour de la programmation système). Mais le langage C présente l'intérêt rare de laisser au programmeur la possibilité de se charger lui-même de certaines tâches, telle que la gestion de la mémoire. Or une bonne compréhension de la manière dont les objets sont gérés en mémoire est inappréciable en informatique.

## 2 Un premier aperçu d'un fichier source C

Sautons directement dans le bain, et examinons un premier programme écrit en C, visant à calculer et afficher le produit $6 \times (6+1)$ :

```c
#include <stdio.h>                                // ①
#include <stdlib.h>

int main(void) {                                  // ②
    int cherry;                                   // ③
    int mango;

    cherry = 6;                                   // ④
    mango = cherry + 1;
    printf("Le produit de %d avec %d vaut %d\n",  // ⑤
              cherry, mango, cherry * mango);

    return EXIT_SUCCESS;                          // ⑥
}
                                                  // ⑦
```

① : le langage C étant relativement « limité », il est fréquent qu'un programme débute par des directives (commençant par le symbole `#`) destinées à permettre l'utilisation de fonctions, constantes et types supplémentaires, fournis soit en standard par le langage (on parle de bibliothèque standard du langage C), soit par d'autres morceaux de code.

Le mécanisme ressemble fortement[7] à celui de la directive `import` de Python, à ceci près qu'il n'est pas possible de sélectionner précisément ce dont on a besoin : on importe la totalité du contenu du module[8] !

Dans un premier temps, nous n'en utiliserons que quelques-unes :

- `#include <stdio.h>` qui contient des fonctions permettant le dialogue avec l'utilisateur, en particulier pour afficher du texte à l'écran et pour lire des choses entrées au clavier (nécessaire ici pour permettre l'utilisation de la fonction `printf`) ;
- `#include <stdlib.h>` qui regroupe quelques fonctions d'usage général, telles que les fonctions `abs` (renvoyant la valeur absolue d'un entier) ou `rand` (renvoyant une valeur pseudo-aléatoire) ou des constantes telles que `EXIT_SUCCESS` ;
- `#include <stdbool.h>` qui définit quelques objets pour permettre l'utilisation de booléens.

Il en existe évidemment de nombreuses autres, que nous introduirons en fonction de nos besoins.

Profitons également de l'occasion pour préciser que « `//` » introduit un commentaire : ce qui suit est ignoré par le compilateur, et ce jusqu'à la fin de la ligne. Si l'on souhaite écrire des commentaires sur plusieurs lignes, on fera précéder le commentaire par « `/*` » et on indiquera la fin du commentaire avec « `*/` ».

② : le cœur d'un programme C débute avec l'une des deux mentions suivantes :

- `int main(void)`
- `int main(int argc, char* argv[])`

C'est la partie qui suit cette mention (entre[9] les symboles `{` et `}`) qui sera exécutée. On trouvera donc naturellement toujours l'une de ces deux formes dans toute source C qui doit être exécutée par la machine. La seconde forme est utilisée lorsque le système chargé de lancer l'exécution du programme est en mesure de lui fournir des paramètres, et donne un moyen d'accéder à ces paramètres. Son écriture peut varier quelque peu, et nous nous pencherons dessus ultérieurement. Pour l'instant, nous utiliserons la première forme.

③ : on observe ici des *déclarations* de variables, qui servent principalement à réserver de la place dans la mémoire pour y mémoriser des informations. Dans la norme C99, ces déclarations peuvent se trouver à divers endroits, en fonction des besoins. Les déclarations font l'objet de la prochaine section.

---

5. Si l'on peut parfois craindre de se tirer dans le pied, le C tend à fournir un bazooka chargé pour le faire.
6. En particulier parce que le remplacer nécessite souvent de repartir presque à zéro et de changer beaucoup d'habitudes, ce qui n'est pas souvent chose facile.
7. Du moins dans l'esprit, le fait que Python soit usuellement un langage interprété et que le langage C soit compilé donne naturellement à des différences dans la pratique.
8. Du moins les noms des fonctions contenues dans le module. Généralement, seules les fonctions utiles se retrouveront dans le fichier compilé.
9. Dans une source C, il peut y avoir de nombreuses accolades `{` et `}`. Comme les parenthèses en français ou en mathématiques, les symboles ouvrant et fermant sont associés deux à deux, selon les mêmes principes. La fin du principal morceau de code correspond donc à l'accolade fermante `}` qui est le pendant de l'accolade ouvrante `{` qui suit immédiatement le `int main(...)`. Pour des raisons pratiques, les éditeurs de code mettent généralement en évidence les symboles qui sont associés deux à deux.

④ : le programme commence pour de bon avec les premières affectations et les premiers calculs. On remarquera que chaque *instruction* (de même que chaque déclaration, précédemment) se termine par un point-virgule. De même que les points règlent le rythme du discours en français, les points-virgules ont ce rôle en C. Peut-être savez-vous qu'ils ont le même rôle en Python, mais qu'on les y rencontre beaucoup moins souvent : en effet, en Python, la fin de la ligne joue spontanément le rôle de terminaison d'une instruction (et on recommande généralement de ne pas mettre plusieurs instructions sur une même ligne). En C, le retour à la ligne ne joue pas ce rôle (les instructions peuvent s'étaler sur plusieurs lignes) donc les points-virgules sont indispensables [10].

⑤ : l'instruction `printf` est celle qui, généralement, permettra d'afficher des résultats pour l'utilisateur de votre programme. Elle permet de faire de nombreuses choses, nous en étudierons la syntaxe au fur et à mesure de nos besoins. Dans l'exemple présent, la phrase délimitée par les guillemets doubles est affichée, mais les trois occurrences de `%d` dans cette phrase sont remplacées par les valeurs des trois entiers fournis ensuite. Cet appel de fonction s'étend sur deux lignes, ce qui ne pose aucune difficulté particulière.

Profitons de l'occasion de signaler que l'indentation n'a pas de sens particulier en C, contrairement à ce qu'il se passe en Python. Ainsi, rien n'empêche un programme en langage C de ressembler à ceci [11] :

```c
                                                    extern int
                                                       errno
                                                         ;char
                                                            grrr
                             ;main(                           r,
  argv, argc )            int    argc                           ,
   r        ;           char *argv[];{int                     P( );
#define x  int i,       j,cc[4];printf("      choo choo\n"      ) ;
x  ;if    (P(  !        i              )        |  cc[  !      j ]
& P(j     )>2  ?        j              :        i  ){*  argv[i++ +!-i]
;              for    (i=              0;;    i++                   );
_exit(argv[argc- 2    / cc[1*argc]|-1<<4 ]    ) ;printf("%d",P(""));}}
  P  (    a  )   char a   ;  {    a  ;   while(    a  >      "  B   "
  /* -    by E            ricM    arsh             all-      */);
```

Toutefois, on conviendra bien volontiers que le programme précédent n'a rien de lisible. Un programme étant généralement bien plus souvent lu qu'il n'est écrit, et dans la mesure où les langages de programmation ont été créés pour être plus lisibles que le langage machine, il est plus que recommandé de garder les bonnes habitudes prises avec le langage Python !

En ce qui concerne l'indentation et le positionnement des accolades, nous utiliserons dans ce cours une légère variante du style « K&R », employé par Brian Kernighan et Dennis Ritchie, notamment dans leur livre introduisant le langage C.

⑥ : en général, il est attendu qu'un programme C renvoie une valeur numérique entière signalant si l'exécution s'est déroulée correctement ou non. Si tout s'est bien passé, on renverra la valeur 0. Sinon, on renvoie un entier indiquant ce qui s'est mal passé (il suffit ensuite de documenter la signification de chacune des valeurs non-nulles). la constante `EXIT_SUCCESS`, définie dans `stdlib.h`, correspond ici simplement à la valeur signifiant « tout s'est bien déroulé ».

⑦ : on le verra, le C a quantité de règles curieuses ou surprenantes... la norme C impose ainsi que tout programme C doit, en théorie, impérativement se terminer par un retour à la ligne (et donc une dernière ligne « vide »). Historiquement, cette règle existe pour simplifier le traitement automatisé du fichier. La plupart des compilateurs s'en sortent très bien même si l'on oublie ce dernier retour à la ligne, mais vous pouvez vous attendre à ce que dans certains cas cette erreur soit signalée [12].

# 3 Déclarations

## 3.1 Principe

Les programmes que l'on va écrire en C suivent assez fidèlement la philosophie originale décrire par l'architecture de Von Neumann. Très fréquemment, nous allons donc récupérer des données depuis la mémoire et les placer dans les registres du processeur, effectuer des calculs sur ces données, et ranger les résultats obtenus en mémoire. Fort heureusement, le compilateur se chargera de gérer les déplacements de données entre la mémoire et le processeur, mais il nous faut néanmoins savoir où aller chercher les données dans la mémoire et où inscrire les résultats.

Il serait fort malcommode d'avoir à désigner les emplacements en mémoire directement à partir de leurs *adresses* (c'est-à-dire, en gros, le numéro de la « case » (ou des cases) dans la mémoire où sont rangées les données concernées). En effet, il faudrait que le programmeur sache à tout moment quelles sont les zones de la mémoire libres et utilisables, qu'il les gère correctement, et qu'il se souvienne où sont rangés précisément chacune des données manipulées par son programme.

10. Et leur oubli sera probablement l'erreur que vous rencontrerez le plus souvent à la compilation, surtout si vous ne faites pas du C en permanence !

11. Une des contributions à la compétition IOCCC, par Eric Marshall en 1986. Cette compétition vise à écrire des programmes aussi remarquables qu'illisibles. Quelques exemples de codes avec une mise en page « artistique » se trouvent à cette adresse : `dgopstein.github.io/articles/ioccc-ascii-art/`

12. En fait, c'est plus fourbe que cela, si ce retour est manquant, le comportement du programme compilé est considéré comme indéfini, ce qui signifie que le compilateur peut en théorie produire un exécutable qui fait n'importe quoi, y compris quelque chose qui n'a absolument rien à voir avec votre fichier source. On y reviendra en détail ultérieurement.

Pour faciliter l'usage de la mémoire au programmeur, le langage C met donc à sa disposition des *variables*. On crée une variable grâce à une *déclaration*, telle que

```c
int cherry;
```

On y trouve deux informations : le *type* de la variable souhaité (ici `int`), et le *nom* qui permettra dans la suite d'y faire référence (`cherry` dans le cas présent [13]).

Le type de la variable correspond à la nature de la donnée que l'on veut pouvoir stocker en mémoire. On dispose de nombreux types en C, que nous étudierons au fur et à mesure de nos besoins. Pour le moment, nous nous contenterons de trois d'entre eux :

- le type `int` qui permet de mémoriser des entiers relatifs « pas trop grands [14] » tels que 0, −37 ou 42 ;
- le type `double` [15] qui fait référence à des *flottants*, c'est-à-dire des valeurs numériques qui peuvent utiliser des virgules ou des exposants, telles que 4.625, −17.0 ou $2.2 \times 10^{11}$ (que l'on notera `2.2e11`), et qui sont ce que l'informatique a généralement de plus proche des réels [16] ;
- le type `bool` [17] qui correspond aux *booléens*, plus précisément l'ensemble constitué des valeurs booléennes `true` et `false`.

Le nom de variable, qui identifiera la variable dans la suite du programme, doit être une succession de caractères, choisis parmi les vingt-six lettres usuelles de l'alphabet (non accentuées, minuscules ou capitales), les dix chiffres et le tiret inférieur _. Il doit impérativement débuter par une lettre. Et bien entendu, ce nom de variable doit être distinct des mots réservés du langage.

13. Il est bien évidement que les noms de variable n'ont pas à être des noms de fruits ! Toutefois, dans le cas de programmes qui ne font rien de bien intéressant, et pour lesquels il est difficile de trouver des noms de variable parlants, c'est une habitude fréquente pour éviter toute confusion possible entre un nom de variable et un mot réservé du langage. Profitons de l'occasion pour préciser que l'on choisira, dans ce cours, des noms de variables et de fonctions en anglais, comme c'est fréquemment l'usage en informatique. Nous conserverons cependant des commentaires en français.

14. Nous reviendrons en détail sur les contraintes que le stockage impose aux valeurs qui peuvent être mémorisées. Sur un ordinateur de bureau, les entiers représentables peuvent par exemple correspondre aux entiers $n$ vérifiant $2^{-31} \leqslant n < 2^{31}$. Toutefois, ces limites dépendent de la machine et du compilateur. Le module `limits.h` fournit les constantes `INT_MIN` et `INT_MAX` qui correspondent au plus petit et plus grand entiers représentables par le type `int`.

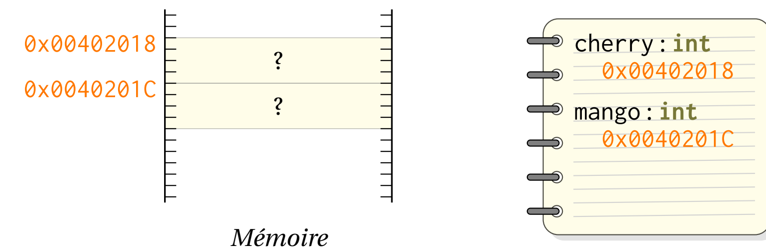
15. Le nom peut surprendre, nous verrons qu'il existe un type `float`, `double` faisant référence à des flottants de plus grande précision, devenus le standard *de facto* en langage C.

16. Il s'agit en fait plus exactement essentiellement d'un ensemble fini de nombres dyadiques (c'est-à-dire de la forme $k \times 2^p$ où $k$ et $p$ sont des entiers relatifs). La aussi, nous reviendrons ultérieurement sur les limitations que cela impose.

17. À la condition de charger préalablement le module `stdbool`. En fait, nous verrons que le langage C ne dispose pas réellement d'un type « booléen ». Toutefois la version C99 a introduit un (faux) type `_Bool` qui permet de stocker de façon explicite des valeurs booléennes. Cette notation commençant par un tiret inférieur a été choisie pour ne pas causer des problèmes avec des programmes déjà écrits. L'inclusion du module `stdbool` permettant d'utiliser l'écriture `bool` comme alternative à `_Bool`, ainsi que les deux constantes booléennes `true` et `false`.

La déclaration d'une variable réserve, dans la mémoire [18], une zone de taille suffisante pour mémoriser un objet du type spécifié. Dans la suite du programme, le nom choisi permettra de faire référence à cette zone mémoire, dans laquelle on pourra donc mémoriser des résultats que l'on rappellera ensuite.

Le compilateur mémorise, le temps de la compilation, le type associé à chacune des variables. Ainsi, il sait, lorsqu'il communique avec la mémoire, comment les données en mémoire doivent être extraites ou inscrites. Les déclarations des variables entières `cherry` et `mango` dans notre programme d'exemple réservent ainsi deux zones mémoires et mémorisent les informations qui leur sont associées de la sorte :



*Mémoire*

Enfin, chaque variable a une *portée*, qui correspond à la zone du programme dans laquelle il est possible d'utiliser la variable. Lorsque l'on sort de cette zone, la zone mémoire réservée pour la variable est automatiquement libérée, et il n'est plus possible d'y faire référence. Il est particulièrement utile que les variables aient une portée limitée pour qu'il ne soit pas nécessaire d'inventer un nom différent à chaque fois.

Les déclarations jouent donc un rôle triple :

- réserver de la place en mémoire pour stocker une information, et libérer cette zone mémoire lorsqu'elle n'est plus utile ;
- conserver les informations sur la façon dont ces données doivent être stockées dans cette zone mémoire (en lien avec le type choisi) ;
- fournir à l'utilisateur un moyen simple de faire référence à cette zone mémoire, grâce au nom choisi.

Le nom de variable n'a que peu d'importance pour le compilateur, mais il est en revanche crucial pour les programmeurs ! Il convient donc de le choisir avec grand soin. Il ne faut pas oublier que l'on passera souvent bien plus de temps à relire et corriger un programme qu'à l'écrire, la programmation est donc aussi un exercice de communication. On évitera par conséquent d'utiliser des noms de variable tels que `v1`, `v2`, `v3`... au profit de noms indiquant clairement la destination (et suggérant possiblement le type) de la variable.

18. Il s'agit d'une petite simplification ici, le compilateur C ayant une obligation de résultat et non de moyens... S'il trouve une autre façon de procéder (telle que conserver la valeur dans les registres du processeur, ou bien effectuer tous les calculs durant la compilation) qui n'ait pas de conséquences sur ce que l'utilisateur lui demande, il a toute latitude de le faire. Quoi qu'il arrive, du point de vue du programmeur, tout se passe comme si une zone de la mémoire a été réservée.

Signalons, pour en terminer, qu'il est possible de déclarer plusieurs variables de même type d'un seul coup, en séparant les différents noms avec des virgules. Ainsi, on aurait pu, de manière équivalente, déclarer les deux variables en une seule fois :

```c
int cherry, mango;
```

## 3.2 Devenir des variables dans les fichiers binaires

Dans le code machine produit par le compilateur, les informations concernant le nom et le type des variables ne sont pas nécessairement conservées [19] : les instructions machines générées par le compilateur tiennent compte du type des données manipulées et utilisent directement leurs adresses en mémoire.

Par exemple, le programme d'exemple peut générer, lors de la compilation, un code machine contenant le passage suivant (à gauche, le code affiché dans un format hexadécimal, à droite sa « traduction » en langage assembleur) :

```
c7 45 fc 06 00 00 00    movl   $0x6, -0x4(%rbp)
8b 45 fc                mov    -0x4(%rbp), %eax
83 c0 01                add    $0x1, %eax
89 45 f8                mov    %eax, -0x8(%rbp)
8b 45 fc                mov    -0x4(%rbp), %eax
0f af 45 f8             imul   -0x8(%rbp), %eax
```

Il n'est pas nécessaire de comprendre en détail ce morceau de langage machine, mais on remarquera que les noms `cherry` et `mango` n'y apparaissent pas. Les expressions `-0x4(%rbp)` et `-0x8(%rbp)` représentent [20] les deux adresses mémoire correspondant aux variables `cherry` et `mango`.

La taille en mémoire des données manipulées est soit renseignée par le mnémonique (le `l` suivant `mov` du premier mnémonique), soit déduit des arguments, par exemple lorsque l'on fait référence au registre EAX.

Ainsi, la première ligne range la valeur 6 à l'adresse correspondant à `cherry`, les trois suivantes rappellent cette valeur dans un registre (ici le registre EAX), lui ajoutent 1 et enregistrent les résultats à l'adresse correspondant à `mango`. Les deux dernières lignes récupèrent la valeur de `cherry` dans le registre EAX et effectuent une multiplication entière avec la valeur de `mango` pour obtenir le résultat du calcul `cherry * mango`, toujours dans ce registre EAX.

19. Elles peuvent l'être toutefois pour faciliter les tests du code produit par le compilateur, dans un processus dit de *déboggage*.
20. Sous une forme qui peut paraître un peu étrange, mais nous l'expliciterons plus tard.

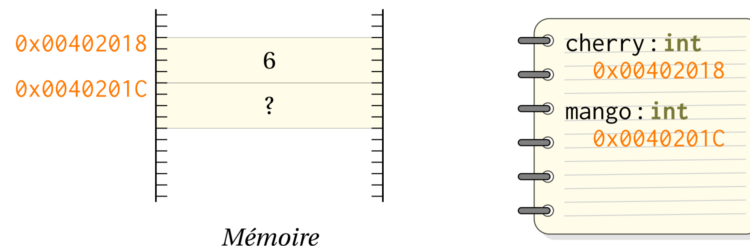
# 4 Utilisation des varibles

## 4.1 Affectation

Une fois les variables déclarées, il est possible de les utiliser, la première étape consistant normalement à placer une valeur à l'emplacement réservé par la déclaration.

Pour mémoriser une valeur dans une variable (on parle d'*affectationaffectation*), on utilise le symbole `=` en plaçant à sa gauche le nom de la variable, et à sa droite la valeur que l'on souhaite y mémoriser.

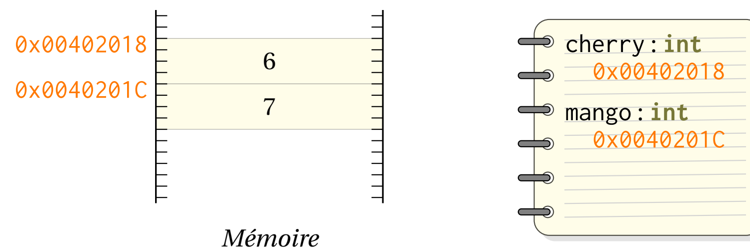
Par exemple, l'instruction « `cherry = 6;` » va conduire à la situation suivante :



*Mémoire*

Lorsqu'un nom de variable apparaît dans une expression, la valeur est extraite de la mémoire afin d'effectuer le calcul. Ainsi, l'expression « `cherry + 1` » correspond donc à `6 + 1`, et est évaluée donc à la valeur 7.

Si à droite du signe `=` on trouve une expression, celle-ci est évaluée et c'est le résultat qui est rangé en mémoire. L'instruction « `mango = cherry + 1` » conduit donc à :



*Mémoire*

Pusiqu'une déclaration de variable conduit à une réservation d'une zone mémoire libre, notons que deux noms de variable font nécessairement *toujours* référence à deux zones distinctes de la mémoire [21]. Ainsi, l'affectation d'une variable n'aura jamais d'effet direct sur le contenu autre variable. Même si l'on écrit « `mango = cherry` », si l'on modifie par la suite la valeur de `cherry` de quelque façon que ce soit, cela n'aura jamais d'impact sur la valeur mémorisée par `mango`.

21. C'est un comportement très différent de langages comme Python ou OCaml, où deux noms peuvent très bien faire référence au même objet, éventuellement temporairement. Nous étudierons par la suite comment un tel comportement est possible.

Signalons de suite une des règles les plus importantes de la programmation en langage C : il est **impératif** d'affecter une valeur à une variable avant de l'utiliser [22] [23].

Pour éviter les maladresses, il est donc préférable d'initialiser une variable aussitôt après sa déclaration. C'est la raison pour laquelle le langage permet de combiner déclaration et initialisation d'une variable en une seule et unique instruction. On aurait ainsi pu remplacer les deux lignes de déclarations et les deux lignes d'initialisation de notre programme d'exemple par [24] :

```
int cherry = 6;
int mango = cherry + 1;
```

### 4.2 Affectations et conversions implicites

Lorsque, dans le cadre d'une affectation, la valeur (ou le résultat de l'évaluation de l'expression) qui se trouve à droite du signe `=` n'a pas le même type que la variable dont le nom se trouve à gauche, il se produira une *conversion* avant que le résultat ne soit rangé en mémoire.

Pour l'instant, les seules conversions raisonnables que l'on puisse envisager sont entre les deux types numériques [26] `int` et `double`. Lorsqu'un entier est converti en flottant, la conversion n'est pas forcément exacte car il peut exister des valeurs de type `int` qui ne sont pas représentables comme `double` [27], et on obtient le flottant immédiatement inférieur ou supérieur [28]. Lorsqu'un flottant est converti en entier, l'éventuelle partie décimale est tronquée (ce qui revient à un arrondi de la valeur vers 0).

Dans tout les cas, la valeur convertie doit impérativement être représentable pour que le comportement du programme soit correct. Convertir une valeur de type `double` supérieure à `INT_MAX` en `int` est par exemple interdit [29].

En cas de besoin, on peut provoquer ce genre de conversion à tout moment en plaçant le nom du type entre parenthèses devant une expression. Ainsi, « `(int)7.8` » s'évalue en la valeur de type `int` 7. Souvent, on évite ces conversions explicites en C lorsqu'elles ne sont pas indispensables. Lorsqu'il s'agit d'arrondir un flottant, on préférera généralement utiliser les fonctions flottantes `floor`, `ceil`, `round` ou `trunc` du module `math`.

Chaque constante écrite dans le fichier source a un type. S'il s'agit d'une suite de chiffres (éventuellement précédée d'un signe plus ou moins), ce sera un `int`, et si on y trouve le séparateur décimal « `.` » et/ou l'indicateur d'exposant, alors il s'agit d'un `double`.

## 5 Expressions

### 5.1 Instructions et expressions

Précisons un instant le vocabulaire que l'on utilisera. On qualifiera d'*instruction* une notation dans le source qui provoque l'exécution par le processeur d'une ou plusieurs commandes. Il s'agit généralement d'un morceau de code entre deux points-virgules, par exemple un affichage avec « `printf("Hello!")` ».

Les *expressions* sont quant à elles généralement construites à partir de constantes et de variables, d'opérateurs et d'éventuels appels de fonction et renvoient un résultat. Par exemple, « `x + floor(y)` » est une expression.

Toutefois, en C, la différence entre les deux est souvent mince. Ainsi, une affectation est naturellement une instruction (elle demande au processeur la mémorisation d'une valeur), mais elle peut également faire office d'expression : elle produit en effet un résultat (la valeur qui a été affectée), qui peut être, sous certaines conditions, directement utilisé dans d'autres calculs. Ainsi, on peut écrire :

```
mango = 2 + (cherry = 3);
```

Dans cet exemple, on place la valeur 3 dans la variable nommée `cherry`, et cette valeur est ensuite ajoutée à la valeur 2 pour obtenir 5, valeur à son tour affectée à la variable nommée `mango`.

Puisque dans une affectation on évalue le membre droit en premier, il est donc tout à fait possible d'écrire :

```
mango = cherry = 37;
```

22. Les conséquences, dans le cas contraire, ne sont ni une erreur, ni une valeur arbitraire qui pourrait se trouver par hasard à cet emplacement mémoire, mais un comportement imprévisible du programme (on parle de comportement indéfini, nous reviendrons sur cet aspect du langage un peu plus tard). Si une variable `n` n'est pas initialisée, alors `n<0 && n>0` (*n* est strictement positif *et* strictement négatif) peut très bien être vrai si cela arrange le compilateur, n'en déplaise aux mathématiques !

23. Nous verrons ultérieurement que dans certains cas particuliers, une variable peut parfois être automatiquement initialisée, mais dans le doute, un excès de prudence ne nuit jamais.

24. Précisons que le nom `cherry` existe théoriquement dès après le « `int cherry` », de sorte que si le compilateur rencontrait l'instruction « `int cherry = cherry + 1` », il n'aurait aucun mal à connaître l'adresse associée au nom `cherry` à droite du signe `=`. Mais le lecteur attentif aura sans doute compris que l'expression « `cherry + 1` » fait usage du nom `cherry` *avant* son initialisation, et que c'est donc une forme à proscrire [25] (si tant est que l'on puisse lui trouver du sens).

25. Bon, l'initialisation automatique de certaines variables peut créer des exceptions, mais gardons cela sous le tapis.

26. Les booléens se trouvant, en réalité, être un type entier, il est théoriquement possible de convertir un booléen en entier ou flottant et inversement. Le programme de CPGE dissuade d'user de telles conversions, donc nous ne nous attaderons pas dessus, précisons cependant que le compilateur n'a aucune raison de protester s'il rencontre une telle conversion.

27. Ce n'est pas le cas, cependant, des architectures que nous utiliserons.

28. Le choix en revient au compilateur, il ne s'agit pas nécessairement de la valeur la plus proche.

29. Il s'agit, pour être exact, d'un autre cas de comportement indéfini.

Dans ce dernier exemple, la valeur 37 est affectée successivement à la variable `cherry` puis à la variable `mango`.

Inversement, rien n'empêche en principe d'utiliser une expression en lieu et place d'une instruction, la valeur calculée sera simplement jetée [30].

## 5.2 Opérateurs arithmétiques

On dispose de plusieurs opérateurs arithmétiques binaires [31] pour effectuer des calculs avec des valeurs numériques. Les quatres opérations élémentaires sont en particulier accessibles avec les opérateurs `+` (pour l'addition, que l'on a déjà croisé), `-` (pour la soustraction), `*` (pour la multiplication) et `/` (pour la division). Comme habituellement en mathématiques [32], les opérandes sont à placer de part et d'autre de l'opérateur.

Par exemple, l'expression [33] « `15 + 22` » est évaluée à 37, l'expression « `12 - 29` » à −17 et l'expression « `3 * 14` » à 42.

Les symboles « `+` » et « `-` » peuvent également servir d'opérateurs unaires (ce qui permet d'écrire « `cherry = -mango` » par exemple). Le comportement est celui auquel on s'attend en mathématiques.

Dans un calcul, le type du résultat dépend des types des opérandes. Pour l'instant, les deux seuls types que l'on manipule sont `int` (entiers) et `double` (flottants). Les règles sont alors simples : si les deux opérandes sont de type `int`, le calcul est effectué sur des entiers, et le résultat est de type `int`. Si en revanche au moins l'un des opérandes est de type `double`, alors les arguments sont convertis en `double` si nécessaire avant le calcul, et le résultat est de type `double`.

Par conséquent, l'opérateur `/` fournit le quotient de la division entière [34] si les deux opérandes sont entiers, et le quotient « usuel », flottant, sinon. Ainsi, l'expression « `11 / 4` » est évaluée à 2 (de type `int`), tandis que les expressions « `11 / 4.0` », « `11.0 / 4` » et « `11.0 / 4.0` » sont toutes les trois évaluées à 2.75 (de type `double`).

Attention, rappellons que si le résultat est ensuite l'objet d'une affectation, il est converti dans le type de la variable qui va l'accueillir. Ainsi :

```c
int banana = 11.0 / 4.0;  // banana ← 2
double pear = 11 / 4;     // pear ← 2.0
```

30. Comme cela peut être involontaire, dans certains cas, le compilateur pourrait bien vous avertir que l'écriture le surprend, mais cela n'empêchera pas la compilation du programme.
31. C'est-à-dire faisant intervenir deux opérandes.
32. Ce n'est pas le cas de tous les langages !
33. Il n'est pas nécessaire de mettre des espaces avant et après l'opérateur, il ne sont présents ici que pour faciliter la lecture.
34. Dans le cas d'une division entière, la partie décimale du résultat est simplement jetée, ce qui conduit à un arrondi à l'entier inférieur si le résultat est positif, à l'entier supérieur si le résultat est négatif.

Si besoin, on peut recourir à des conversions pour obtenir le comportement souhaité. Si `cherry` et `mango` sont deux variables de type `int`, on peut obtenir le quotient usuel en écrivant [35] « `(double)cherry / mango` ». Attention, « `(double)(cherry / mango)` » ne conviendra pas car le calcul (sur les entiers) est effectué *avant* la conversion, comme c'était le cas pour la définition de `pear` dans l'exemple précédent.

Puisque l'opérateur `/` permet d'obtenir le quotient d'une division entière, on dispose de l'opérateur `%` qui fournit lui le reste de la division entière. Il nécessite que les deux opérandes soient entiers. Le résultat de l'expression « `p % q` » est l'entier `r` vérifiant [36] :

- `p=k×q+r` où `k` est un entier relatif ;
- `0≤r<q` si `p` est positif ;
- `q<r≤0` si `p` est négatif.

Il n'existe pas d'opérateur d'exponentiation. Pour le calcul de $x^y$ il faudra faire appel à la fonction `pow(x, y)` du module `math` [37] (qui transformera si besoin ses arguments en valeurs de type `double` et renvoie une valeur de ce même type). Ce même module contient un grand nombre de fonctions, telles que les fonctions trigonométriques `sin`, `cos` et `tan` et leurs inverses `asin`, `acos` et `atan` [38], la racine carrée `sqrt`, le logarithme *naturel* `log` et ses variantes binaire `log2` et décimale `log10`, l'exponentielle `exp`, les fonctions d'arrondi évoquées précédemment et bien d'autres !

Pour chaque opérateur arithmétique, il existe par ailleurs un opérateur d'assignation augmenté*opérateur d'assignation augmenté* qui effectue une opération arithmétique suivie d'une affectation. Ainsi, « `a += b` » est un raccourci pour « `a = a+b` ». Le principe est le même pour « `-=` », « `*=` », « `/=` » et « `%=` ». Comme une affectation, une assignation augmentée est aussi une expression, dont la valeur est le résultat de l'opération (donc la valeur assignée). Le but recherché ici est la concision (on évite d'écrire deux fois le nom de variable). Ce n'est pas le seul « raccourci » que le langage met à notre disposition, nous en verrons d'autres un peu plus loin.

## 5.3 Opérateurs de comparaison

On dispose également d'opérateurs de comparaison, permettant de construire des valeurs booléennes en comparant des valeurs numériques. Les opérateurs « `<` », « `<=` », « `>` » et « `>=` » permettent de déterminer si une valeur est plus petite ou plus grande qu'une autre. L'opérateur « `==` » permet de tester l'égalité entre deux valeurs numériques ou booléennes [39]. Enfin, « `!=` » permet de tester si deux valeurs numériques ou booléennes

35. Il est inutile de convertir les deux opérantes en écrivant par exemple« `(double)cherry / (double)mango` », la présence d'un seul argument flottant étant suffisant.
36. On se méfiera de l'opérateur `%` dont le comportement varie d'un langage à un autre lorsque l'on travaille avec des entiers négatifs. En C, le signe du résultat correspond au signe de l'opérande de gauche, tandis que dans un langage comme Python, il s'agit du signe de l'opérande de droite !
37. On adjoindra au fichier source la directive « `##include <math.h>` ».
38. Ainsi qu'une fonction `atan2` déterminant un angle dont on connaît le sinus et le cosinus.
39. Les autres opérateurs de comparaison acceptent également des booléens, mais cela n'a guère de sens en mathématiques et résulte d'un choix d'implémentation, aussi laisserons cette possibilité de côté.

ne sont *pas* égales. Ainsi, « `42 < 37` », « `42 <= 37` » et « `42 == 37` » sont évaluées à `false`, « `42 > 37` », « `42 >= 37` » et « `42 != 37` » à `true`.

Les valeurs de part et d'autre de l'opérateur n'ont pas nécessairement à être de même type, elles subiront une conversion avant la comparaison si nécessaire, en suivant les mêmes règles que pour les opérateurs arithmétiques. C'est à dire que si l'on compare une valeur de type `int` et une valeur de type `double`, la première sera convertie en une valeur de type `double` préalablement à la comparaison.

Signalons que, puisque le résultat de la comparaison est un booléen, l'expression « `a < b < c` » n'a pas le sens qu'on peut lui donner en mathématiques [40].

## 5.4 Opérateurs logiques

On dispose également de deux opérateurs logiques binaires et d'un opérateur logique unaire. Il s'agit :

- de l'opérateur binaire « `&&` » qui traduit un « et » logique (le résultat est `true` si et seulement si les deux opérandes sont évalués à `true`) ;
- de l'opérateur binaire « `||` » qui traduit un « ou » logique (le résultat est `true` si et seulement si au moins un des deux opérandes est évalué à `true`) ;
- de l'opérateur unaire « `!` » qui traduit la négation de l'expression placée à sa droite.

## 5.5 Précédence

Toutes ces opérations arithmétiques et logiques, comparaisons et affectations obéissent à des règles de priorité, dont une petite partie est présentée ci-dessous.

| | opération | associativité |
|---|---|---|
| `+ - !`<br>`(type)` | plus/moins/négation unaire<br>conversion | de la droite vers la gauche |
| `* / %` | multiplication/division | de la gauche vers la droite |
| `+ -` | addition/soustraction | de la gauche vers la droite |
| `< <= > >=` | comparaisons | de la gauche vers la droite |
| `== !=` | égalité/non égalité | de la gauche vers la droite |
| `&&` | « et » logique | de la gauche vers la droite |
| `\|\|` | « ou » logique | de la gauche vers la droite |
| `= +=` ... | affectation et affectations augmentées | de la droite vers la gauche |

40. Elle a pourtant un sens, car nous verrons qu'il n'est pas illégal de comparer un booléen et une valeur numérique, même si nous éviterons *soigneusement* de le faire !

Les opérations situées dans les cases élevées du tableau sont effectuées en priorité par rapport à celles qui sont situées en-dessous. Les multiplications, par exemple, sont prioritaires sur les additions, comme on s'y attends généralement en mathématiques.

L'instruction suivante détermine par exemple si l'entier $n$ est divisible par 2 mais pas par 3 et range le résultat dans la variable booléenne `b` déclarées pour l'occasion :

```
bool b = n % 2 == 0 && n % 3 != 0;
```

En effet, on commence par créer la variable `b`, puis on évalue l'expression de droite en calculant les restes des divisions entières, puis en les comparant à zéro, ce qui donne deux booléens, lesquels sont ensuite combinés avec l'opérateur logique « et ». Le résultat de l'évaluation de l'expression, booléen, est affecté à la variable `b`.

On y trouve également dans le tableau les règles d'associativité des différents opérateurs. Ainsi, lorsque l'on écrit « `a - b - c` », le calcul s'effectue de la gauche vers la droite. On calcule donc « `a - b` », puis on retire `c` au résultat.

Si l'ordre d'évaluation ne convient pas, on peut librement user de parenthèses « `(` » et « `)` » afin de modifier l'ordre dans lequel les éléments sont évalués.

Précisons que si le langage précise fort heureusement dans quel ordre les différents opérateurs sont appliqués, il ne dit rien, en général, sur l'ordre d'évaluation des opérandes. Dans une addition, le compilateur peut librement choisir s'il évalue d'abord l'expression à gauche du signe `+` puis celle à droite ou l'inverse [41]. Dans certains cas, cela peut avoir de l'importance.

Il y a cependant une exception à cette règle : les opérateurs binaires booléen évaluent toujours l'opérande de gauche avant l'opérande de droite, car s'il permet de conclure [42], le second opérande ne sera pas évalué, toujours pour des raisons d'efficacité. Cette stratégie est qualifiée d'*évaluation paresseuse*.

41. Il pourrait même, théoriquement, effectuer les évaluations de concert !

42. Si l'opérande de gauche pour `&&` est évalué à `false`, le résultat sera nécessairement `false`, quelle que soit l'expression à droite. De même, si l'opérande de gauche pour `||` est évalué à `true`, le résultat sera nécessairement `true`.

# 2 Structures de contrôle en C

« Beware of bugs in the above code; I have only proved it correct, not tried it. »

— *Donald Knuth*

## 1 Structure conditionnelles

### 1.1 Exécution conditionnelle avec « `if` »

Il est très fréquent en informatique d'avoir besoin d'exécuter un morceau de code uniquement si une condition est vérifiée. On parle d'*exécution conditionnelle*.

Pour ce faire, on dispose en langage C du mot-clé `if` qui s'utilise de la sorte :

```c
if (condition) instruction;
```

`condition` doit être une expression qui sera évaluée et donner un booléen. Si le résultat de cette évaluation est `true`, alors `instruction` est exécutée. Dans le cas contraire, `instruction` est ignorée. Par exemple, le programme suivant affiche un message si l'entier `mango` est pair [1 2 3] :

```c
int mango;
...
if (mango%2 == 0)
    printf("n est pair !\n");
```

1. Rappelons que retour à la ligne et l'indentation ne sont là que pour faciliter la lecture, ils ne sont pas requis par le langage.
2. Il est d'usage de vérifier la parité, ou de façon plus générale la divisibilité d'un entier par un autre entier, en contrôlant le reste d'une division entière, effectuée à l'aide de l'opérateur `%`. Si d'autres solutions, plus exotiques, sont théoriquent possibles, par exemple « `mango/2*2 == mango` », pour une bonne compréhension du code, il est plus que vivement conseillé de s'y tenir! En particulier, on évitera à tout prix de faire intervenir des flottants et des arrondis.
3. Une valeur *doit impérativement* avoir été affectée à la variable `mango` dans la partie éludée du programme préalablement au test, sinon il s'agit d'un cas de comportement indéfini. Il en est de même pour la variable `banana` dans l'exemple suivant.

Précisons que les parenthèses autour de la condition sont requises, même s'il s'agit simplement de vérifier le contenu d'une variable booléenne. Ainsi, on écrira :

```c
bool banana;
...
if (banana)
    printf("banana contient la valeur true\n");
```

Profitons de l'occasion pour signaler qu'il ne serait guère bienvenu d'écrire la condition dans l'exemple précédent « `if (banana == true)` ». En effet, banana étant déjà un booléen, la comparaison ne fait qu'alourdir l'écriture. La condition « `if (banana == false)` » pourrait éventuellement être considérée, mais on préférera de *très loin* utiliser la négation booléenne et écrire « `if (!banana)` ».

### 1.2 Précision sur les tests en langage C

Avant de continuer, prenons un instant pour préciser quelques points du langage à même de jouer de mauvais tours. Le langage C original ne contenait pas la notion de booléens. Dans un test, il suffit en fait que la condition puisse être convertie en une valeur numérique. Une valeur nulle sera considérée comme `false`, et une valeur non-nulle comme `true`. Nous verrons que les booléens ne sont en fait que des valeurs numériques un peu particulières.

Combinée avec le fait que les affectations sont des expressions, cela peut occasionner de très vicieuses erreurs. Par exemple, pour vérifier si un entier `mango` est nul, on écrira par exemple, correctement :

```c
if (mango == 0)
    printf("la variable mango contient 0\n");
```

Mais que se passe-t-il si on écrit, par erreur :

```c
if (mango = 0)
    printf("la variable mango contient 0\n");
```

Le programme précédent est parfaitement correct : il affecte 0 à la variable `mango`, et considère 0 comme le résultat de l'expression, ce qui est considéré comme « faux ». Par conséquent, l'affichage n'est jamais effectué, quelle que soit la valeur initialement contenue dans la variable `mango`, et cette valeur est remplacée par 0 !

L'erreur est suffisamment fréquente [4] pour que le compilateur soit tenté d'afficher un avertissement lors de la compilation s'il rencontre ce genre d'écriture (qui peut parfois être ce que l'on souhaite écrire). C'est également une raison supplémentaire d'éviter toute

4. *A fortiori* quand l'autre langage que vous manipulerez cette année utilisera `=` pour les comparaisons !

comparaison lorsque l'on dispose d'une variable contenant déjà une valeur booléenne! De même, ne vous étonnez pas que certains programmeurs emploient une notation que l'on qualifie parfois de « condition Yoda » :

```
if (0 == mango)
    printf("la variable mango contient 0\n");
```

En effet, cette seconde écriture, si l'on oublie par mégarde un des deux signes =, ne saurait être interprétée comme une affectation puisque l'on ne trouve pas un nom à gauche du signe =, aussi aura-t-on une erreur lors de la compilation.

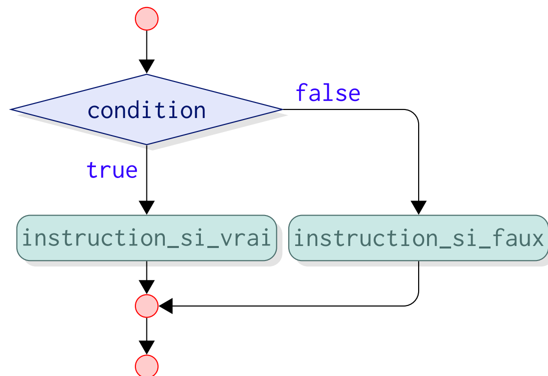
Dans ce cours, nous nous efforcerons de toujours utiliser des expressions explicitement booléennes comme conditions, à quelques rares exceptions près, idiomatiques, sur lesquelles nous reviendrons.

### 1.3 Alternative avec « `else` »

Revenons à nos instructions conditionnelles. On peut également souhaiter exécuter des instructions différentes selon que la condition est vraie ou non. On utilisera alors[5] :

```
if (condition)
    instruction_si_vrai;
else
    instruction_si_faux;
```

Le comportement de cette construction est résumé dans l'organigramme ci-dessous. Selon la valeur booléenne obtenue par l'évaluation de la condition, on effectue l'une *ou* l'autre des deux instructions.




5. On aurait pu présenter les choses différemment, puisque le langage C ne se préoccupe pas des retours à la ligne ou de l'indentation. Seule la position des mots-clés `if` et `else` et des points-virgules a de l'importance. On préfère toutefois, on l'a dit, respecter des règles d'indentation pour faciliter la lecture du programme.

Prenons par exemple le cas de la suite $(u_n)_{n \in \mathbb{N}}$ dite de Syracuse, définie par la relation de récurrence

$$u_{n+1} = \begin{cases} u_n/2 & \text{si } u_n \text{ est pair} \\ 3u_n + 1 & \text{si } u_n \text{ est impair} \end{cases}$$

Si on choisit par exemple $u_0 = 42$, on aura $u_1 = 21$, $u_2 = 64$, $u_3 = 32$, $u_4 = 16$, $u_5 = 8$ et ainsi de suite.

Cette suite est populaire en raison de la conjecture de Collatz (qui n'a toujours pas été démontrée à l'heure actuelle[6]) affirmant que, quel que soit l'entier strictement positif choisi pour $u_0$, il existe toujours un $k \in \mathbb{N}$ tel que $u_k = 1$ (les termes suivants formant alors une séquence $1 \mapsto 4 \mapsto 2 \mapsto 1$... qui se répête infiniment).

Dans un programme où une variable u désignerait un terme de la suite, on pourrait passer d'un terme à celui qui le suit avec ce morceau de code :

```
// la variable nommée u contient la valeur u_n
if (u%2 == 0)
    u = u/2;
else
    u = 3*u+1;
// la variable nommée u contient la valeur u_{n+1}
```

### 1.4 Grouper les instructions en « blocs »

Cette écriture élémentaire des tests peut suffire, mais elle présente quelques difficultés. La première apparaît lorsque l'on veut exécuter *plusieurs* instructions comme conséquence d'un test.

Pour s'en sortir, on va créer un *bloc* d'instructions. Un bloc d'instructions est une séquence (potentiellement complexe) d'instructions regroupées de façon à se comporter comme une instruction unique.

En langage C, on représente un bloc en l'encadrant avec des accolades « { » et « } ». Ainsi, on peut écrire :

```
if (u%2 == 0)
    { printf("u est pair\n"); u = u/2; }
else
    { printf("u est impair\n"); u = 3*u+1; }
```

6. En tout cas de façon admise par la majorité des personnes familières avec la quesiton, même si c'est un sujet suffisamment populaire pour que divers auteurs affirment en avoir établi une preuve.

Si u contient un entier pair, alors on effectuera le premier l'affichage et on divisera la valeur de u par deux, et dans le cas contraire on effectuera le second affichage et on triplera puis incrémentera la valeur de u.

Comme l'indentation, le positionnement exact des accolades encadrant les instructions d'un bloc n'a pas d'importance et n'est qu'une question de style d'écriture. En se basant sur le style popularisé par B. Kernighan et D. Ritchie, et en adoptant une écriture privilégiant une instruction par ligne, on préférera souvent écrire :

```
if (u%2 == 0) {
    printf("u est pair\n");
    u = u/2;
} else {
    printf("u est impair\n");
    u = 3*u+1;
}
```

L'usage de blocs permet de résoudre une autre difficulté, qui apparaît lorsque l'on tente d'imbriquer des tests. Considérons par exemple :

```
if (condition_1)
    if (condition_2)
        instruction_A;
    else
        instruction_B;
```

Le **else** fait-il référence au premier **if**, ou comme le laisse penser l'indentation, au second ? On rappelle que le langage ignore l'indentation !

La règle utilisée par le langage est simple : un **else** fait normalement référence au **if** le plus proche (dans le même bloc). L'indentation correspond donc bien ici au fonctionnement, et `instruction_B` sera exécutée si et seulement si, à la fois, `condition_1` est vraie et `condition_2` est fausse.

Pour plus de sécurité et de clarté, on peut se servir de blocs et écrire les choses ainsi :

```
if (condition_1) {
    if (condition_2) {
        instruction_A;
    } else {
        instruction_B;
    }
}
```

En effet, le **else** ne peut se rapporter qu'à un **if** situé dans le même bloc, et seul le second **if** se trouve dans le même bloc que le **else**.

Si on souhaite que le **else** se rapporte au premier **if**, c'est parfaitement possible en se servant de blocs d'instructions et en écrivant :

```
if (condition_1) {
    if (condition_2) {
        instruction_A;
    }                    // Fin du second "if"
} else {                 // Se rapporte au premier "if"
    instruction_B;
}
```

L'utilisation de blocs lorsqu'il n'y a qu'une seule instruction peut sembler excessive. C'est toutefois une précaution que l'on prend fréquemment, pour éviter la situation potentiellement problématique où l'on ajoute une seconde instruction à la suite de la première en oubliant d'ajouter les accolades pour former un bloc !

Quoi qu'il en soit, il s'agit de situations où une discipline stricte sur l'indentation est plus que bienvenue, si l'on veut faciliter la compréhension du code que l'on écrit.

## 1.5 Tests en cascade

Le fait que **else** fasse référence au **if** le plus proche permet d'écrire des tests en cascade sans avoir besoin d'imbriquer de nombreux blocs, simplement en faisant suivre [7] un **else** d'un **if**.

On testera par exemple différents cas (nullité, parité, signe) pour une variable entière mango de la sorte, où l'instruction exécutée est celle suivant le *premier* test validé :

```
if (mango == 0) {
    printf("mango est nul\n");
} else if (mango%2 == 0) {
    printf("mango est un entier pair non nul\n");
} else if (mango > 0) {
    printf("mango est un entier impair strictement positif\n");
} else {
    printf("mango est un entier impair strictement négatif\n");
}
```

7. Comme le langage n'a pas de contraintes d'indentation, il n'existe pas de mot-clé « **elif** » comme c'est le cas en Python par exemple.

## 2 Blocs d'instructions et portée des variables

### 2.1 Principe

Nous avons laissé entendre, tantôt, que les variables ont une *portée*, c'est-à-dire que la zone mémoire qui a été réservée finit par être libérée après un certain temps. Cette portée est directement liée au bloc dans lequel la variable a été déclarée : sa portée s'étend jusqu'à la fin du bloc (ou du programme si elle a été déclarée à l'extérieur de tout bloc). Il n'est donc possible d'accéder aux variables que si elles ont été déclarées *préalablement* dans le bloc courant, dans un bloc qui l'englobe, ou hors de tout bloc.

Afin de mieux illustrer la notion de portée, considérons l'ébauche de programme ci-dessous :

```c
...                             // ①

int mango = 1;                  // ②

...                             // ③

int main(void) {
    ...                         // ④
    int cherry = 2;             // ⑤
    ...                         // ⑥
    int apple = 3
    ...                         // ⑦
    if (...) {
        ...                     // ⑧
        int plum = 4;
        ...                     // ⑨
    }                           // ⑩
    ...
    return 0;
}                               // ⑪

...                             // ⑫
```

En ①, aucune des quatre variables entières nommées `cherry`, `apple`, `mango` et `plum`, mentionnées dans le programme, n'a encore été déclarée, et n'est donc utilisable.

La toute première variable, `mango`, est déclarée (et définie) sur la ligne ②, à l'extérieur de tout bloc. Elle est ainsi utilisable, en lecture comme en écriture, de sa définition à la fin du fichier, comme illustré ci-dessus. La mémoire qui lui est associée ne sera libérée que lorsque le programme se terminera. On parle de variable *globale*.

À l'intérieur du bloc définissant `main`, la variable `mango`, définie à l'extérieur de ce bloc, est donc tout à fait accessible. Par conséquent, il serait possible de s'en servir dans la déclaration de `cherry` sur la ligne ④ (« `cherry = mango+1` » serait parfaitement correct).

La variable `cherry` est utilisable de sa déclaration jusqu'à la fin du bloc. Sa portée s'étend donc de la ligne ⑤ à la ligne ⑪ (plus précisément, de la mention de son nom sur la ligne ⑤ à l'accolade fermante de la ligne ⑪), et on peut donc la manipuler dans cette partie du code source. On qualifie généralement en informatique une telle variable de *locale*.

Lorsque l'on se trouve en ⑥, les variables `mango` et `cherry` sont utilisables, mais pas la variable `apple`, qui n'est définie que plus tard. Les trois variables sont en revanche manipulables lorsque l'on se trouve en ⑦. La portée de la variable `apple` s'étend également jusqu'à l'accolade fermante de la ligne ⑪.

En ⑧, ces trois mêmes variables sont disponibles, mais pas encore la variable `plum`, qui le deviendra après sa déclaration, dans la zone ⑨.

En ⑩, on atteint la fin de la portée de la variable `plum`, et la mémoire qui lui avait été attribuée est réservée. Dans les lignes qui suivent, il n'est plus possible d'y faire référence (la compilation provoquera une erreur disant que la variable `plum` est inconnue).

Le même sort est réservée à `cherry` et `apple` sur la ligne ⑪, de sorte que s'il y a du code au-delà, dans la zone ⑫, seule la variable `mango` peut encore être utilisée.

Il est possible de créer un bloc d'instruction à tout endroit où l'on attend une instruction, ils ne sont pas réservés aux structures conditionnelles telles que le **`if`** .... On peut donc créer un bloc pour restreindre volontairement la portée d'une variable, par exemple pour un résultat de calcul intermédiaire que l'on ne souhaiterait pas conserver dans la suite.

### 2.2 Un peu plus loin

Pour être parfaitement juste, il peut exister une différence entre la zone du programme où une variable est utilisable et celle où elle a une existence en mémoire, et il arrive en C qu'une variable *existe* en mémoire sans toutefois être accessible.

C'est notamment le cas des variables globales en C, qui ont une existence en mémoire durant toute la durée du programme. `mango` existe donc déjà en mémoire dès la ligne ①, même s'il n'est pas possible encore d'y faire référence.

Signalons qu'il existe d'autres situations similaire (hors programme en MP2I/MPI), par exemple dans le cas en C de variables dites *statiques*[8], déclarées à l'intérieur d'un bloc avec le mot-clé « **`static`** », qui bien qu'étant déclarées dans un bloc, sont des variables globales, mais visibles uniquement à l'intérieur du bloc où elles ont été déclarées.

8. Le terme « statique » a plusieurs sens et usages en C, ce qui ne simplifie pas les choses.

## 2.3 Occultation de variables

Deux variables déclarées dans le même bloc ne peuvent jamais porter le même nom. En revanche, il est possible de déclarer des variables de même nom dans des blocs distincts[9]. Si ces blocs sont imbriqués, alors il est possible de se retrouver en un endroit du programme à l'intérieur de la portée de deux variables de même nom !

Dans ce cas, le nom désigne la variable déclarée le plus récemment, ou, de façon complètement équivalente, celle dans le bloc le plus interne. On parle de *masquage* ou d'*occultation* de variable (*variable shadowing* en anglais). Considérons par exemple l'exemple suivant :

```
... {
    ...                         // ①
    int mango = 37;             // ②
    ...                         // ③
    ... {
        ...                     // ④
        int mango = 42          // ⑤
        ...                     // ⑥
    }                           // ⑦
    ...                         // ⑧
}                               // ⑨
```

La variable déclarée à la ligne ② a une portée qui s'étend jusqu'à la ligne ⑨. La variable déclarée à la ligne ⑤ a une portée qui s'étend jusqu'à la ligne ⑦.

Lorsque l'on se trouve en ①, aucune variable `mango` n'a encore été déclarée, et le nom n'est donc pas valide. En ③, c'est naturellement à la variable déclarée ligne ② que le nom `mango` fait référence. C'est encore le cas ligne ④, même si on est entré dans un bloc[10].

En revanche, à partir de la déclaration de la ligne ⑤, deux variables de même nom coexistent. Celle déclarée ligne ⑤ va masquer celle déclarée ligne ②, et en ⑥, c'est donc à la variable déclarée ligne ⑤ que le nom `mango` fait référence.

Enfin, lorsque l'on se trouve en ⑧, on est au-delà de la portée de la variable déclarée ligne ⑤. Il n'y a donc plus de conflit, et le nom `mango` fait à nouveau référence à la variable déclarée ligne ②.

Bien que ce mécanisme puisse parfois être utile, dans la mesure du possible, on évitera de définir des variables de même nom. Cela rend fréquemment les programmes plus difficiles à lire et peut conduire à des problèmes difficiles à localiser.

9. Ou bien dans un bloc et à l'extérieur de tout bloc, avec une portée s'étandant à l'intégralité du programme.
10. Sur ce point, on notera que le comportement du C diffère de Python.

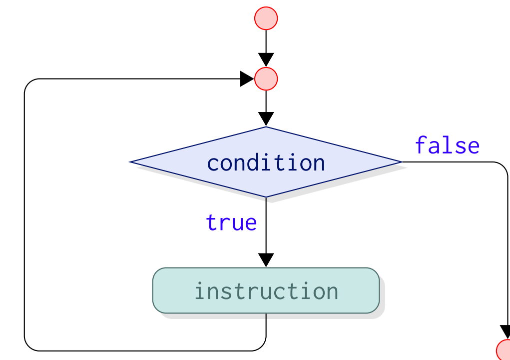
# 3 Boucles conditionnelles

## 3.1 Boucles « `while` »

Lorsque l'on écrira des programmes, on cherchera également fréquemment à répéter un morceau de code plusieurs fois de suite. La première façon de le faire est d'utiliser une boucle `while`, qui exécute une instruction tant qu'une condition booléenne est vraie. Sa syntaxe, fort semblable à celle d'un test conditionnel, est la suivante :

```
while (condition) instruction;
```

Le comportement de cette structure de contrôle peut être résumée par l'organigramme ci-dessous :




Par exemple, pour ajouter 17 à une variable `mango` autant de fois que nécessaire pour qu'elle devienne supérieure ou égale à 42, on peut écrire[11] :

```
int mango;
...
while (mango < 42) mango = mango + 17;
```

Comme pour les tests conditionnels, on utilisera généralement un bloc d'instructions (même s'il s'agit d'une instruction unique), et on écrira plutôt :

```
while (mango < 42) {
    mango = mango + 17;
}
```

Signalons deux aspects important de ces boucles conditionnelles. Tout d'abord, on commence par évaluer l'expression booléenne représentant la condition. Si cette évaluation

11. Comme précédemment, on supposera que la variable `mango` s'est vu affecter une valeur dans la partie éludée du programme, sinon la comparaison serait une situation de comportement indéfini !

initiale donne `false`, alors l'instruction ou le bloc d'instructions associé au `while` est totalement ignoré. Ensuite, cette évaluation booléennes n'a lieu qu'entre deux exécutions du bloc. Si la condition devient vraie durant une exécution du bloc, cela ne conduit aucunement à l'arrêt immédiat de l'exécution du bloc, et si elle redevient fausse avant la fin du bloc (et l'évaluation de la condition), alors une itération supplémentaire sera effectuée !

Dans quelque rares cas, il est possible que l'on n'ait pas le besoin d'exécuter d'instruction à la suite du test, auquel cas on peut écrire

```
while (condition);
```

ou bien [12]

```
while (condition) {}
```

Dans ce cas, le programme va évaluer en boucle la condition jusqu'à ce qu'elle soit vraie. Bien évidemment, cela n'a de sens que si la condition est liée à un événement extérieur (une entrée de l'utilisateur) ou si l'évaluation de `condition` modifie d'une manière ou d'une autre l'état de la mémoire, de sorte que le résultat puisse passer, après un certain temps, de `false` à `true` !

Attention toutefois, la condition sera évaluée répétitivement aussi vite que le processeur le pourra. S'il s'agit d'attendre une entrée de l'utilisateur, ce n'est pas très sage, sauf si le but est de faire chauffer le processeur ! Dans ce genre de situation, il est fréquent d'avoir une instruction dans la boucle qui dit au processeur « suspend l'exécution du programme pendant quelques (milli)secondes [13] » pour réduire la fréquence de l'évaluation. Toutefois, si la condition n'est que très momentanément vraie (issue d'un capteur, par exemple), il y a un possible risque de manquer le moment où elle le devient [14].

## 3.2 Quelques exemples concrets

**Suite de Syracuse**

Revenons à notre suite de Syracuse pour un exemple un peu plus élaboré. Supposons que l'on choisisse $u_0 = 17$, et que l'on souhaite afficher tous les termes de la suite de $u_0$ à $u_k$ inclus, où $k$ est tel que $u_k = 1$ soit la première apparition de 1 dans la suite.

12. Si l'on a *réellement* besoin d'une itération « vide », cette seconde option est souvent plus facile à lire car on a vite fait de manquer le point-virgule, et donc à privilégier dans le cadre des concours, mais dans la pratique, la première écriture est fréquemment utilisée.
13. S'il y a un système d'exploitation dans la machine, ce temps sera naturellement utilisé pour réaliser d'autres tâches et poursuivre l'exécution d'autres programmes.
14. Cela peut théoriquement même arriver lorsque les évaluations se font au maximum de la vitesse du processeur, s'il s'agit de détecter de façon fiable un événement très bref, il faudra envisager d'autres mécanismes.

Afin d'éviter d'alourdir la fonction, on suppose que l'on peut passer d'un terme de la suite au suivant en écrivant « `u = next(u)` » (nous verrons un peu plus loin comment rendre cette écriture correcte).

Une possible solution pour afficher l'ensemble des termes serait :

```
int u = 17;            // u ← u0

while (u != 1) {
    printf("%d", u)
    u = next(u);
}
printf("%d", u);       // on affiche uk (autrement dit 1)
```

Il convient toujours d'être très prudent lorsque l'on écrit une boucle, on a tôt fait d'oublier un élément ou d'avoir un élément en trop. Dans l'exemple précédent, on affiche bien $u_0$ puisqu'il y a un `printf` avant la première mise à jour de `u`. En revanche, la boucle n'affichera pas $u_k$ car on sort de la boucle avant de le faire. Il faut donc traiter l'affichage de $u_k$ à part.

Voici une autre possibilité que l'on peut envisager :

```
int u = 17;            // u ← u0

printf("%d", u);       // on affiche u0
while (u != 1) {
    u = next(u);
    printf("%d", u)
}
```

Dans ce second cas, le dernier terme $u_k$ est bien affiché dans la boucle, mais c'est le terme $u_0$ qui doit être traité à part.

Dans chacun des deux programmes proposés, on peut également vérifier que si l'on choisit $u_0 = 1$, alors il n'y a bien que le terme $u_0$ qui sera affiché.

**Extraction des chiffres d'un entier**

Considérons un autre exemple, dont nous utiliserons fréquemment des variations. Supposons que l'on dispose d'un entier $n$ *strictement* positif, par exemple 1492, et que l'on souhaite effectuer quelque chose (par exemple un affichage) avec chacun des chiffres composant son écriture décimale.

Les opérateurs arithmétiques nous fournissent un moyen simple d'obtenir le chiffre des unités. En effet, l'opérateur « `%` » donne le reste d'une division entière, de sorte que « `n % 10` » fournit directement le chiffre des unités. Ainsi, « `1492 % 10` » donne 2.

Pour poursuivre, il nous faut un moyen de « gommer » ce 2, qui aura été traité. Cela se fait au moyen d'une division entière par 10, ce qui en C s'écrit s'implement « `n / 10` » lorsque `n` est un entier. On obtient 149 lorsque `n` contient par exemple 1492.

On continuera de procéder ainsi tant que `n` est non nul, ce qui permettra d'obtenir tour à tour tous les chiffres [15] de l'écriture décimale de $n$, considérés de gauche à droite, soit 2, 9, 4 et 1 pour 1492. Le programme peut par exemple être :

```c
int n = 1492;

while (n != 0) {
    printf("%d ", n % 10);
    n = n / 10;
}
printf("\n");     // Un retour à la ligne à la fin
```

Notons que ce principe s'étend à la détermination des chiffres d'un entier strictement positif dans une base $b > 1$ quelconque, il suffira d'utiliser « `n % b` » et « `n / b` ». Dans la mesure où la décomposition binaire d'un entier est souvent utile, ce morceau de programme servira fréquemment avec `b`= 2.

Obtenir les chiffres de gauche à droite est bien plus difficile, et donc à réserver aux cas où c'est nécessaire. On peut par exemple envisager de trouver la puissance de dix strictement supérieure à `n`, puis revenir sur les puissances inférieures, en utilisant les mêmes opérateurs de division entière [16]. À des fins d'illustrations, on peut proposer :

```c
int n = 1492;

int m = 1;
while (m <= n) {
    m = m * 10;
}
// m est la plus petite puissance de 10 strictement supérieure à n
while (m > 1) {
    m = m / 10;
    printf("%d ", n / m);
    n = n % m;
}
printf("\n");
```

15. Précisons que si `n` est nul, le programme afficherait une ligne vide, et si `n` est strictement négatif, compte tenu du fonctionnement de l'opérateur `%` en C, les chiffres affichés seraient chacun précédés du signe « `-` ».
16. Avec des listes, des tableaux ou des piles, on pourra également envisager de renvoyer le résultat de l'algorithme précédent !

## 3.3 Variation sur le thème

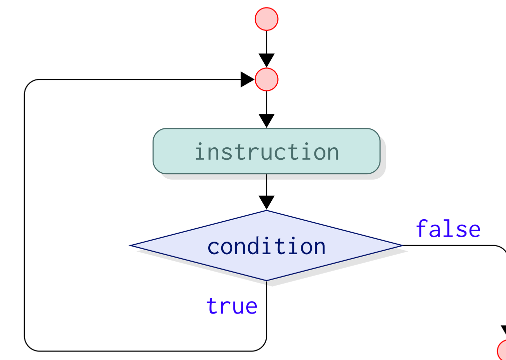
Signalons qu'il existe une autre façon d'écrire des boucles conditionnelles, mais qui n'est pas au programme de CPGE. Il s'agit de la construction

```c
do instruction while (condition);
```

qui peut naturellement être utilisée avec un bloc d'instructions :

```c
do {
    instructions;
} while (condition);
```

Le fonctionnement est le même que dans le cas d'une boucle **`while`** classique, mais la condition est évaluée à la *fin* de la boucle plutôt qu'au début de celle-ci, comme illustré sur l'organigramme suivant :




Une conséquence est que exécute toujours au moins une fois le contenu de la boucle ! Dans certains cas, cela peut simplifier l'écriture. Par exemple, dans le cadre de la détermination des chiffres de l'écriture décimale d'un nombre, cette construction permettrait, en réécrivant la boucle comme ci-dessous, d'afficher un (unique) 0 dans le cas où $n = 0$, tout en donnant les mêmes résultats pour les entiers $n$ strictement positifs.

```c
do {
    printf("%d ", n % 10);
    n = n / 10;
} while (n != 0)
```

Cette construction peut cependant parfois être source de malentendus, aussi s'efforce-t-on en général de l'éviter, et cela explique son absence du programme.

# 4 Écrire des boucles correctes

## 4.1 De l'importance et du bon usage des commentaires

Rien que sur l'exemple précédent, qui n'est pas bien complexe, on voit qu'il est aisé d'écrire un programme incorrect, qui oublie une opération ou en effectue une de trop. Bien évidemment, on veut s'efforcer d'écrire des programmes corrects. Mais en fait, on s'efforcera d'aller plus loin. Le fait que le programme fasse ce qu'il est sensé faire doit apparaître comme une évidence à sa lecture. Encore une fois, il ne faut pas oublier qu'un programme sera potentiellement relu, maintenu et modifié pendant des années, et que les intentions de l'auteur du code original doivent être claires pour celui qui devra le relire ou l'altérer.

Les commentaires jouent un rôle crucial dans cette communication. Ils ne doivent pas paraphraser le code : si l'on écrit « `mango = 0` », il est inutile de mettre un commentaire indiquant que l'on mets 0 dans la variable `mango`, car le code est clair sur ce point [17].

Le commentaire peut en revanche apporter des éclaircissement sur l'usage qui sera fait de la variable (si le choix d'un nom adapté pour la variable ne suffit pas à le faire) et éventuellement sur la raison pour laquelle on mets un zéro dedans à cet endroit du programme si elle est subtile. Commenter un programme de façon pertinente [18] peut être un exercice plus difficile encore que d'écrire le programme lui-même.

## 4.2 Invariants

Une des manières de convaincre le lecteur [19] du bon fonctionnement d'une partie du programme est de déterminer des *invariants*.

**Définition.** Un *invariant* est une affirmation, exprimée en langage naturel ou mathématique, associée à un point précis du programme qui doit être vraie à chaque passage en ce point du programme.

La façon la plus simple de documenter un invariant est de l'inclure comme un commentaire dans le programme. Ainsi, on règle directement la question du point du programme auquel il est associé !

Comme les boucles sont des sources fréquentes d'erreurs et des endroits du programme où l'on passe à de nombreuses reprises, elles sont un lieu privilégié pour inclure des invariants. On parlera d'*invariants de boucle*.

Supposons que nous cherchons à déterminer, pour notre suite de Syracuse, le plus petit $k$ tel que $u_k = 1$ pour un $u_0$ donné. Le programme, avec des invariants en commentaires, pourrait s'écrire :

```
int u = 17;            // u ← u_0
int k = 0;

while (u != 1) {
    // u contient u_k          ①
    u = next(u);
    // u contient u_{k+1}      ②
    k = k+1;
    // u contient u_k          ③
}
// u = 1  et  u contient u_k   ④
printf("k = %d", k);
```

On peut aisément ensuite vérifier que tous les invariants sont corrects. Lorsque l'on entre dans la boucle pour la première fois [20], u contient $u_0$ et $k = 0$, donc ① est vraie.

Durant toute itération, si ① est vraie, alors ② puis ③ le seront également. En outre, si ③ est vraie à l'issue d'une itération, et qu'il y a une itération suivante, alors ① sera vraie à l'itération suivante (ni la valeur de u, ni la valeur de k ne changent).

Enfin, si lors de la dernière itération ③ est vraie, alors ④ le sera également : u contient bien 1 puisque l'on est sorti de la boucle **`while`**, ce qui garantit que `u != 1` est faux, et u contient bien $u_k$ d'après ③.

Grâce aux invariants, il est aisé de se convaincre que le programme est correct et que son comportement est celui attendu. C'est la raison pour laquelle il faut réfléchir avec soin à ces détails. En fait, il n'est pas utile ici de détailler autant [21], un unique invariant dans la boucle serait bien suffisant :

```
while (u != 1) {
    // u contient u_k
    u = next(u);
    k = k+1;
}
```

17. Une raison majeure de ne pas paraphraser le code est que, lorsque l'on modifie le code, il ne faudrait pas oublier de modifier le commentaire ! On ne peut en effet pas imaginer grand-chose de pire qu'un commentaire qui dit blanc lorsque le code dit noir... Les commentaires sont surtout là pour éclairer des choses que l'on ne peut pas déduire naturellement et simplement en lisant le code.
18. Tout en élaborant le programme de façon à avoir à le commenter le moins possible.
19. Et, peut-être plus important encore, de se convaincre soi-même !

20. En supposant donc $u_0 \neq 1$. Si $u_0 = 1$, on n'entre pas dans la boucle, et il est aisé de vérifier que l'affirmation ④ à la sortie de la boucle est vraie aussi, avec $u_0 = 1$ et $k = 0$.
21. C'est même rapidement malvenu, pour les raisons évoquées précédemment : on a largement paraphrasé le code avec les deuxième et troisième commentaires, et c'est une surcharge de travail inutile lorsqu'on le modifie.

Dans le cas de l'affirmation qui suit la boucle, on parle généralement de *post-condition*. De la même façon, une affirmation qui précède une boucle sera qualifiée de *pré-condition*.

Parfois, on parlera d'invariant de boucle pour une boucle donnée sans préciser exactement à quel endroit de la boucle il fait référence. Dans ce cas, il concerne toujours le *début* de la boucle, avant la séquence d'instructions constituant la boucle proprement dite. On s'arrange généralement pour le rédiger de sorte qu'il soit également valable en tant que post-condition (ce qui est le cas ici, u désignant $u_k$ également à la sortie de la boucle). Dans notre exemple, « u contient $u_k$ » peut donc être qualifié d'invariant de boucle pour la boucle `while` sans davantage de précisions.

## 4.3 Assertions

Dans certains cas particuliers, les invariants (qu'il s'agisse d'invariants de boucle, de pré- ou de post-conditions) peuvent être exprimés dans le langage C lui-même sous forme d'une condition booléenne (qui doit être vraie). Dans ce cas, on dispose d'une alternative aux commentaires, l'*assertion*. On l'inclue dans le code de la façon suivante :

```
assert(condition);
```

Par exemple, si une variable `mango` doit rester strictement positive au début de chaque itération d'une boucle, on pourra écrire au début de cette boucle :

```
assert(mango > 0);
```

Le but est essentiellement le même qu'un commentaire, mais avec une particularité : avec certaines options de compilation, la condition indiquée est évaluée à chaque passage, et si le résultat obtenu est `false`, l'exécution du programme cesse immédiatement, généralement avec un message indiquant que l'assertion n'est pas valide. Cela peut, lors du développement, aider à la recherche d'erreurs [22].

## 4.4 Correction et terminaison

Prenons un autre exemple, le calcul d'un PGCD de deux entiers positifs $a$ et $b$ par la méthode d'Euclide. Celle-ci est basée sur l'application de deux règles :

- le PGCD de $a$ et de 0 est $a$ ;
- si $b > 0$, le PGCD de $a$ et de $b$ est égal au PGCD de $b$ et du reste de la division entière de $a$ par $b$.

On applique donc la seconde règle tant que $b$ est non nul, et lorsque $b$ est nul, la première règle permet de conclure quand au PGCD recherché.

Le programme s'écrit donc par exemple :

```
int a = ...;
int b = ...;

while (b != 0) {
    // Le PGCD recherché est PGCD(a,b)
    int r = a % b;
    a = b;
    b = r;
}
printf("Le PGCD est %d\n", a);
```

L'invariant de boucle indiqué dans le programme permet de se convaincre que si le programme affiche un résultat, alors ce résultat est forcément correct [23]. En effet, l'invariant est vrai lorsque l'on entre pour la première fois dans la boucle, et le reste d'une itération sur l'autre par application de la seconde règle. Enfin, l'invariant peut faire office de post-condition, et donc le PGCD recherché est bien le PGCD de $a$ et $b$ à la sortie de la boucle. Mais puisque l'on est sorti de la boucle, c'est nécessairement que $b = 0$, et par application de la première règle, le PGCD recherché est donc bien la valeur contenue dans $a$ !

**Définition.** On dit qu'un programme est *partiellement correct* si, lorsqu'il produit un résultat, alors ce résultat est celui souhaité.

Notre programme est donc ici pariellement correct. Pourquoi « partiellement » ? Parce que pour l'instant, on a uniquement la garantie que *si* un résultat est affiché, il est correct. Mais la boucle `while` peut très bien ne jamais se terminer, et le programme pourrait ne jamais renvoyer de résultat !

**Définition.** On dit qu'un programme *termine* s'il produit un résultat en un temps fini.

**Définition.** On dit qu'un programme est *correct* s'il est partiellement correct et termine.

Précisons que lorsqu'un programme travaille sur des données ou accepte des arguments, il est nécessaire pour discuter de la correction d'un programme de préciser les données ou

22. En fait, l'activation des `assert` est généralement le comportement par défaut du compilateur. C'est une fois le programme mis au point qu'une option de compilation permet d'ignorer les `assert`, lesquels sont alors ignorés, tels de simples commentaires. Les conditions ne sont alors plus vérifiées (et le programme ne s'interrompt plus même si la condition se trouve être fausse), ce qui permet de ne pas effectuer de tests inutiles lors de l'exécution et permet parfois au compilateur de procéder à quelques optimisations supplémentaires.

23. Il existe des définitions plus formelles des invariants de boucle, par exemple dans la logique dite de Floyd-Hoare, qui vise à prouver formellement qu'un programme est correct au sens développé dans cette section. Dans ce cadre, les invariants de boucle doivent pouvoir également représenter une post-condition, ce qui généralement est le cas. Dans ce cours, nous nous contenterons d'un usage moins formel et plus pratique des invariants.

arguments qui sont permis, et pour affirmer que, par exemple, le programme est correct, il doit l'être pour toute donnée ou tout argument valide qu'on lui fournit.

Comment prouver que notre programme termine ? Nous étudierons cet aspect en détail plus tard, mais dans le cas présent, on peut le faire relativement aisément. L'idée est d'exhiber un *variant* de boucle. Il s'agit d'une quantité qui évolue à chaque boucle, et dont l'évolution est telle qu'il ne peut y avoir de boucle « infinie ».

Dans le cas présent, il suffit de voir que si $a$ et $b$ sont initialement des entiers positifs, alors $a$ et $b$ restent des entiers positifs tout au long de l'évolution du programme, et que $b$ diminue strictement à chaque itération. Les valeurs de $b$ forment donc une suite strictement décroissante dans $\mathbb{N}$, qui ne peut infinie. Par conséquent, on va nécessairement sortir de la boucle tôt ou tard !

Dans la très grande majorité des cas, il est possible de prouver la terminaison de la fonction en exhibant une grandeur entière, positive et strictement décroissante à chaque itération. Le terme de « variant » fait donc parfois référence, dans certains ouvrages, spécifiquement à cette situation. Dans ce cours, nous utiliserons une définition un peu plus large du terme « variant » pour gagner quelque peu en souplesse, mais on pourra être amené à devoir prouver qu'une séquence infinie est impossible. Nous aurons ultérieurement l'occasion de présenter un formalisme mathématique adapté à ces questions.

# 5 Itérations

## 5.1 Principe

Supposons que l'on souhaite déterminer la factorielle d'un entier n positif. On peut pour ce faire utiliser un accumulateur et une boucle **while**, ainsi qu'une variable entière pour égrenner les entiers de 2 à n inclus. Cela peut s'écrire par exemple :

```c
int n = ...;
int acc = 1;   // Un accumulateur pour le calcul
int k = 2;     // Un entier qui prendra les valeurs 2, 3, ... n

while (k <= n) {
    // acc contient (k-1)!
    acc = acc * k;
    k = k+1;
}
// k contient (n+1), acc contient n!
```

L'invariant de boucle en commentaire nous assure de la correction du code[24].

On se trouve ici dans le cas d'une boucle où on connait exactement le nombre d'itérations que l'on veut faire, et ce serait une situation naturelle pour une boucle « **for** » dans la plupart des langages.

Le langage C ne dispose pas réellement d'une telle construction. Mais compte tenu de son utilité, il fournit quand même un mot-clé **for** qui est une facilité d'écriture pour certaines boucles. Ainsi,

```c
for (①; ②; ③) ④
```

se trouve être un « raccourci » pour la structure suivante :

```c
{
    ①;
    while (②) {
        ④
        ③;
    }
}
```

- ① est une étape de « préparation », typiquement où l'on initialise (et éventuellement déclare) la variable qui gouvernera la boucle ;
- ② est une expression qui correspond à la condition de la boucle : tant qu'elle est évaluée à « vrai », la boucle se poursuit ;
- ③ est une instruction exécutée à l'issue de chaque itération, afin de préparer la suivante, très fréquemment une incrémentation ;
- ④ est le corps proprement dit de la boucle.

Comme pour les constructions **if** et **while**, si ④ peut être une unique instruction[25], on y trouvera la plupart du temps un bloc de code encadré par des accolades.

Notre calcul de factorielle pourra donc s'écrire, par exemple :

```c
int n = ...;
int acc = 1;

for (int k=2; k<=n; k=k+1) {
    // acc contient (k-1)!
    acc = acc * k;
}
```

24. En tout cas de la correction partielle, mais on peut également voir que le code termine puisque les valeurs de $k$ forment une séquence strictement croissante dans $\mathbb{N}$ majorée par $n$, qui ne peut pas être infinie ! Si on préfère, on peut se ramener au cas habituel d'une séquence entière positive strictement décroissante en considérant les valeurs successivement prises par $n-k$.

25. voire, comme dans le cas du **while** dans certain cas exceptionnels, vide !

Avec cette écriture, on comprend plus facilement que l'on considérera un entier k initialisé à 2, qui sera incrémenté entre chaque itération, et que la boucle se poursuivra tant que k<=n. Les valeurs successives de k seront donc $2, 3, \ldots, n-1, n$.

## 5.2 Particularités des boucles `for` en C

Nous avons donc à faire à une construction relativement similaire [26] à la construction Python « `for k in range(2, n+1)` ». Cependant, il est important de noter que les deux constructions présentent d'importantes différences !

Tout d'abord, il faut garder à l'esprit qu'il ne s'agit que d'une boucle `while` déguisée. Ainsi, tandis qu'en Python, la boucle `for`

```python
for k in range(0, 10):
    k = k+2
    print(k)
```

affiche les entiers $2, 3, \ldots, 10, 11$ (les modifications de k dans la boucle étant « oubliées » lors de l'itération suivante, il y a autant de noms k qu'il y a d'éléments dans le `range`, n'existant que le temps d'une itération particulière), en langage C, la boucle

```c
for (int k=0; k<10; k=k+1) {
    k = k+2;
    printf("%d\n", k);
}
```

affiche en revanche les entiers $2, 5, 8, 11$ ! La valeur contenue dans la variable k est en effet incrémentée de 2 au début de chaque boucle, et de 1 à la fin de chaque boucle, mais il s'agit toujours de la *même* variable $k$ dans toutes les itérations de la boucle.

En outre, le lecteur avisé aura peut-être remarqué que le mot-clé `for` définit spécifiquement un bloc d'instructions supplémentaires englobant toute la boucle, ce qui fait que si l'on déclare une variable en ①, sa portée ne dépasse pas de la boucle `for` [27]. Dans les deux exemples de boucle `for` proposés précédemment, le nom k n'est plus défini une fois sorti de la boucle [28]. Pour rappel, en Python, le nom utilisé dans la boucle `for` conserve sa dernière valeur après la boucle [29].

26. On pourrait écrire « `for(int k=2; k<n+1; k=k+1)` », parfaitement équivalent, pour souligner encore davantage la ressemblance.
27. Signalons que le même mécanisme existe dans le cas d'un `if` : on peut en théorie déclarer une variable au niveau de la condition, et sa portée ne s'étend que jusqu'à la fin du test. C'est toutefois moins souvent utilisé. Ce n'est en revanche pas possible dans la condition d'une boucle `while` car, pour qu'il n'y ait pas un conflit de nom, il faudrait que soit créé un bloc distinct pour *chaque* itération.
28. Les règles usuelles d'occultation de nom s'appliquent si un nom k existait préalablement à la boucle
29. Un comportement qui n'est pas apprécié de tous, d'ailleurs, et qui est relativement spécifique au langage Python.

## 5.3 Raccourcis d'écriture

Les incrémentations et décrémentations étant des opérations fréquentes, on a naturellement eu envie de trouver des raccourcis d'écriture. On sait déjà que l'on peut remplacer « `k = k+1` » par « `k += 1` ». Mais on dispose d'une écriture encore plus succinte ! « `++k` » est un autre raccourci pour ces deux expressions (et, de même, « `--k` » un raccourci pour « `k-=1` »). Vous verrez donc souvent :

```c
for (int k=2; k<=n; ++k) {
    ...
}
```

De même que « `k=k+1` » et « `k+=1` », « `++k` » peut être utilisé comme une expression dont la valeur est la valeur de la variable k après son incrémentation. Ainsi, dans le morceau de code suivant :

```c
int mango = 42;
int cherry = ++mango;
```

la valeur de mango est incrémentée (mango contient donc la valeur 43) et cette valeur 43 est également mémorisée dans la variable cherry. Comme on incrémente la variable avant d'utiliser la valeur, on parle d'opérateur de *pré-incrémentation*.

Inévitablement, les créateurs du langage en sont venu à se demander : et si on voulait la valeur *précédant* l'incrémentation ? Pour ce faire, on dispose également d'un opérateur de *post-incrémentation* où le `++` se trouve de l'autre côté du nom, comme sur cet exemple :

```c
int mango = 42;
int cherry = mango++;
```

Dans cette écriture, la valeur de mango est mémorisée *avant* son incrémentation, et si mango contient 43 à l'issue de ce morceau de code, cherry lui se voit affecter la valeur 42 qui a été mémorisée. Tout se passe comme si on avait écrit :

```c
int mango = 42;
int cherry;
{
    int tmp = mango;
    mango = mango + 1;
    cherry = tmp;
}
```

On remarquera que l'exécution a nécessité la création d'une variable temporaire. Ce second opérateur est d'un usage plus subtil, et recèle des dangers et des limitations sur lesquelles nous reviendrons. Parfois, il vaut mieux s'éviter d'y avoir recours.

Si on n'utilise pas « k++ » ou « ++k » comme des expressions, les deux écritures sont toutefois équivalentes (le compilateur devrait aisément se rendre compte qu'il n'a pas besoin de créer de variable temporaire dans ce cas). C'est pourquoi il n'est pas inhabituel de voir :

```c
for (int k=2; k<=n; k++) {
    ...
}
```

## 5.4 Interrompre les itérations

On dispose par ailleurs de deux instructions particulières intéragissant avec les boucles `while` et `for` : il s'agit de « `break` » et « `continue` ». Ces possibilités sont signalées ici à titre d'information, nous illustrerons plus tard leur usage sur des exemples concrets.

Si dans le corps d'une boucle on parvient à une instruction « `break` », alors on sort immédiatement de l'itération en cours, et on reprend l'exécution après le corps de la boucle. Cela peut parfois être utile, par exemple pour interrompre une recherche lorsque l'on a trouvé ce que l'on cherche, mais il faut éviter d'en abuser.

Le souci est, une fois de plus, lié à la bonne compréhension du programme par celui qui le lit. On s'attend, en général, qu'à la sortie de la boucle, la condition présente dans le `while` ou le `for` soit évaluée à « faux »[30]. Malheureusement, si l'on sort de la boucle prématurément à cause d'un `break`, cette « promesse » risque fort de ne pas être tenue. Un commentaire judicieusement placé fera toutefois beaucoup pour modérer ce problème.

L'instruction « `continue` », quant à elle, interrompt également le cours d'une itération, mais à sa suite, la condition de la boucle est évaluée. Si le résultat obtenu est « vrai », la boucle se poursuit avec une nouvelle itération. Elle pose moins de difficultés concernant invariants et post-conditions, mais est également un peu moins souvent utile.

## 5.5 Séquences d'expressions

Il peut arriver que l'on souhaite effectuer plusieurs opérations entre deux itérations d'une boucle `for`. On peut évidemment placer ces opérations à la fin du corps de la boucle, mais on peut être tenté de les placer en ③. Il n'est malheureusement pas possible de créer un bloc d'instructions à cet endroit.

Cependant, le langage C offre une autre solution. Il est en effet possible d'enchaîner un nombre quelconque d'expressions en les séparant par des virgules, de la sorte :

```c
expression_1, expression_2, expression_3
```

30. Et bien souvent, aussi, que l'invariant de boucle fasse aussi office de post-condition.

L'ensemble forme lui-même une seule expression. Lorsqu'elle est évaluée, chacune des sous-expressions est évaluée tour à tour, dans l'ordre et le résultat de l'évaluation de l'ensemble correspond au résultat de l'évaluation de la dernière expression (les autres résultats étant simplement ignorés). Ainsi, si l'on écrit

```c
mango = 22-11, 37+9-4;
```

la variable `mango` reçoit la valeur 42 (l'expression « `22-11` » est évaluée, mais le résultat est « jeté »). Ce genre de construction n'a cependant d'intérêt que si les expressions, en dehors de la dernière, effectuent un peu plus qu'un calcul, par exemple une affectation.

On peut donc par exemple, si l'on souhaite créer une boucle avec deux indices progressant l'un vers l'autre, jusqu'à ce qu'ils se rejoignent, écrire :

```c
for (int i=0, j=n-1; i<=j; i++, j--) {
    ...
}
```

Attention donc, du fait de l'existence de cet opérateur « virgule »,

```c
int mango, cherry = 1, 2;
```

est correct, mais si `cherry` est initialisé à 2, `mango` n'est *pas* initialisée !

## 5.6 Omission d'éléments dans un `for`

Les différents éléments constituant un `for` peuvent éventuellement être omis. Par exemple, s'il n'est pas besoin de déclarer ni d'initialiser la variable `k` (par exemple parce que la variable existe préalablement), on peut écrire

```c
for (; k<10; k=k+1) {
    ...
}
```

Ainsi, « `for (; condition; ) { ... }` » se comporte très exactement comme la boucle « `while (condition) { ... }` ».

En théorie, on peut même omettre la condition ②, auquel cas la boucle `for` ne s'arrêtera jamais[31] (comme si l'évaluation de la condition donnait toujours « vrai »), sauf si elle rencontre un `break`. « `for (;;) { ... }` » se comporte donc comme « `while (true) { ... }` ».

31. La question des boucles « infinies » est délicate en C, toujours pour des impératifs d'optimisation. Le compilateur s'est vu octroyer le droit de faire l'hypothèse qu'une boucle, dans un programme correct, va toujours terminer (sous certaines conditions), et agir en conséquence, ce qui peut donner des résultats surprenants.

# 6 Fonctions

## 6.1 Définitions de fonctions

Afin de mieux organiser le code et d'en faciliter l'écriture, la lecture et la maintenance, en langage C comme dans tout langage de programmation, on s'efforce de découper le programme en « morceaux » de taille raisonnable, des *fonctions*.

Une fonction, en langage C, est un morceau de programme qui attend éventuellement des données (qualifiées généralement d'*arguments* ou de *paramètres*), effectue une série de calculs ou d'opérations, et éventuellement peut renvoyer un résultat. Les fonctions portent un nom afin de pouvoir leur faire référence ailleurs dans le programme, et provoquer ainsi l'exécution du morceau de code concerné.

Une *définition de fonction* en langage C consiste en un bloc d'instructions (et donc encadré par des accolades, généralement appelé le *corps* de la fonction) précédé de sa *signature*, qui précise le nom attribué à la fonction, le type des paramètres qu'elle attend (ainsi que le nom que ces paramètres porteront dans le corps de la fonction), et le type de ce qui sera renvoyé par la fonction.

La signature débute par la mention du type du résultat, suivi du nom de la fonction, et enfin des informations concernant les arguments, entre parenthèses, sous la forme ressemblant à une série de déclarations séparées par des virgules [32] [33]. Il n'y a aucun mot-clé indiquant que ce qui suit est une déclaration de fonction.

Une fonction appelée `disc_area` qui prendrait en argument un flottant (supposé positif) correspondant au rayon d'un disque et renvoyant un flottant correspondant à son aire se présenterait donc sous cette forme :

```
double disc_area(double radius) {
    ... // calcul de l'aire
}
```

Pour l'aire d'un polygone régulier à $n$ côtés, à partir du nombre de côté et de leur taille, on aurait par exemple quelque chose de la sorte :

```
double reg_poly_area(int nb_sides, double side_length) {
    ... // calcul de l'aire
}
```

32. Comme une virgule sépare chaque argument, il faut nécessairement répéter le type pour chacun des arguments.
33. Dans les premières versions du langage, la syntaxe pouvait être quelque peu différente, avec la seule mention des noms des paramètres entre les parenthèses, et des déclarations indiquant leur type entre la parenthèse fermante et l'accolade ouvrante indiquant le corps de la fonction.

Dans le corps de la fonction, les éventuels arguments se comportent comme des variables locales, qui seront initialisées avec une valeur fournie lors de l'appel. Il est donc possible de les utiliser librement dans les calculs, et de leur affecter de nouvelles valeurs.

Lorsqu'une fonction ne prend aucun argument, on indiquera « `void` » entre les parenthèses là où se trouvent usuellement les arguments, sur ce modèle [34] [35] :

```
int foo(void) {
    ... // corps de la fonction
}
```

Le mot-clé `return` permet d'indiquer qu'on en a terminé avec la fonction, et de spécifier la valeur « résultat » qu'elle doit renvoyer. Par exemple :

```
double disc_area(double radius) {
    double pi = 3.1415926535897932;

    return pi * radius * radius;
}
```

Si la valeur (ou le résultat de l'évaluation de l'expression) suivant le mot-clé « return » a un type qui ne correspond pas à celui déclaré dans la signature comme le type renvoyé par la fonction, une conversion vers le type idoine a lieu.

Le mot-clé `return` provoque la sortie immédiate de la fonction. On le trouve donc le plus souvent à la fin de la fonction, mais il est possible de le placer n'importe où. Si l'on reprend par exemple notre suite de Syracuse, on peut écrire une fonction `next` prenant en argument un terme $u_n$ et renvoyant le terme $u_{n+1}$ de la sorte :

```
int next(int u) {
    if (u%2 == 0) {
        return u/2;
    }
    // <- n'est atteint que si u est impair
    return 3*u+1;;
}
```

34. Comme pour les noms de variables empruntés aux fruits, il existe des noms couramment utilisés pour les fonctions pour lesquelles on ne peut trouver de nom pertinent. Les noms les plus couramment utilisés sont `foo` et `bar`. Leur origine est difficile à établir, mais ils sembleraient avoir été introduits par les membres d'un club du MIT qui, possiblement, se serait inspirés du terme d'argot militaire FUBAR signifiant « bousillé au point d'être méconnaissable » (Fucked-Up Beyond All Recognition).
35. Si l'on écrit « `int foo()` », sans le `void` entre les parenthèses, le compilateur ne protestera pas, et le programme produit va fonctionner. Mais il ne s'agit pas alors d'une fonction ne prenant pas de paramètres, mais d'une fonction dite *variadique* prenant *un nombre quelconque* de paramètres, non spécifiés. La façon d'accéder aux paramètres dans ce genre de cas dépasse le cadre de ce cours, nous ne nous y intéresserons donc pas.

Pour diverses raisons, sortir d'une fonction grâce à un return en plein milieu de son corps peut être mal perçu [36]. On évitera de le faire sans une bonne raison, dans la mesure du possible dans des fonctions pas trop longues, et de le mettre bien en évidence.

Une fonction déclarée comme renvoyant une valeur n'est considérée correcte que si son exécution se termine *toujours* par un `return`. On trouve dont toujours un `return` à la fin du corps d'une telle fonction [37].

Précisons enfin qu'en langage C, il n'existe pas de fonctions « locales ». Il n'est en effet pas permis de définir une fonction à l'intérieur d'un bloc, et notamment à l'intérieur d'une autre fonction.

## 6.2 Appels de fonction

Pour faire appel à une fonction, on utilise son nom suivi de parenthèses, contenant autant d'expressions que la fonction compte d'arguments. Chacune des expressions est évaluée [38], et les résultats obtenus servent d'argument à la fonction. On peut par exemple faire appel à la fonction `aire_poly_regulier` définie précédemment de la sorte :

```
    // Calcul de l'aire d'un hexagone de côté 3.5
    double area = reg_poly_area(6, 3.5);
```

L'initialisation des arguments fonctionne comme une affectation normale. Ainsi, si les valeurs fournies lors de l'appel à la fonction n'ont pas le type attendu, le compilateur s'efforcera de les convertir. Ainsi, l'appel suivant est tout aussi correct, car le compilateur sait convertir l'entier 3 en flottant :

```
    double area = reg_poly_area(6, 3);
```

L'ennui, c'est que l'on perd une opportunité d'identifier de potentielles erreurs [39]. Si par exemple on se trompe dans l'ordre des arguments et que l'on écrit :

```
    double area = reg_poly_area(3.5, 6);
```

le compilateur ne protestera pas, convertira la valeur flottante 3.5 en entier (en tronquant la valeur, ce qui nous donne un polygone à trois côtés), et la valeur entière 6 en flottant. On obtiendra donc l'aire d'un triangle équilatéral de côté 6 !

36. Pour des raisons de lisibilité, mais aussi et surtout parce que cela peut provoquer des « fuites » de mémoire si l'on n'est pas prudent, comme on le verra plus tard.
37. `main` constitue une exception, il est toléré d'omettre le `return`, la fonction renvoyant alors 0.
38. L'ordre d'évaluation n'est pas spécifié, il n'est même pas garanti qu'elles soient évaluées une à une. Le langage exige que l'évaluation des expressions ne dépendent pas de l'ordre d'évaluation, de sorte que si `foo` est une fonction qui prend en argument deux entiers, et si `i` est un entier, « `foo(i++, i++)` » n'est pas permis.
39. De nombreuses conversions ne sont pas possibles, et dans ce cas, l'erreur de type sera bien signalée lors de la compilation.

Comme dans le cas des déclarations de variables, une fonction existe dès l'écriture de la signature. On peut donc appeler une fonction depuis le corps de cette même fonction. Une telle fonction est qualifiée de *récursive*. Par exemple, la fonction factorielle prenant en argument un entier positif [40] peut être définie de la sorte :

```
int fact(int n) {
    if (n==0) {
        return 1;
    }
    return n * fact(n-1); // <- appel récursif
}
```

Le fait de pouvoir écrire des fonctions récursives dans un langage offre beaucoup de possibilités, mais amène également de nombreuses difficultés tant techniques (pour la compilation et la gestion de la mémoire) que théoriques (lorsque l'on se pose des questions de correction et de terminaison). Nous aurons l'occasion de les étudier en détail un peu plus tard.

## 6.3 Fonctions et procédures

Parfois, il n'est pas utile que la fonction renvoie un résultat : dans le cas de fonctions effectuant simplement un affichage, par exemple. Pour déclarer une fonction ne renvoyant pas de résultat, on utilisera `void` en lieu et place du type du résultat, comme sur cet exemple qui affiche un message indiquant le carré de son argument :

```
void print_square(int n) {
    printf("Le carré de %d est %d\n", n, n*n);
}
```

Une fonction ne renvoyant pas de résultat est généralement appelée *procédure* [41]. Le `return` n'est alors plus requis, sauf pour interrompre l'exécution d'une fonction, comme ci-dessous où le message n'est affiché que si l'entier `n` passé en argument est impair :

```
void foo(int n) {
    if (n%2==0) {
        return;
    }
    // <- n'est atteint que si u est impair
    printf("Hello World\n");
}
```

40. Et pas trop grand, car les valeurs calculées ne doivent pas dépasser `INT_MAX`.
41. Le terme *fonction* pouvant alors être réservé aux seules « fonctions » renvoyant effectivement un résultat.

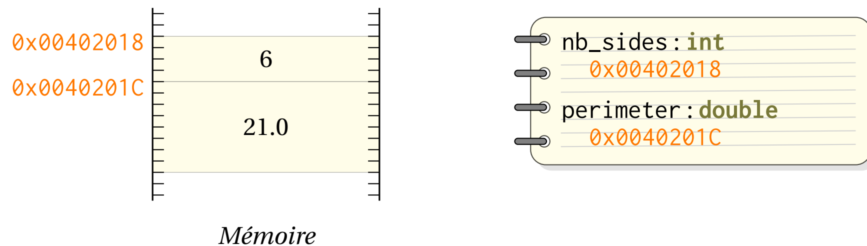
## 6.4 Gestion de la mémoire

Lorsque l'on fait appel à une fonction, une variable est créée pour chacun des paramètres, en réservant un espace adéquat dans la mémoire. Chacun des paramètres correspond donc à une variable *locale* à la fonction, dont la portée s'étend jusqu'à la fin de la fonction [42]. Ces variables sont immédiatement initialisées avec la valeur fournie lors de l'appel, après une conversion si cela était nécessaire.

Considérons par exemple le programme suivant, déterminant l'aire d'un polygone régulier connaissant le nombre de côtés et le périmètre :

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

double reg_poly_area(int nb_sides, double side_length) {
    double pi = 3.1415926535897932;

    double area = nb_sides * side_length * side_length
                                    / tan(pi / nb_sides) / 4.0;
    return area;
}

int main(void) {
    int nb_sides = 6;        // Nombre de côtés du polygone
    double perimeter = 21.0; // Périmètre

    double area = reg_poly_area(nb_sides, perimeter/nb_sides);
    printf("L'aire du polygone est %f\n", area);
    return 0;
}
```

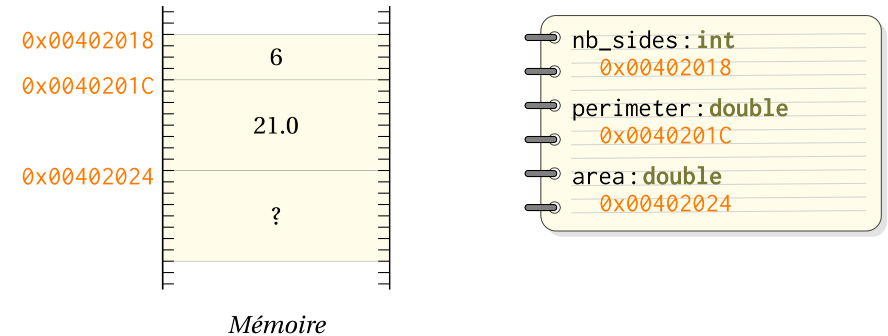
Le programme commence par déclarer et initialiser deux variables nb_sides et perimeter.




*Mémoire*

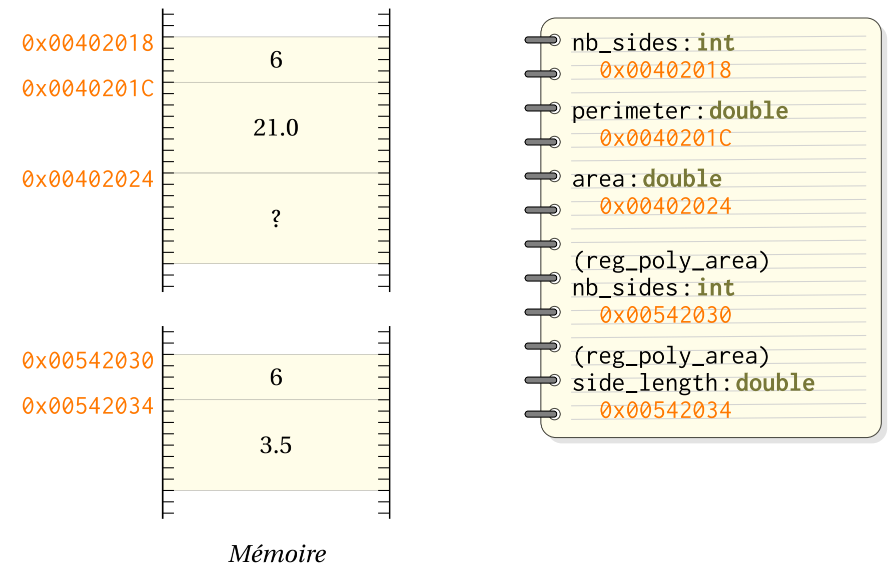
42. Tout se passe comme si les variables correspondant aux paramètres avaient été déclarées dans le corps de la fonction, il n'est donc pas possible de déclarer une variable locale portant le même nom qu'un paramètre de la fonction.

Puis il déclare une variable area qui contiendra le résultat de l'appel de fonction :




*Mémoire*

Ensuite, les expressions correspondant aux arguments de l'appel de fonction sont évalués (on obtient 6 et 3.5), et au moment de l'appel, on réserve, dans la mémoire, de la place pour deux variables locales initialisées avec ces valeurs correspondant aux arguments de la fonction [43] :




*Mémoire*

On remarquera que le compilateur doit gérer ici deux variables *différentes* portant le même nom, nb_sides : la variable déclarée dans main, et la variable correspondant à l'argument de reg_poly_area. Lorsque l'on se trouve dans la fonction reg_poly_area, c'est cette seconde variable qui est disponible.

43. Les adresses et emplacement en mémoire utilisés ici n'ont qu'un but d'illustration et ne sont pas réalistes. Nous verrons plus tard en détail comment est gérée la mémoire disponible et le placement exact des variables dans celle-ci.

Ces variables sont complètement indépendantes : les modifications éventuelles de nb_sides dans la fonction reg_poly_area n'auront aucune conséquence sur la valeur mémorisée dans la variable nb_sides de la fonction main.

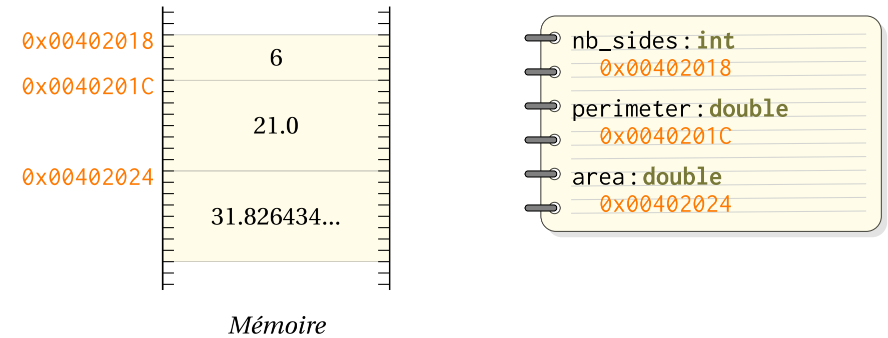
Cette façon de passer les arguments d'une fonction est appelée *passage par valeur*. D'un langage de programmation à un autre, la manière dont les arguments sont manipulés lors d'un appel de fonction peut grandement varier. Le mécanisme utilisé par le langage C est très différent, on le verra, de celui utilisé par OCaml ou Python.

Une conséquence du passage des arguments par valeur est que, quoi que fasse une fonction avec ses variables locales, cela n'aura jamais de conséquences directes sur le contenu de variables définies ailleurs. La fonction reg_poly_area n'a ici *aucun* moyen d'agir sur le contenu des variables de main : elles sont hors de portée [44].

L'exécution de la fonction reg_poly_area suit ensuite normalement son cours. On déclare et initialise deux variables supplémentaires, pi et area, elles aussi locales à la fonction reg_poly_area.

*Mémoire*

On parvient alors à la fin de la fonction. La valeur renvoyée est, là encore, *copiée* vers un emplacement mémoire appartenant à la fonction appelante, ici main. Dans le cas présent, la valeur renvoyée est utilisée pour l'initialisation de la variable area de la fonction main. Comme nous sommes parvenus à la fin de la fonction, on a atteint la fin de la portée des variables locales de la fonction reg_poly_area, et la mémoire qu'elles occupaient est libérée. L'état de la mémoire est alors :




*Mémoire*

L'appel à la fonction printf se déroule ensuite de la même façon, la valeur de la variable area notamment est à nouveau copiée le temps de l'appel.

## 6.5 Variables « globales » et effets de bord

Nous avons laissé entendre tantôt qu'il était possible de déclarer une variable en-dehors de tout bloc, avec une portée s'étendant de leur déclaration jusqu'à la fin du fichier. Les variables ainsi déclarées sont qualifiées de *variables globales*. Ces variables sont utilisables par n'importe quelle fonction définie après la déclaration de la variable. On peut par exemple se servir d'une variable globale pour compter le nombre d'appels à une fonction :

```c
int nb_calls = 0;  // Nombre d'appels à la fonction area

double area(double radius) {
    double pi = 3.1415926535897932;

    nb_calls = nb_calls + 1; // On comptabilise l'appel
    return pi * radius * radius;
}
```

Il est toutefois recommandé de limiter au maximum l'usage de variables globales. En effet, la signature d'une fonction n'indique pas si elle accédera à (ou modifiera) une variable globale. C'est donc une source fréquente d'erreurs.

Dans la mesure du possible, les données nécessaires à l'exécution d'une fonction devraient être transmises uniquement par l'intermédiaire de ses paramètres, et les fonctions ne devraient pas modifier de données autres que celles auxquelles leurs paramètres

44. Nous verrons cependant bientôt un mécanisme qui permet de passer outre cette limitation.

pourraient faire référence. En cas d'entorse à cette règle générale, il est important de documenter la fonction.

De manière générale, on qualifie les fonctions selon le fait que leur exécution a de l'influence ou non sur le reste du programme.

**Définition.** On dit qu'une fonction a des *effets de bord* si son exécution peut avoir des conséquences qui ne se limitent pas à la modification de ses propres variables locales.

Ainsi, si une fonction modifie une variable globale, si elle intéragit avec l'utilisateur par le biais d'un affichage ou d'une lecture du clavier, si elle écrit sur un disque ou transmets des données sur le réseau, on dit qu'elle a des effets de bord [45] [46].

**Définition.** On dit qu'une fonction est *déterministe* si le résultat qu'elle revoie est toujours le même lorsqu'on lui fournit les mêmes arguments.

**Définition.** On qualifie une fonction de *pure* lorsqu'elle est déterministe et n'a pas d'effet de bord.

Les fonctions pures sont ce qui correspond le plus aux fonctions mathématiques : elles fournissent bien un résultat, toujours le même, pour chaque élément du domaine constitué par les arguments admissibles.

Elles sont particulièrement intéressantes d'un point de vue compilation, car elles ne réservent pas de mauvaises surprises : on pourrait, en principe et avec beaucoup de temps et de mémoire, précalculer tous les résultats pour chacune des combinaisons d'arguments possibles, et ne faire que profiter de ces précalculs lors de l'exécution.

Bien évidemment, une fonction ne peut généralement pas être pure si elle fait appel, lors de son exécution, à des fonctions qui ne le sont pas. Précisons enfin que la définition de « pure » peut parfois subtilement varier d'un ouvrage à l'autre concernant le caractère déterministe (on peut par exemple tolérer que le résultat renvoyé dépende du contenu de variables globales).

## 6.6 Déclarations

Considérons les suites $(u_n)_{n\in\mathbb{N}}$ et $(v_n)_{n\in\mathbb{N}}$ définies par

$$\begin{cases} u_0 = 1 \quad \text{et} \quad u_n = u_{n-1} + 2v_{n-1} \\ v_0 = 1 \quad \text{et} \quad v_n = 3u_{n-1} + v_{n-1} \end{cases}$$

45. On parle de « *side-effects* » en anglais, la traduction française n'est probablement pas très fidèle, mais s'est imposée.

46. Nous verrons plus tard qu'il y a d'autres situations correspondant à des effets de bords, par exemple si la fonction possède des variables locales qui sont préservées d'un appel sur l'autre, ou si elle modifie le contenu d'emplacements mémoires qui ne lui appartiennent pas via une adresse qui lui aurait été transmise.

Avec un minimum de travail, il est possible de calculer une valeur $u_n$ ou $v_n$ quelconque avec une simple boucle. On pourrait toutefois vouloir les retranscrire directement en langage C sous la forme de deux fonctions en écrivant [47] :

```c
int u(int n) {
    if (n==0) { return 1; }
    return u(n-1) + 2 * v(n-1);  // <- v n'est pas définie !
}

int v(int n) {
    if (n==0) { return 1; }
    return 3 * u(n-1) + v(n-1);
}
```

Rien ne s'y oppose à priori, si ce n'est un détail : dans le corps de la fonction `v`, les fonctions `u` et `v` sont parfaitement définies. En revanche, dans le corps de la fonction `u`, la fonction `v` n'est pas encore définie, donc l'appel `v(n-1)` pose problème [48] !

Pour le résoudre, il est possible de *déclarer* une fonction sans pour autant la définir, en renseignant simplement sa signature. Dans le cas présent, pour déclarer `v`, on ajoutera simplement, avant la définition de la fonction `u` :

```c
int v(int n);
```

Cette seule ligne suffit à indiquer au compilateur qu'on définira (ultérieurement) une fonction nommée `v` prenant en argument un entier et renvoyant un entier. Même s'il ne connaît pas encore la séquence d'instructions correspondant à cette fonction, cela lui permet d'ores et déjà d'accepter des appels à cette future fonction, dont le `v(n-1)` qui apparaît dans la définition de la fonction `u`.

Il n'est pas requis de préciser les noms des arguments dans les déclarations de fonctions [49], car seuls les types sont réellement importants si on souhaite savoir ce qu'attend la fonction, même s'ils sont bienvenus pour rendre la déclaration plus claire. On pourrait donc simplement écrire :

```c
int v(int);
```

47. Dans le cas présent, signalons que ce n'est pas la meilleure idée qui soit, car un calcul de $u_n$ conduira à effectuer $2^n - 1$ appels de fonctions, quand bien même il n'y a que $2n-1$ valeurs à déterminer !

48. Historiquement, si le compilateur trouvait un appel à une fonction `foo` sans qu'il en connaisse la signature, il faisait l'hypothèse qu'elle avait pour signature « `int foo()` » (un nombre quelconque d'arguments non spécifiés et un résultat entier). Une signature « par défaut » qui, ici, conviendrait pour `v` ! Cette possibilité, source d'erreur, a été supprimée dans le standard C99.

49. Et si on mentionne un nom et qu'il change entre la déclaration et la définition, ce n'est pas non plus un problème, seul le type importe réellement.

## 6.7 Découpage du fichier source et liaison

Afin de ne pas avoir à recompiler l'intégralité du code source si l'on modifie une seule ligne d'une fonction, il est courant de diviser les sources d'un programme en plusieurs fichiers. Ces fichiers peuvent alors être compilés séparément en différents fichiers binaires, lesquels sont ensuite regroupés en un unique exécutable en une étape dite de *liaison*.

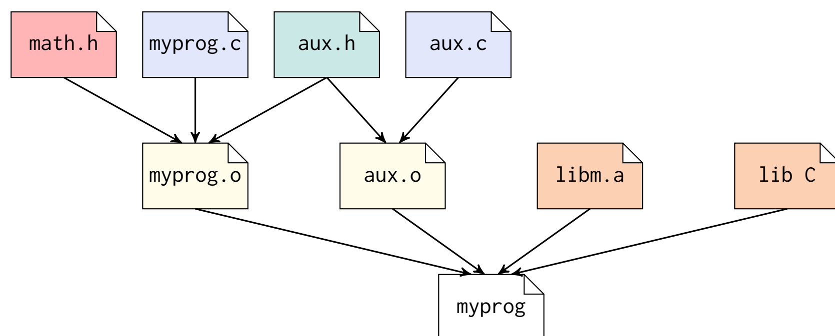
C'est d'autant plus important car l'on ne souhaite pas recompiler les fonctions de la bibliothèque standard (telles que `printf`, `tan`...) à chaque fois que l'on compile un programme !

Seulement, si l'on tente de compiler isolément un morceau du programme, son contenu va faire appel à des fonctions qui sont définies à un autre endroit, et que le compilateur ne connaît donc pas ! Il faut donc disposer, dans le fichier source que l'on compile, des *déclarations* des fonctions qui seront définies ailleurs.

En général, lorsque l'on écrit un fichier source C contenant des fonctions qui seront utilisées ailleurs, on crée un fichier supplémentaire, dit fichier d'*en-tête*, avec généralement l'extension « `.h` », qui contient les déclarations des fonctions qui pourront être appelées depuis un *autre* fichier source.

Ainsi, lorsqu'un morceau de code souhaite faire appel à des fonctions extérieures, il n'a qu'à inclure le fichier d'en-tête correspondant pour disposer de toutes les déclarations. C'est ce que l'on fait notamment lorsque l'on utilise les directives telles que « `#include <math.h>` ».

Par exemple, on peut, dans un projet, vouloir regrouper un ensemble de fonctions auxiliaires dans un fichier source « `aux.c` » et avoir accès à ses fonctions depuis un fichier source « `myprog.c` » contenant le programme principal (dont la fonction `main`). Pour ce faire, on regroupe des déclarations des fonctions définies dans `aux.c` qui doivent être accessibles depuis `myprog.c` dans un fichier `aux.h`, que l'on inclue, par une directive `#include`, dans le fichier source `myprog.c`. Si le programme a également besoin de fonction de la bibliothèque standard C, il incluera d'autres fichiers d'en-tête, tel que `math.h` par exemple. La structure du projet serait par exemple la suivante :




La compilation produit des fichiers « objets », compilés mais non exécutables en l'état car incomplets, généralement d'extension « `.o` ». La liaison regroupe les différents fichiers objets (dont les fichiers objets fournis par la bibliothèque standard) pour construire le fichier exécutable `myprog` souhaité.

Les outils de compilations peuvent effectuer chacune de ces étapes indépendamment, ou bien se charger seul de l'ensemble du travail. Précisons qu'outre le fait que l'on puisse gagner du temps en ne compilant pas l'intégralité du code source à chaque modification mineure, le passage par des fichiers objets permet également, sous certaines conditions, de regrouper dans un même programme des morceaux de code écrits dans des langages différents !

Ce mécanisme présente cependant une difficulté : les fichiers inclus avec un `#include` peuvent eux-même avoir besoin d'autres définitions, et il est fréquent de trouver des `#include` dans les fichiers d'en-tête (par exemple, un fichier d'en-tête pourrait avoir besoin d'un type, tel que `bool` [50], défini par un autre fichier d'en-tête). Le hic, c'est que cela peut conduire à des inclusions multiples du même fichier, or il n'est pas permis de procéder plusieurs fois à la même déclaration.

Pour contourner le problème, on protège généralement les fichiers d'en-tête de la façon suivante :

```c
#ifndef MY_HEADER_H
#define MY_HEADER_H
// contenu du fichier d'en-tête
#endif
```

`MY_HEADER_H` est un identifiant que l'on peut choisir librement (typiquement lié au nom du fichier d'en-tête). `#ifndef` est une directive destinée au préprocesseur (qui opère avant le compilateur). Si l'identifiant qui ne suit est connu du préprocesseur, toute la partie du fichier jusqu'au `#endif` suivant est ignoré. `#define` est une directive qui définit l'identifiant à destination du préprocesseur [51].

Ainsi, lors de la première inclusion, l'identifiant est inconnu, et le contenu du fichier d'en-tête est inclus normalement, définissant l'identifiant au passage. Lors d'éventuelles tentatives inclusions ultérieures, le contenu du fichier d'en-tête sera ignoré, évitant ainsi des déclarations multiples. On parle d'*idempotence*.

50. Dans ce cas précis, il pourrait utiliser `_Bool` qui est déjà disponible, mais nous verrons d'autres types et constantes où ce n'est pas si simple.

51. Cette directive permet de faire bien d'autres choses, y compris définir des « macros-commandes » qui permettront au préprocesseur d'altérer le fichier source préalablement à l'étape de compilation. Ces mécanismes, qui peuvent causer de nombreuses difficultés, sont hors-programme, et les seuls utilisations que nous ferons des directives destinées au préprocesseurs sont celles déjà vues.

# Exercices

## Ex. 2.1 – Années bissextiles

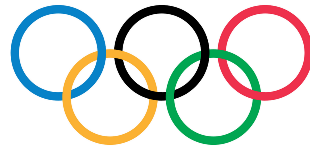
Afin de s'assurer que les solstices et équinoxes ne changent pas de date dans l'année, il n'est pas possible d'avoir des années dont la durée est toujours la même. Aussi a-t-on introduit le principe d'années *bissextiles*, qui comptent une journée de plus en février. Avant 1582, ces années étaient celles divisibles par 4. Après 1582, pour plus de précision, les années divisibles par 100 ne sont plus bissextiles, excepté si elles sont divisibles par 400 (2000 était bissextile, mais 1900 ne l'était pas).

**1.** Proposer une expression booléenne contenant une variable entière positive y désignant une année passée, donnant un booléen indiquant si l'année y est bissextile.

**2.** Essayer de simplifier au maximum l'expression booléenne (trois opérateurs « `||` » et/ou « `&&` » suffisent [52]).

## Ex. 2.2 – Jeux olympiques

Les jeux olympiques modernes sont organisés tous les quatre ans, depuis la première édition en 1896. Depuis 1924 ont également lieu des jeux d'hiver, tous les quatre ans jusqu'en 1992, puis à nouveau tous les quatre ans depuis 1994. Les guerres mondiales 1914-1918 et 1939-1945 ont vu l'annulation de ces événements, et les jeux olympiques d'été 2020 ont dû être déplacé à l'été 2021 pour cause de Covid.



**1.** Proposer une expression booléenne contenant une variable entière positive y désignant une année passée, et renvoyant un booléen indiquant si des jeux olympiques d'été modernes se sont tenus l'année désignée. On s'efforcera de trouver l'expression la plus simple (succinte) possible.

**2.** Faire de même pour les jeux olympiques d'hiver.

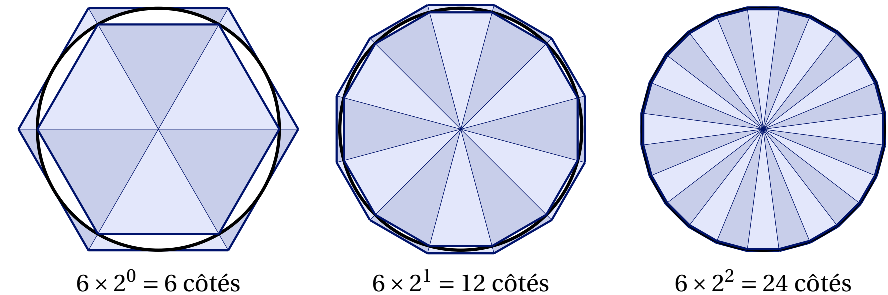
## Ex. 2.3 – Grands nombres

Écrire un programme C déterminant le plus petit entier strictement positif $k$ tel que 7654321 divise $3^k - 2^k$. On réfléchira à la façon d'écrire le programme pour qu'il soit raisonnablement efficace et compatible avec les limites de représentations des nombres en C.

52. En partant du principe que les termes entre les opérateurs doivent rester très simples. Il serait techniquement possible de construire une expression booléenne sans aucun opérateur booléen qui réponde à la question, même sans utiliser la propriété qu'a le langage C de pouvoir interpréter les booléens comme des entiers.

## Ex. 2.4 – Méthode d'Archimède

Archimède a proposé une méthode d'encadrement de π qui consiste à calculer les périmètres, notés respectivement $p_n$ et $p'_n$, des polygones à $6 \times 2^n$ côtés inscrits dans un cercle de diamètre unité et circonscrits à celui-ci :




On peut montrer que la suite $(p_n)_{n\in\mathbb{N}}$ est strictement croissante, la suite $(p'_n)_{n\in\mathbb{N}}$ strictement décroissante, et que toutes deux tendent [53] vers la valeur π, de sortent qu'elles permettent d'obtenir un encadrement de π avec autant de précision qu'on le souhaite.

Pour calculer la suite $p_n$, notons $a_n$ la distance entre un côté du polygone inscrit à $6^n$ côtés et le centre du cercle. On vérifie aisément que

$$a_0 = \sqrt{\frac{1}{4} - \frac{1}{16}} \quad , \quad a_{n+1} = \sqrt{\frac{a_n}{4} + \frac{1}{8}} \quad \text{et} \quad p_0 = 3 \quad , \quad p_{n+1} = \frac{p_n}{2a_{n+1}}$$

**1.** Proposer un programme affichant les 10 premières valeurs [54] de la suite $(p_n)_{n\in\mathbb{N}}$. On peut, en C, afficher le contenu d'une variable flottante `x` en écrivant par exemple « `printf("x = %lf\n", x)` ».

**2.** Établir des relations de récurrence similaires pour la suite $(p'_n)_{n\in\mathbb{N}}$, et modifier le programme pour qu'il affiche également les valeurs $p'_n$.

**3.** Á l'époque, Pythagore avait proposé une estimation de π en effectuant le calcul à la main jusque $p_4$ (soit un polygone à 96 côtés). Combien de chiffres significatifs a-t-il réussi à obtenir?

**4.** Proposer un programme déterminant et affichant une estimation de la valeur de π à $10^{-12}$ près [55] [56] (on n'effectuera pas plus de calculs que nécessaire).

53. Il semble intuitif que ces périmètres tendent progressivement vers le périmètre du cercle, donc π, mais la démonstration rigoureuse est un peu plus délicate.
54. Bien évidemment, les `double` n'ont pas une précision infinie, et ces valeurs ne seront qu'approchées.
55. Rappelons que $10^{-12}$ peut s'écrire « `1e-12` ».
56. On admettra que les erreurs dûes aux calculs avec des `double` ne remettent pas en cause le résultat renvoyé par le programme, ce qui peut se montrer rigoureusement mais est une tâche très complexe. Le programme ne permettra cependant pas d'obtenir des estimations beaucoup plus fines, car on se trouve à la limite de la précision des `double`.

### Ex. 2.5 – Suite de tribonacci

La suite de « tribonacci »[57] est définie par $u_0 = u_1 = 0$, $u_2 = 1$, et pour tout $n \geqslant 3$, $u_n = u_{n-1} + u_{n-2} + u_{n-3}$.

Proposer un programme affichant les 50 premiers termes de cette suite.

### Ex. 2.6 – Suite des carrés des chiffres

On considère une suite récurrente où, pour tout $n \geqslant 1$, $u_n$ est égal à la somme des carrés des chiffres de $u_{n-1}$. On peut montrer que dans cette suite, on voit toujours apparaître soit l'entier 1 (la suite devient alors constante), soit l'entier 89 (la suite se poursuivant dans ce cas sur le cycle $89 \longmapsto 145 \longmapsto 42 \longmapsto 20 \longmapsto 4 \longmapsto 16 \longmapsto 37 \longmapsto 58 \longmapsto 89$).

**1.** Proposer un programme indiquant, pour un $u_0$ donné, le plus petit $k$ tel que $u_k = 1$ ou $u_k = 89$. Y a-t-il un risque de dépassement de capacité ?

**2.** En déduire un programme déterminant le $u_0$ dans $[\![1..99999]\!]$ pour lequel ce $k$ est le plus grand.

### Ex. 2.7 – Paraskevidékatriaphobie

Sachant que le 1er janvier 2001 était un lundi, déterminer combien le XXIe siècle (du 1er janvier 2001 au 31 décembre 2100) comprendra de vendredi 13.

### Ex. 2.8 – Méthode de Brown

Une méthode originale pour calculer une approximation de la valeur de π a été, semble-t-il, proposée par Kevin Brown. Elle consiste dans un premier temps à calculer, pour un $p$ donné, le terme $a_{p-2}$ d'une séquence d'entiers définie par :

- $a_0 = p$ ;
- pour $0 < k \leqslant p-2$, $a_k$ est le plus petit entier supérieur ou égal à $a_{k-1}$ divisible par $p-k$.

On obtient alors une valeur approchée[58] de π avec $\dfrac{p^2}{a_{p-2}}$.

Ainsi, pour $p = 10$, on a $a_0 = 10$, $a_1 = 18$ (divisible par $10-1 = 9$), $a_2 = 24$, $a_3 = 28$, $a_4 = 30$, $a_5 = 30$, $a_6 = 32$, $a_7 = 33$, $a_8 = 34$ et donc une estimation qui vaut $10^2/34 \simeq 2.941176$.

Proposer un programme C qui, étant donné une variable entière `n` strictement positive, calcule une estimation flottante de `pi`.

57. Suite A000073 de l'OEIS.
58. Précisons que la convergence est très lente !

### Ex. 2.9 – Spirale de nombres

Considérons le placement des entiers de 1 à $n^2$ dans une grille de taille $n \times n$ en partant du centre et en évoluant en « spirale », de la sorte :

| | | | | |
|---|---|---|---|---|
| **25** | 10 | 11 | 12 | **13** |
| 24 | **9** | 2 | **3** | 14 |
| 23 | 8 | **1** | 4 | 15 |
| 22 | **7** | 6 | **5** | 16 |
| **21** | 20 | 19 | 18 | **17** |

Proposer un programme déterminant, pour un $n$ donné, la somme des entiers situés sur les diagonales (en gras ci-dessus).

### Ex. 2.10 – Simplifications abusives

Il existe des quadruplets de chiffres non nuls $(a, b, c, d) \in [\![1..9]\!]^4$ tels que

$$\frac{a}{d} = \frac{ab}{cd} = \frac{abb}{ccd} = \frac{abbb}{cccd} = \ldots$$

où les écritures telles que $abb$ doivent être comprises comme le nombre résultant de la concaténation des chiffres.

Par exemple, le quadruplet $(3, 9, 6, 5)$ vérifie :

$$\frac{3}{5} = \frac{39}{65} = \frac{399}{665} = \frac{3999}{6665} = \ldots$$

Proposer un programme affichant l'ensemble de tels quadruplets où les quatres chiffres sont *distincts*.

### Ex. 2.11 – Déplacement d'une reine d'échecs

On s'intéresse à un échiquier sur lequel est posé une unique reine sur la case $(i_1, j_1)$, pouvant se déplacer d'un nombre quelconque de cases en ligne, en colonne, ou en diagonale. Proposer une fonction « `int nb_moves(int i1, int j1, int i2, int j2)` » renvoyant le nombre de déplacements nécessaires pour qu'elle se retrouve en $(i_2, j_2)$.

### Ex. 2.12 – Fractions

Proposer une fonction « `bool frac_equals(int a, int b, int c, int d)` » prenant en argument quatre entiers (`b` et `d` étant non nuls) et renvoyant un booléen indiquant si les deux fractions $\frac{a}{b}$ et $\frac{c}{d}$ sont égales.

## Ex. 2.13 – Nᵉ chiffre

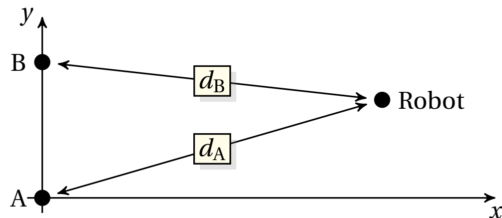
On construit une suite de chiffres en énumérant les chiffres des entiers de $\mathbb{N}$ :

12345678910111213141516171819202122223...

Proposer une fonction « `int nth(int n)` » renvoyant le chiffre en position `n` (on supposera que l'indexation débute à 0, aussi « `nth(5)` » devra renvoyer 4, et « `nth(14)` » devra renvoyer 2). On pourra envisager différentes approches et comparer leurs efficacités en terme de temps de calcul.

## Ex. 2.14 – Localisation spatiale d'un robot

Un robot peut se déplacer librement dans le plan $(O, \vec{e}_x, \vec{e}_y)$. Pour se situer, il dispose de deux balises situées aux points A de coordonnées $(0,0)$ et B de coordonnées $(0,1)$, et est en mesure de déterminer (par exemple par temps de vol) les distances $d_A$ et $d_B$ qui le séparent de ces deux balises.




**1.** Proposer une fonction « `double pos_y(double dA, double dB)` » qui, à partir de ces deux distances (supposées positives), renvoie l'ordonnée $y$ du robot dans le plan.

**2.** Existe-t-il des valeurs de $d_A$ et $d_B$ positives mais incohérentes? Comment avez-vous géré ce cas particulier dans votre fonction?

**3.** Est-il possible de déterminer l'abcisse $x$ du robot?

## Ex. 2.15 – Retournement

Proposer une fonction « `int flip(int n)` » prenant un entier strictement positif $n$ et renvoyant l'entier $p$ correspondant au retournement de l'écriture décimale de $n$. Par exemple, `flip(137)` devra renvoyer 731, et `flip(8320)` devra renvoyer 238.

On pourra supposer que $p$ est représentable et ignorer d'éventuels problèmes de débordement.

## Ex. 2.16 – Nombres particuliers

**1.** Déterminer s'il existe un ou plusieurs nombre(s) entier(s) $n$ strictement plus grand(s) que 1, dont l'écriture décimale $c_0 c_1 c_2 ... c_k$, constituée de chiffres $c_i$, vérifie [59] $n = \Sigma_i c_i^{c_i}$.

Par exemple, ce n'est pas le cas de 137, car $1^1 + 3^3 + 7^7 \neq 137$.

**2.** Prouver que l'on a bien identifié l'ensemble des éventuelles solutions (nécessairement sans pouvoir tester chacun des éléments de $\mathbb{N}$!)

**3.** Même chose pour $n = \Sigma_i c_i!$, pour $n > 2$ (où ! désigne la factorielle).

## Ex. 2.17 – Diviseurs et nombres parfaits

**1.** Écrire une fonction « `bool is_prime(int n)` » prenant en argument un entier strictement positif `n`, et déterminant si `n` est un nombre premier. On pourra simplement regarder si des entiers de plus en plus grands sont des diviseurs de $n$, mais s'efforcera de s'arrêter dès que l'on peut conclure. On s'abstiendra de manipuler des flottants afin d'éviter d'éventuels problèmes de précision.

**2.** Cette fonction renvoie-t-elle un résultat correct pour *absolument* tout entier strictement positif entre 1 et `INT_MAX`? Si ce n'est pas le cas comment la corriger [60] ?

**3.** Proposer une fonction « `bool no_mult(int n)` » déterminant si la multiplicité [61] de tous les diviseurs premiers d'un entier `n` strictement positif est 1. Indication : il n'est *pas* nécessaire de déterminer la décomposition en facteurs premiers de `n`!

**4.** Écrire une fonction « `int sum_divisors(int n)` » renvoyant la somme des diviseurs propres [62] d'un entier `n` positif.

Un *nombre parfait* est un entier strictement positif égal à la somme de ses diviseurs propres. 28 est un exemple d'un tel entier (car $28 = 1 + 2 + 4 + 7 + 14$).

**5.** En déduire une fonction « `bool is_perfect(int n)` » déterminant si `n` est un nombre parfait.

59. Dans la mesure où il n'y a pas de consensus sur la valeur de $0^0$, on ajoutera la condition que les chiffres $c_i$ sont tous non nuls.

60. Il peut être particulièrement complexe de garantir la correction d'une fonction pour tous les entiers positifs représentables, il n'est pas inconcevable dans la pratique de parfois se limiter à des entiers « pas trop grands », mais il conviendra de préciser explicitement le domaine de validité dans la documentation de la fonction.

61. La multiplicité d'un diviseur premier pour un entier $n$ est l'exposant qui lui est associé dans la décomposition en facteurs premiers de $n$. Par exemple, pour $n = 360 = 2^3 \times 3^2 \times 5$, les multiplicités associes à 2, 3 et 5 sont respectivement 3, 2 et 1.

62. Diviseurs entre 1 et $n-1$.

## Ex. 2.18 – Paires amicales et chaînes aliquotes

Une *paire amicale* est un couple d'entiers $(n, n')$ *distincts*, où $n$ et la somme des diviseurs propres de $n'$ et $n'$ la somme des diviseurs propres de $n$. Le couple $(220, 284)$ est un exemple de paire amicale, connue depuis l'antiquité.

**1.** Écrire une fonction « `bool is_amicable(int n)` » déterminant si un entier positif `n` est membre d'une paire amicale. On pourra se servir des fonctions de l'exercice précédent.

**2.** Écrire un programme identifiant les cinq paires amicales de nombres ayant au plus quatre chiffres [63].

Une *chaîne aliquote* de longueur $k > 1$ est un ensemble $u_0, u_1, u_{k-1}$ de $k$ entiers distincts tels que, pour tout $i$ dans $[\![1 .. k-1]\!]$, $u_i$ est la somme des diviseurs premiers de $u_{i-1}$, et $u_0$ la somme des diviseurs premiers de $u_{k-1}$. Un exemple de telle chaîne est l'ensemble $\{12496, 14288, 15472, 14536, 14264\}$. Notons qu'une chaîne aliquote de longueur 2 constitue une paire amicale.

**3.** Proposer un programme déterminant la plus longue chaîne aliquote d'entiers tous inférieurs à $10^6$.

Une suite récurrente telle que $u_{n+1}$ est la somme des diviseurs de $u_n$ est dite *suite aliquote*. Ces suites font l'objet de nombreuses études à l'heure actuelle, et sont encore mal connues [64]. La conjecture de Catalan-Dickson, énoncée en 1888, affirme que toute suite aliquote est bornée, et donc finit par atteindre 1 (le terme précédent étant premier), un entier parfait ou membre d'une chaîne aliquote. Inversement, la première conjecture de Garambois affirme qu'une suite aliquote démarrant sur un entier pair a une chance sur 3 de croître indéfiniment. Les « cinq » de Lehmer sont cinq entiers (276, 552, 564, 660, 966) pour lesquels on ne sait pas encore s'ils conduisent à une croissance infinie ou non.

## Ex. 2.19 – Triplets pytagoriciens

**1.** Proposer une fonction « `int triplets(int n)` » prenant en argument un entier positif et renvoyant le nombre de triplets d'entiers $(a, b, c)$ positifs ou nuls vérifiant $a^2 + b^2 = c^2$ avec $a \leqslant b$ et $a + b + c = n$ (on ne se préoccupera pas des dépassement).

**2.** Si ce n'est pas déjà le cas, pouvez-vous écrire une fonction, plus efficace, qui n'utilise qu'une seule boucle et teste moins de $\lceil n/2 \rceil$ triplets ?

63. On pourra tenter d'aller plus loin, il y a par exemple 42 paires amicales dont le plus petit entier du couple est inférieur à $10^6$. Mais plus les nombres seront grands, plus il faudra réfléchir à la meilleure façon de procéder pour que les calculs prennent un temps raisonnable. On trouvera une base de données des paires amicales connues à l'adresse `sech.me/ap`.

64. On pourra trouver des informations sur l'état des recherches sur le site de Jean-Luc Garambois, `www.aliquotes.com`.

## Ex. 2.20 – Algorithme de Stein

Une autre façon de déterminer le PGCD de deux entiers positifs `a` et `b` est l'algorithme de Stein. Il repose sur les affirmations suivantes :

- $\text{pgcd}(a, 0) = a$ et $\text{pgcd}(0, b) = b$ ;
- $\text{pgcd}(a, b) = 2\,\text{pgcd}(a/2, b/2)$ lorsque $a$ et $b$ sont pairs ;
- $\text{pgcd}(a, b) = \text{pgcd}(a/2, b)$ lorsque $a$ est pair et $b$ impair ;
- $\text{pgcd}(a, b) = \text{pgcd}(a, b/2)$ lorsque $a$ est impair et $b$ pair ;
- $\text{pgcd}(a, b) = \text{pgcd}(|a - b|, \min(a, b))$ lorsque $a$ et $b$ sont impairs.

**1.** Proposer une fonction « `int GCD_Stein(int a, int b)` » prenant en argument deux entiers positifs et renvoyant leur PGCD par l'algorithme de Stein.

**2.** Justifier que le nombre d'itérations de l'algorithme de Stein, lors du calcul de $\text{pgcd}(a, b)$ pour $a > 0$ et $b > 0$, est majoré par $1 + 2\left\lceil \log_2(a) + \log_2(b) \right\rceil$.

## Ex. 2.21 – Nombres avec zéro(s)

Un nombre est dit « avec zéro » s'il contient un zéro dans sa représentation décimale usuelle. Par exemple : 40, 100, 107, 10203, etc.

**1.** Proposer une fonction « `bool has_zero(int n)` » indiquant si l'entier $n$, supposé strictemet positif, contient un zéro dans sa représentation décimale usuelle.

**2.** Proposer une fonction « `int count(int p)` » prenant un entier $p > 1$ et renvoyant combien de tels nombres d'au plus $p$ chiffres existent (c'est-à-dire des nombres compris entre 1 et $10^p - 1$).

**3.** Peut-on écrire cette fonction sans tester tous les entiers entre 1 et $10^p - 1$ ? Si oui, comment ?

**4.** Modifier ces deux fonctions pour qu'elles prennent un second argument entier supérieur ou égal à 2, noté `b`, indiquant la base dans laquelle on travaille (par exemple, les entiers 37, 42 et 53 s'écrivent respectivement $1101_3$, $1120_3$ et $1222_3$ en base 3, aussi `has_zero(37, 3)` et `has_zero(42, 3)` devront renvoyer `true`, mais `has_zero(53, 3)` devra renvoyer `false`). Pour `count`, on étudiera les entiers de 1 à $b^p - 1$.

## Ex. 2.22 – Racine entière

Pour un entier $p$ positif, on souhaite déterminer la *racine carrée entière* de $p$, c'est-à-dire l'entier $\lfloor \sqrt{p} \rfloor$ (où $\lfloor \bullet \rfloor$ désigne la fonction *partie entière*), **sans recourir aux flottants**. Pour ce faire, on s'intéresse à la suite $(u_n)_{n \in \mathbb{N}}$ d'entiers définie par

$$u_0 = p \qquad \text{et} \qquad u_{n+1} = \left\lfloor \frac{u_n^2 + p}{2u_n} \right\rfloor \quad \text{pour tout } n \geqslant 0$$

Le premier terme $u_i$ de la suite vérifiant $u_i \leqslant u_{i+1}$ est tel que $u_i = \lfloor\sqrt{p}\rfloor$.

**1.** Déterminer les termes de la suite pour $p = 37$ de $u_0$ à $u_{i+1}$, et vérifier que $u_i = \lfloor\sqrt{37}\rfloor$.

**2.** Proposer une fonction « `int isqrt(int p)` » utilisant cette suite pour déterminer et renvoyer la racine carrée entière de son argument `p`. On supposera `p` positif et que les calculs ne provoquent pas de dépassement.

On souhaite à présent montrer la correction de la fonction (et que $u_i = \lfloor\sqrt{p}\rfloor$).

**3.** Montrer qu'il existe nécessairement un $i$ tel que $u_{i+1} \geqslant u_i$, et en déduire que la fonction `isqrt` termine.

**4.** Montrer que pour tout réel $t$ positif, on a $\frac{t^2+p}{2t} \geqslant \sqrt{p}$.

**5.** En déduire que $u_i \geqslant \lfloor\sqrt{p}\rfloor$.

**6.** Conclure sur la correction de la fonction `isqrt`.

## Ex. 2.23 – Fléchettes

Selon la règle du jeu de fléchettes « 501 », les joueurs partent d'un score de 501 et déduisent de ce score les points marqués en envoyant les fléchettes sur la cible.

La cible contient des zones rapportant de 1 à 20 points, ainsi qu'une zone rapportant 25 points. Dans la zone de la cible rapportant 25 points, il existe une zone où les points sont doublés (rapportant ainsi 50 points).

De même, dans chacune des zones rapportant 1 à 20 points, il existe une zone où les points sont doublés et une zone où les points sont triplés.

Le but du jeu est d'atteindre *exactement* le score de 0, les derniers points devant impérativement provenir d'une zone de points *doublés*.

**1.** Proposer une fonction prenant en argument un entier positif et indiquant, en renvoyant un booléen, s'il est possible de gagner avec une volée d'au plus trois fléchettes.

Par exemple, il est possible de gagner en partant de 3 (une fléchette dans la zone « 1 point » et une fléchette dans la zone « 2 × 1 points »), ou bien de 161 (« 3 × 20 points », « 3 × 17 points » puis « 2 × 25 points »). Il est en revanche impossible de terminer en partant d'un score de 1 de ou 168.

**2.** En déduire un programme déterminant combien de scores différents permettent de terminer une partie en une volée d'au plus trois flèches.

## Ex. 2.24 – Poinçon d'orientation

Un poinçon d'orientation se présente sous la forme d'une pince équipée de $n$ aiguilles, destinée à créer un motif de « trous » dans une carte. Ces $n$ aiguilles peuvent être placées librement dans une grille de $p \times q$ emplacements, permettant ainsi de créer de nombreux motifs.

Proposer une fonction `int nb_shapes(int n, int p, int q)` prenant en argument les entiers $n$, $p$ et $q$ et renvoyant le nombre de motifs *distincts*[65] qu'il est possible de réaliser en plaçant différemment les $n$ aiguilles.

Pour simplifier (un peu), on pourra supposer $p \leqslant 5$ et $q \leqslant 5$ (ce qui signifie que seules les rotations d'angle $k\pi/2$ sont à considérer).

65. Deux dispositions qui ne diffèrent que par une translation, une rotation ou une symétrie donnent un même motif.

# Les tableaux en C

> « Should array indices start at 0 or 1? My compromise of 0.5 was rejected without, I thought, proper consideration. »
>
> — *Stan Kelly-Bootle*

## 1 Création et manipulation de tableaux

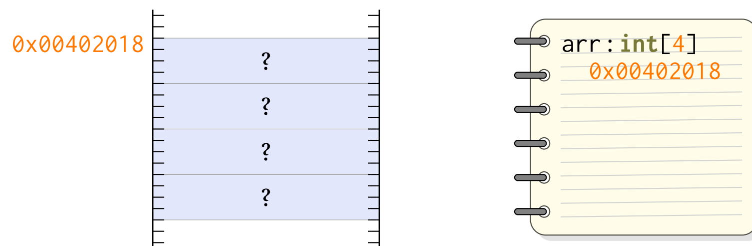
### 1.1 Déclaration

Si l'on devait réserver séparément pour *chaque* donnée manipulée une case en mémoire et lui attribuer un nom, il serait proprement impossible de traiter de gros volumes d'information. Il est donc indispensable de pouvoir déclarer d'un coup un grand nombre de données et de les manipuler.

Les tableaux sont une des solutions à ce problème. Déclarer un tableau revient à réserver de la place en mémoire pour N données de même type. La déclaration d'un tableau s'effectue de façon très similaire à la déclaration d'un unique élément du type de ses éléments. On spécifie simplement, entre crochets, après le nom qui servira à désigner le tableau, le nombre d'éléments que l'on souhaite pouvoir mémoriser (concrètement, le nombre de « cases » du tableau). Ainsi, le morceau de programme suivant réserve de la place en mémoire pour un tableau contenant quatre entiers :

```
int arr[4];
```

Les éléments seront placés en mémoire de façon contiguë, sans laisser d'espace entre ces derniers. Ainsi, la déclaration précédente conduit, en mémoire, à ce résultat :




*Mémoire*

Le nombre d'éléments que contiendra un tableau C doit être une information *disponible lors de la compilation*. On ne peut donc pas utiliser le contenu d'une variable pour indiquer la taille d'un tableau [1], donc « `int arr[n]` » ne définit pas un tableau quand bien même `n` serait une variable entière [2].

Le compilateur mémorise la taille des tableaux créés. Cela lui est indispensable car, comme toute variable, les tableaux ont une portée, et viendra un moment où la mémoire qui leur a été allouée devra être libérée. Il faut donc mémoriser où cette zone mémoire commence, mais également où elle se termine.

Plutôt que de mémoriser le nombre d'éléments dans le tableau, le compilateur note en fait le nombre de « cases mémoires » ou, plus précisément, le nombre de *bytes* qu'il occupe, un byte étant défini comme l'unité de stockage élémentaire en mémoire [3]. Il est possible d'obtenir cette information dans un programme grâce à la « commande » `sizeof`. Dans le cadre de notre exemple, « `sizeof(arr)` » fournira probablement la valeur 16 : le tableau contient en effet quatre `int`, et chaque `int` occupe, dans la plupart des architectures, quatre bytes en mémoire.

En dépit de son apparence, `sizeof` n'est pas une fonction : l'information est disponible lors de la compilation, et les valeurs sont directement déterminées par le compilateur [4]. Cette construction très versatile ne prend pas seulement des noms de variables en « argument ». Ainsi, on peut tout aussi bien écrire :

- « `sizeof(x)` » où `x` est un nom de variable [5] (un tableau `arr`, un entier `mango`...), le résultat sera le nombre de bytes occupés en mémoire par la variable [6] ;
- « `sizeof(int)` » ou « `sizeof(double)` », le résultat étant le nombre de bytes occupés en mémoire par un objet du type spécifié ;
- « `sizeof(42)` » ou « `sizeof(3.14)` », le résultat étant alors le nombre de bytes qu'occuperait la valeur indiquée si elle était stockée en mémoire.

On peut même, en théorie, utiliser une expression comme argument de `sizeof` mais attention : si l'on écrit par exemple « `sizeof(foo(42)+37)` », le compilateur ne se préoccupe que du *type* du résultat, il détermine donc lors de la compilation le type qu'aurait

1. Il existe une autre variété de tableaux, dit *tableau de longueur variable* ou *variable-length array* (VLA) qui s'affranchit de cette condition, mais leur utilisation induit d'autres contraintes. Ces tableau particuliers ne sont pas au programme, et leur support n'est pas garanti dans la majorité des versions du langage C, mais nous y reviendrons brièvement ultérieurement.
2. Même déclarée, comme le langage C le permet, « constante ».
3. Dans la quasi-totalité des architectures modernes, un byte correspond à un *octet*, c'est-à-dire que chaque case en mémoire contient huit bits. Dorénavant, il est même d'usage d'assimiler byte et octet, quitte à être prudent lorsque l'on travaille avec des architectures exotiques.
4. Avec de très rares exceptions : les tableaux de longueur variable. Dans ce cas, le compilateur est forcé d'ajouter des instructions permettant d'obtenir, lors de l'exécution du programme, la taille occupée en mémoire.
5. Dans ce cas, et également dans le cas où `sizeof` est appliqué à une expression, les parenthèses ne sont pas requises et il est permis d'utiliser « `sizeof x` » (de même que « `sizeof 42` »). Certains même le recommandent. Toutefois, la question de la précédence peut parfois se poser (`sizeof` a la même priorité que la négation unaire), et la présence de parenthèses peut occasionnellement éviter un malentendu.
6. Même si le compilateur décidait de ne jamais ranger cette valeur en mémoire et de la conserver dans un registre, il s'agirait de la place qu'*occuperait* cette variable en mémoire.

le résultat de l'évaluation de « `foo(42)+37` » pour en déduire la taille mémoire nécessaire. Mais l'expression *ne sera jamais évaluée*!

En combinant ces différentes possibilités, on peut retrouver le nombre d'éléments contenus dans un tableau `arr` contenant des éléments de type `int` en écrivant [7], profitant du caractère contigu du stockage des éléments en mémoire :

```
sizeof(arr)/sizeof(int)
```

En dehors de cette circonvolution, il n'existe aucun moyen d'obtenir, dans un programme, la dimension d'un tableau. Il est donc fréquent de mémoriser la dimension des tableaux que l'on manipule en C dans une variable.

## 1.2 Initialisation

Comme n'importe quelle variable, lire une case d'un tableau sans y avoir préalablement rangé une variable est formellement interdit [8]. On dispose donc de mécanismes pour initialiser un tableau dès sa déclaration. Ces mécanismes sont réservés à l'initialisation, car nous verrons qu'il n'est pas possible d'affecter plusieurs valeurs d'un coup à un tableau.

Pour spécifier toutes les valeurs d'un tableau lors de sa déclaration, on écrira la séquence des valeurs qu'il doit contenir entre accolades, séparées par des virgules :

```
int arr[4] = { 17, 37, 42, 54 };
```

La commande précédente réserve donc quatre emplacements consécutifs en mémoire, et place les valeurs 17, 37, 42 et 54 dans chacune des cases.

Un tableau ne peut pas être « partiellement » initialisé. Si l'on initialise une partie d'un tableau, *tout* le tableau est initialisé. Si l'on fournit un nombre de valeurs insuffisant, les autres cases sont initialisées avec une valeur par défaut (0 pour des `int`, 0.0 pour des `double`, `false` pour des booléens...) Ainsi,

```
int arr[4] = { 17, 37 };
```

réserve quatre cases, et place dans chacune de ces cases les valeurs 17, 37, 0 et 0.

Précisons qu'une initialisation vide (« `{}` ») n'est pas permise en C. La façon la plus succinte d'initialiser l'ensemble d'un tableau à 0 est d'écrire :

```
int arr[4] = { 0 }; // Toutes les cases initialisées à 0
```

7. Pour éviter d'avoir besoin de connaître le type des éléments du tableau, on peut également écrire « `sizeof(arr)/sizeof(arr[0])` ».

8. Chaque case est considérée indépendamment, pour pouvoir lire le contenu d'une case, il faut avoir préalablement écrit quelque chose spécifiquement dans *cette* case.

Inversement, si l'on fournit tous les éléments, le langage C permet que l'on ne spécifie pas la taille du tableau (donc sans rien mettre entre les crochets), il la déduira du nombre d'éléments dans l'initialisation. Ainsi, la déclaration suivante est une déclaration valable d'un tableau initialisé de taille 4 :

```
int arr[] = { 17, 37, 42, 54 };
```

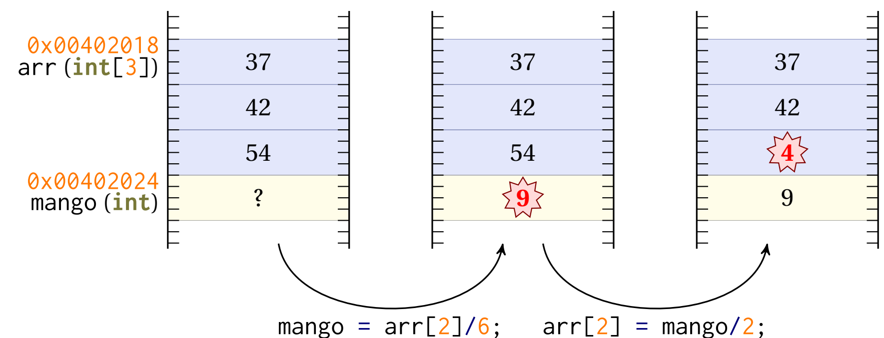
## 1.3 Manipulation des éléments

Pour désigner l'emplacement mémoire correspondant à une « case » d'un tableau, on fait suivre le nom désignant le tableau de crochets encadrant un entier correspondant à l'index de la case dans le tableau. Comme dans deux nombreux langages, l'indexation débute à 0 [9]. Ainsi, on fait référence au troisième élément d'un tableau `arr` en écrivant « `arr[2]` ». Cette écriture se comporte comme un nom de variable classique, et permet tout autant de d'extraire une valeur d'une case du tableau que de ranger une valeur dans une de ses cases.

Ainsi, le programme ci-dessous :

```
int arr[3] = { 37, 42, 54 };
int mango;

mango = arr[2]/6;
arr[2] = mango/2;
```

se déroule de la façon suivante en mémoire :




L'index peut parfaitement être une expression, mais l'évaluation de l'expression doit impérativement donner un entier. Un flottant, par exemple, *ne sera pas* converti en un

9. Nous aurons l'occasion d'en expliquer les raisons dans le prochain chapitre, mais on trouve trace des arguments de l'informaticien Edsger Dijkstra en 1981 en faveur de ce choix à l'adresse `www.cs.utexas.edu/users/EWD/ewd08xx/EWD831.PDF`.

entier, comme c'est le cas lorsque l'on passe un flottant à une fonction attendant un entier par exemple.

Par ailleurs, **l'index doit impérativement correspondre à une case valide du tableau**. C'est-à-dire que, pour un tableau à $n$ cases, cet index doit être positif ou nul [10] et strictement inférieur à $n$. Si ce n'est pas le cas, le comportement du programme est *indéfini*, c'est-à-dire qu'il peut se passer rigoureusement n'importe quoi.

## 1.4 Un mot sur le comportement indéfini en C

Dans la plupart des langages, le compilateur (ou l'interpréteur) ajoute quelques instructions vérifiant que l'on ne « déborde » pas d'un tableau avant de tenter d'accéder à une de ses cases, instructions qui produisent une erreur, interrompant généralement l'exécution, lorsque l'index ne désigne pas une case valide. S'ils agissent de la sorte dans tous les cas qui pourraient éventuellement poser des problèmes, ces langages sont qualifiés de *sûrs*.

Cependant, la présence de ces instructions ralentit très légèrement l'exécution du programme, aussi les créateurs du langage C ont-il pris une décision radicalement différente en décidant de considérer que *tout* comportement du programme, dans une telle situation, est admissible, quand bien même les conséqences pourraient être graves. À charge du programmeur, donc, d'ajouter lui-même des tests vérifiant le non-débordement s'il juge la chose utile [11]. Le langage C n'a donc rien d'un langage sûr !

Lorsque l'on effectue une opération interdite, la norme C dit que le comportement du programme est *indéfini* (*undefined behavior* en anglais, souvent abrégé en « UB »). Ce qui se passe est décrit dans la FAQ du langage : en gros, *tout* peut arriver, même que, par le plus grand des hasards, le programme fonctionne quand même [12].

> « Anything at all can happen; the Standard imposes no requirements. The program may fail to compile, or it may execute incorrectly (either crashing or silently generating incorrect results), or it may fortuitously do exactly what the programmer intended. »

Nous avons déjà croisé plusieurs situations conduisant à un comportement indéfini du programme [13]. Sans trop entrer dans les détails, les situations suivantes sont potentiellement problématiques :

- terminer un programme par une ligne non vide ;
- tenter d'obtenir la valeur d'une variable non-initialisée ;
- tenter d'accéder, en lecture ou en écriture, à une case d'un tableau avec un index négatif ou bien supérieur ou égal à la taille du tableau ;
- parvenir au bout d'une fonction (autre que `main`) déclarée comme renvoyant une valeur sans rencontrer de `return` ;
- effectuer une opération arithmétique illégale (telle qu'une division par 0) ;
- effectuer un calcul (ou une conversion) dont le résultat n'est pas représentable par le type manipulé (par exemple un calcul sur des `int` excédant `INT_MAX`) ;
- utiliser des expressions dont le résultat de l'évaluation dépend de l'ordre dans lequel les opérations sont effectuées.

Cette idée de lever toute contrainte au compilateur lorsque l'on se retrouve dans une situation problématique est une caractéristique quasiment unique aux langages C et C++. Certains langages précisent que, dans certains cas, le comportement exact du programme n'est pas parfaitement précisé. Par exemple, lorsque l'on effectue un calcul dont le résultat n'est pas représentable, on ne sait pas quel sera le résultat fourni par le programme, mais il y aura bien un résultat, et le programme poursuivra son exécution normalement.

Ce n'est pas du tout ce que prévoit la norme du langage C : si le programme est amené à effectuer une opération illégale, absolument tout comportement du programme produit est admissible : comme le dit la FAQ, il peut s'arrêter, donner un résultat incorrect, accidentalement correct, effacer votre disque dur, détruire votre ordinateur, lancer un missile nucléaire ou faire apparaître des démons dans vos narines [14].

Naturellement, le compilateur ne produira généralement pas sciemment un programme avec un comportement aussi néfaste qu'effacer le disque dur, ou lancer un missile nucléaire. Cependant, le code généré peut très bien avoir accidentellement ces conséquences [15], c'est pourquoi il convient d'être extrêmement prudent.

Un autre point important est que si l'on se retrouve à devoir exécuter une opération illégale, c'est la *totalité* du programme qui a un comportement indéfini. Y compris potentiellement ce qui *précède* l'instruction concernée !

Si, par exemple, le compilateur trouve une fonction renvoyant un `int` qui se termine par « `return INT_MAX+1;` » ou bien sans `return`, comme en vertu de la norme une telle chose est classé comme provoquant un comportement indéfini, il est parfaitement en droit de ne pas compiler la fonction incriminée, et de remplacer tous les appels à cette fonction par 42 (ou de jouer la traviata à chaque appel). Et ce, sans vous prévenir, évidemment [16].

---

10. Il n'est pas permis d'utiliser des index négatifs comme c'est le cas en Python.
11. Compte tenu de l'importance de tels tests pour garantir le bon fonctionnement de certains programmes, la plupart des compilateurs fournissent des options de compilation pour rajouter automatiquement de tels tests de non-débordement : il s'agit par exemple de l'option `-fsanitize=address` dans le cas des compilateurs `gcc` et `clang` (cette option détecte aussi d'autres problèmes d'accès à la mémoire).
12. Cela arrive régulièrement en pratique, et c'est de cette façon que certains bugs passent inapperçus pendant des années jusqu'à ce qu'ils causent une catastrophe !
13. Et nous en verrons bien d'autres… Une liste indicative relativement exhaustive se trouve dans l'annexe J du document définissant la norme C99.
14. Ce dernier exemple très imagé est celui utilisé sur les listes de discussions du langage pour illustrer qu'aucun comportement n'est exclu, fût-il improbable. Voir par exemple `www.catb.org/jargon/html/N/nasal-demons.html` .
15. un simple `i++` dans un programme avec plusieurs threads peut très bien substituer accidentellement un appel de fonction par un autre, et donc remplacer l'allumage d'un voyant dans la chambre de contrôle d'un site militaire par le lancement d'un des missiles.
16. Une fonction activant un mécanisme de sécurité important a été ainsi purement et simplement supprimée par le compilateur d'un programme développé par Google, et il s'est passé des mois avant que cette disparition soit remarquée. L'incident est reporté à cette adresse : `bugs.chromium.org/p/nativeclient/issues/detail?id=245` . Noter que le décalage incriminé a effectivement un comportement indéfini, et contrairement à ce qui avancé avec précautions dans l'article, le résultat du décalage ne sera pas nécessairement 1.

Toujours pour ces mêmes raisons, si l'on définit une fonction ainsi :

```c
int foo(int i) {
    if (i != 0) {
        printf("Hello World!");
    }
    return 100/i;
}
```

Le compilateur est parfaitement en droit de considérer que vous avez écrit :

```c
int foo(int i) {
    printf("Hello World!");
    return 100/i;
}
```

provoquant donc un affichage du message même si i est nul, avant même que l'on ne parvienne à la division qui pose problème. En effet, si vous appelez la fonction foo avec un argument non nul, le changement n'a pas de conséquence. Si vous appelez la fonction foo avec un argument nul, le comportement de l'ensemble du programme n'étant pas défini, afficher le message ne lui est donc pas interdit.

On le comprend, l'existence de comportements indéfinis est particulièrement préjudiciable pour le programmeur. Qu'y gagne-t-on? Principalement des programmes produits plus efficaces, ce qui est le but avoué du langage. Prenons quelques exemples.

L'une des situations déclenchant des comportements indéfinis est le débordement des capacités pour les entiers. La raison en est que si le processeur permet de représenter des entiers jusque $2^{31}-1$, si l'on incrémente une telle valeur dans le processeur, certains processeurs donneront la valeur $-2^{31}$, d'autres [17] conserveront cette valeur $2^{31}-1$. En ne précisant pas ce qu'il se passe dans la norme, le compilateur peut utiliser le résultat fourni par le processeur sans se poser de question et donc sans ajouter de test supplémentaire.

Cela permet au compilateur des optimisations : s'il rencontre une expression [18] « mango * 2 / 2 », le compilateur a le droit de la simplifier en « mango ». Ou bien « cherry+1 > cherry » en « true ». En effet, s'il n'y a pas de débordement, ces simplifications sont correctes, et si un débordement pourrait se produire, il est en droit de faire rigoureusement ce qu'il veut.

Il suffirait cependant de dire que les résultats, lorsqu'il y a débordement, fournit une valeur qui n'est pas clairement définie. Mais avec la notion de comportement indéfini, on va plus loin. Le compilateur peut faire l'hypothèse très forte que le programme ne contient aucune situation interdite (de toute façon, s'il y en a, tout code produit est acceptable!) Par conséquent, il peut librement supprimer divers morceaux de code tant qu'ils ne servent à rien dans les situations où tout se passe normalement. C'est par exemple sur cette base que le test de l'exemple du dessus peut être éliminé.

17. Utilisant une arithmétique dite *à saturation*.
18. Ce qui, pour diverses raisons, peut arriver assez souvent.

Le but de ce cours n'est aucunement d'étudier en détail tous ces comportements indéfinis, ou les subtilités de la compilation, qui est un processus particulièrement complexe. On gardera simplement en tête que certaines opérations, que nous signalerons au fur et à mesure que nous les rencontrerons, sont interdites en langage C, et qu'il faut y prendre garde, car la conséquence ne se restreint pas à une simple erreur, mais conduisent à un comportement imprévisible, et potentiellement dangereux, du programme produit.

## 1.5 Tableaux, affectations, expressions

Seuls les *éléments* d'un tableau peuvent être manipulés dans un programme C, jamais le tableau lui-même. Il n'est ainsi pas possible d'affecter quoi que ce soit à un tableau. Ainsi, si arr est un tableau, il n'est *jamais* possible d'écrire :

```c
arr = ...  // Illégal dans tous les cas si arr est un tableau
```

Si arr1 et arr2 sont deux tableaux de même taille $n$, si l'on souhaite recopier les éléments de arr2 dans arr1, il n'est notamment pas possible d'écrire « arr1 = arr2 », et on est contraint de copier les éléments un par un :

```c
for (int i=0; i<n; ++i) {
    arr1[i] = arr2[i];      // arr1 ← arr2
}
```

De la même façon, les tableaux ne peuvent pas non plus être directement utilisés dans des expressions. Notablement, leurs contenus ne peuvent **pas** être directement comparés en écrivant « arr1 == arr2 ». Mais il est possible de vérifier si les contenus de deux tableaux de même taille n sont égaux en écrivant [19] :

```c
bool content_equals = true;
for (int i=0; i<n && content_equals; ++i) {
    content_equals = content_equals && arr1[i] == arr2[i];
}
// content_equals contient true si et seulement si
// les contenus des cases sont tous égaux deux à deux
```

Profitons-en pour rappeler que, les variables en C, fussent-elles des tableaux, étant liées à des zones mémoire spécifiques, deux noms tels que arr1 et arr2 ne peuvent jamais correspondre aux *mêmes* données en mémoire, tout au plus à des données égales.

19. On notera la sortie anticipée de la boucle for dès que content_equals reçoit false.

### 1.6 Tableaux et fonctions

Nous avons dit précédemment que le passage des argument d'une fonction était effectué grâce à une copie dans la mémoire. Pour passer un tableau comme argument à une fonction, il faudrait donc recopier l'intégralité du tableau, ce qui serait particulièrement coûteux pour des tableaux de grande taille.

Quand bien même on serait prêt à assumer un tel coût, le langage C ne permet pas qu'un tableau figure parmi les paramètres d'une fonction [20]. On ne peut pas non plus *renvoyer* un tableau car on ne pourrait pas manipuler le résultat.

Nous verrons dans le prochain chapitre un mécanisme permettant de contourner cette difficulté (mécanisme qui donnera l'*apparence*, parfois trompeuse, qu'un tableau peut se retrouver argument d'une fonction). D'ici là, nous éviterons de placer les morceaux de code que nous prendrons exemple dans des fonctions pour éviter tout malentendu.

## 2 Algorithmes élémentaires sur les tableaux

### 2.1 Somme des éléments

Il n'existe pas en C, nous l'avons dit, de manière d'itérer directement sur les éléments d'un tableau. Il est en revanche possible d'effectuer une boucle sur les index des éléments d'un tableau `arr` de taille `n`. Par exemple, pour obtenir la somme de tous les éléments du tableau d'entiers, on peut déclarer une variable entière initialisée à 0 qui servira d'accumulateur et dans laquelle on ajoutera un à un les différents éléments. Cela peut s'écrire :

```c
int accum = 0;
for (int i=0; i<n; ++i) {
    // accum contient la somme des i premiers éléments de arr
    accum += arr[i];
}
// accum contient la somme de tous les éléments de arr
```

Schématiquement, le comportement sera le suivant :

| | | | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| `arr` | 1 | 3 | 5 | 1 | 1 | 3 | 3 | 5 | 1 | 2 | 4 | 2 | 6 | 5 | 7 | 5 | 5 | |
| `accum :` | 0 | 1 | 4 | 9 | 10 | 11 | 14 | 17 | 22 | 23 | 25 | 29 | 31 | 37 | 42 | 49 | 54 | 59 |

itération `i=0`

20. La raison de cette limitation n'est pas parfaitement claire, il semblerait, des dires de son créateur Dennis Ritchie (`www.bell-labs.com/usr/dmr/www/chist.html`), que des questions de compatibilité avec le langage B aient joué un rôle important dans cette décision. La crainte des coûts liés à la copie ne peuvent être la seule raison car nous verrons plus tard qu'un tableau inclus dans une « structure » peut servir d'argument.

### 2.2 Sommes cumulées

Une variante courante de ce problème est le calcul des *sommes cumulées* d'un tableau. Imaginons que l'on ait un tableau `arr` contenant douze entiers correspondant au nombre de jours de chacun des mois de l'année, et que l'on souhaite construire un tableau `cumsum` contenant le total de jours écoulés depuis le début de l'année après chaque mois :

| | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| `arr` | 31 | 28 | 31 | 30 | 31 | 30 | 31 | 31 | 30 | 31 | 30 | 31 |
| `cumsum` | 31 | 59 | 90 | 120 | 151 | 181 | 212 | 243 | 273 | 304 | 334 | 365 |

La démarche est très similaire, on stocke juste les valeurs calculées :

```c
int accum = 0;
int cumsum[12];
for (int i=0; i<n; ++i) { // note : on sait que n=12 ici
    // les cases d'index < i de cumsum ont été remplies
    accum += arr[i];
    cumsum[i] = accum;
}
// toutes les cases de cumsum ont été remplies
```

On pourrait se passer de `accum` en remarquant que `cumsum[0] = arr[0]` et, pour tout $0 < i < n$, `cumsum[i] = cumsum[i-1] + arr[i]`. On traitera alors à part la première case, et on itérera à partir de $i = 1$.

Le calcul peut également être directement effectué *en place*, si on n'a plus besoin des valeurs contenues dans `arr` par la suite, le contenu de `arr` étant alors *remplacé* par les sommes cumulées. Cela donne par exemple :

```c
for (int i=1; i<n; ++i) {
    // les cases d'index < i de arr ont été traitées
    arr[i] += arr[i-1];
}
// toutes les cases ont été traitées
```

### 2.3 Détermination de la valeur maximale

Supposons à présent que l'on ait un tableau `arr` de taille `n`> 0 contenant des entiers et que l'on souhaite en déterminer le maximum. La démarche est similaire et consiste à parcourir le tableau pour examiner les éléments un à un. On va créer une variable `largest` qui contiendra le plus grand élément ayant été vu lors du parcours depuis le début du tableau.

Se alors pose la question de l'initialisation de la variable `largest`. Il n'est pas possible de choisir 0, car si toutes les valeurs du tableau sont négatives, le résultat final, 0, ne serait pas correct. Une solution consiste parfois à utiliser une valeur qui, par définition, est plus petite que toutes les valeurs pouvent figurer dans les cases du tableau. Pour des entiers, on peut donc utiliser `INT_MIN`. Cela s'écrit alors

```
int largest = INT_MIN;
for (int i=0; i<n; ++i) {
    // largest = INT_MIN si i=0
    // largest = max_{0⩽k<i} arr[k] sinon
    if (arr[i] > largest) {
        largest = arr[i];
    }
}
// largest contient le plus grand des éléments de arr
```

| arr | 1 | -3 | -5 | -1 | 3 | 1 | 2 | 4 | -2 | -6 | 5 | 7 | -5 | 5 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| largest : INT_MIN | 1 | 1 | 1 | 1 | 3 | 3 | 3 | 4 | 4 | 4 | 5 | 7 | 7 | 7 |

itération i=0

Mais on peut faire un peu plus simple : initialiser `largest` avec un des éléments du tableau. En général, on choisira le premier élément, qui est toujours disponible pour un tableau non-vide. On n'a alors pas à tester le premier élément et l'itération peut commencer avec la case d'index 1. En outre, l'invariant est plus simple à écrire :

```
int largest = arr[0];
for (int i=1; i<n; ++i) {
    // largest = max_{0⩽k<i} arr[k]
    if (arr[i] > largest) {
        largest = arr[i];
    }
}
// largest contient le plus grand des éléments de arr
```

| arr | 1 | -3 | -5 | -1 | 3 | 1 | 2 | 4 | -2 | -6 | 5 | 7 | -5 | 5 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| largest : | 1 | 1 | 1 | 1 | 3 | 3 | 3 | 4 | 4 | 4 | 5 | 7 | 7 | 7 |

itération i=1

Supposons à présent que nous cherchions le plus grand élément parmi une *partie* seulement des éléments du tableau, par exemple les éléments pairs. Cela change de nombreuses choses.

Tout d'abord, il est possible que tous les éléments soient impairs, ce qu'il nous faudra détecter. On ne peut directement utiliser la première approche car `INT_MIN` est *possiblement* (mais pas nécessairement) pair, et il ne serait pas possible de faire la différence entre un tableau consistant uniquement en des entiers impairs et un tableau où toutes les valeurs paires sont égales à `INT_MIN`.

La seconde approche ne convient pas non plus, car si le premier élément du tableau est impair et que tous les éléments pairs du tableau sont plus petits que cet élément, le résultat obtenu ne sera pas le bon.

Une solution consiste à utiliser un booléen `found_even` supplémentaire indiquant si on a trouvé, parmi les cases déjà examinées, au moins un entier pair dans le tableau. D'une part, cela nous renseignera sur le fait que l'on ait pu obtenir un résultat une fois tout le tableau parcouru, mais surtout, lorsque l'on rencontre un entier pair et que ce booléen est à `false`, on peut directement le placer dans `plus_grand_pair`.

| arr | 1 | -3 | -5 | -1 | 3 | 1 | 2 | 4 | -2 | -6 | 5 | 7 | -5 | 5 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| found_even : | F | F | F | F | F | F | F | T | T | T | T | T | T | T |
| largest_even : | ? | ? | ? | ? | ? | ? | ? | 2 | 4 | 4 | 4 | 4 | 4 | 4 |

itération i=0

Cela s'écrira par exemple :

```
int largest_even;
bool found_even = false;
for (int i=0; i<n; ++i) {
    // found_even contient false si et seulement si
    // tous les éléments d'index <i sont impairs, et sinon
    // largest_even = max_{0⩽k<i, 2|arr[k]} arr[k]
    if (arr[i]%2 == 0 && (!found_even
                          || arr[i] > largest_even)) {
        found_even = true;
        largest_even = arr[i];
    }
}
// si found_even contient true, alors
// largest_even = max_{0⩽k<i,2|arr[k]} arr[k]
```

Il n'est cependant pas rare que l'on cherche à éviter, dans la mesure du possible, d'introduire des variables supplémentaires. On peut le faire ici en initialisant `largest_even` avec une valeur qui ne peut figurer parmi les résultats possibles : un entier impair, par exemple, tel que $-1$. Tant que `largest_even` contient $-1$, c'est que l'on n'a identifié aucun entier pair dans le tableau, et lorsque l'on rencontre un entier pair dans cette situation, on le

place dans `largest_even` même s'il est plus petit que $-1$.

```
int largest_even = -1; // (= aucun entier pair trouvé)
for (int i=0; i<n; ++i) {
    // largest_even contient -1 si et seulement si
    // tous les éléments d'index <i sont impairs, et sinon
    // largest_even = max_{0⩽k<i, 2|arr[k]} arr[k]
    if (arr[i]%2 == 0 && (largest_even == -1
                          || arr[i] > largest_even)) {
        largest_even = arr[i];
    }
}
// si largest_even ≠ -1, alors le tableau contient au moins
// un entier pair et largest_even = max_{0⩽k<i, 2|arr[k]} arr[k]
// si largest_even = -1, il ne contient aucun entier pair
```

| | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| arr | 1 | -3 | -5 | -1 | 3 | 1 | 2 | 4 | -2 | -6 | 5 | 7 | -5 | 5 |
| largest_even : | -1 | -1 | -1 | -1 | -1 | -1 | -1 | 2 | 4 | 4 | 4 | 4 | 4 | 4 | 4 |

itération i=0

## 2.4 Localisation d'un élément particulier

Supposons à présent que l'on recherche l'index du premier élément d'un tableau `arr` de longueur `n` vérifiant une propriété. On peut par exemple chercher un élément pair, un élément ayant une valeur particulière...

Pour rester dans un cadre très général, nous supposerons que l'on peut déterminer si un élément `x` convient en écrivant « `looked_for(x)` » et que le résultat sera `true` s'il s'agit d'un élément tel qu'on en cherche et `false` dans le cas contraire. Si l'on s'intéresse à la présence d'un zéro dans le tableau, il suffira par exemple d'écrire « `x==0` » en lieu et place de « `looked_for(x)` ».

Cette fois encore, on va simplement égrenner les éléments du tableau un à un, et les tester. Si on trouve un élément qui convient, on mémorisera son index dans une variable `index`. Nous reprendrons par ailleurs l'idée d'une valeur particulière, ici -1 (qui ne peut correspondre à un index dans le tableau) pour indentifier l'éventuelle situation où la boucle se termine sans que l'on ait trouvé un élément qui convienne.

Une fois un élément trouvé, on ne souhaite pas poursuivre la recherche dans le tableau, ce serait du travail inutile ! On préfère sortir immédiatement de la boucle. Il serait possible de modifier la condition d'arrêt de la boucle [21], mais il s'agit ici d'un cas où sortir avec `break` convient parfaitement :

```
int index = -1; // (= aucun élément adéquat trouvé)
for (int i=0; i<n; ++i) {
    // Les éléments d'index < i ne vérifient pas la propriété
    if (looked_for(arr[i])) {
        index = i;
        break;
    }
}
// Si index ≠ -1, alors il s'agit de l'index du premier élément
// vérifiant la propriété, sinon aucun ne convient
```

S'il s'agit du *dernier* élément vérifiant la propriété que l'on cherche, il suffirait simplement de retirer le `break`, mais il est plus logique de parcourir le tableau dans l'autre sens, car il est inutile de parcourir systématiquement tout le tableau si l'on peut directement identifier l'élément recherché parmi les derniers :

```
int index = -1; // (= aucun élément adéquat trouvé)
for (int i=n-1; i>-1; --i) {
    // Les éléments d'index > i ne vérifient pas la propriété
    if (looked_for(arr[i])) {
        index = i;
        break;
    }
}
// Si index ≠ -1, alors il s'agit de l'index du dernier élément
// vérifiant la propriété, sinon aucun ne convient
```

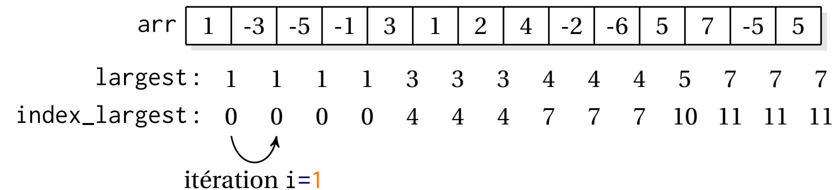
Notons que l'on peut même se passer d'un index de boucle en se servant directement de la variable `index`. Toutefois, si l'on cherche le *premier* élément vérifiant la propriété, comme ci-dessous, un échec à trouver un élément qui convienne conduira à obtenir la valeur `n` (qui ne correspond à aucun index valide du tableau) dans la variable `index` :

```
int index;
for (index=0; index<n; ++i) {
    // Les éléments d'index < index ne vérifient pas la propriété
    if (looked_for(arr[index])) {
        break;
    }
}
// Si index ≠ n, alors il s'agit de l'index du premier élément
// vérifiant la propriété, sinon aucun ne convient
```

21. En écrivant « `for (int i=0; i<n && index==-1; ++i)` » par exemple.

Il arrive fréquemment que l'on ne sache pas définir à l'avance, avant d'avoir parcouru complètement le tableau, si un élément donné convient. Par exemple lorsque l'on recherche l'index du plus grand élément du tableau (que l'on ne connaît pas *a priori*). On pourrait dans un premier temps déterminer le maximum, et ensuite rechercher son index, mais il est plus intéressant de simplement retenir, à tout instant, quel est le plus grand des éléments observés, et l'index de cet élément détenteur du « record » :

```
int largest = arr[0];
int index_largest = 0;
for (int i=1; i<n; ++i) {
    // largest = max_{0<=k<i} arr[k]
    // index_largest = min_{0<=k<i, arr[k]=largest} k
    if (arr[i] > largest) {
        largest = arr[i];
        index_largest = i;
    }
}
// index_largest est l'index du plus grand élément de arr
// (le plus petit index possible s'il y en a plusieurs)
```




Notons que dans ce dernier cas, il est possible [22] de se passer de `largest` en écrivant :

```
int index_largest = 0;
for (int i=1; i<n; ++i) {
    // index_largest = plus petit index d'un élément maximal
    //                 parmi ceux aux positions 0 à i-1 inclus
    if (arr[i] > arr[index_largest]) {
        index_largest = i;
    }
}
// index_largest est l'index du plus grand élément de arr
// (le plus petit index possible s'il y en a plusieurs)
```

22. En contrepartie de l'économie d'une variable et d'affectations, on effectue davantage d'accès aux cases du tableau, cela présente à la fois de légers avantages et des inconvénients. Les deux approches sont tout aussi acceptables.

## 2.5 Vérification de propriétés

Lorsque l'on souhaite vérifier si au moins un élément du tableau vérifie une propriété, c'est encore plus facile. On procède comme si l'on recherchait l'index de cet élément, mais une variable booléenne `at_least_one` indiquant si on a trouvé un élément adéquat suffit ici. On suppose dans un premier temps qu'aucun élément ne convient, et on teste les éléments du tableau. Au premier élément qui convient, on place `true` dans la variable booléenne et on arrête la boucle.

```
at_least_one = false;
for (int i=0; i<n; ++i) {
    // Les éléments d'index < i ne vérifient pas la propriété
    if (looked_for(arr[i])) {
        at_least_one = true;
        break;
    }
}
// at_least_one contient true si et seulement si
// au moins un des éléments du tableau convient
```

Il pourrait sembler un peu plus délicat de vérifier si *tous* les éléments conviennent. Mais un peu de logique suffit à s'apercevoir que cela revient à vérifier qu'aucun élément *ne convient pas*. En termes mathématiques, cela correspond à l'équivalence suivante :

$$\forall x \in \mathrm{E},\ \mathscr{P}(x) \quad \equiv \quad \nexists x \in \mathrm{E} \mid \text{non}\,\mathscr{P}(x)$$

Nous avons donc précisément la *même* structure que dans le cas précédent, avec juste une inversion des booléens ! On fait ainsi l'hypothèse initiale que tous conviennent, et on regarde si l'un des éléments nous contredit (auquel cas, on peut là aussi sortir immédiatement de la boucle) :

```
all_fit = true;
for (int i=0; i<n; ++i) {
    // Les éléments d'index < i vérifient la propriété
    if (!looked_for(arr[i])) {
        all_fit = false;
        break;
    }
}
// all_fit contient true si et seulement si
// tous les éléments du tableau conviennent
```

Il peut arriver que la propriété à vérifier ne concerne pas les éléments pris séparément, mais des relations entre les éléments. Par exemple, on peut souhaiter vérifier que les

éléments d'un tableau sont rangés par ordre croissants. Cela revient à vérifier que

$$\forall i \in [\![1 \mathinner{..} n-1]\!],\ \texttt{tab[i-1]} \leqslant \texttt{tab[i]}$$

La traduction en langage C est très similaire au cas précédent : puisqu'il s'agit de vérifier que c'est vrai *pour tout* $i \in [\![1 \mathinner{..} n-1]\!]$, on part de l'hypothèse que c'est vrai, et on parcourt la liste à la recherche d'un éventuel contre-exemple. Cela donne :

```c
is_increasing_sequence = true;
for (int i=1; i<n; ++i) {
    // Les éléments d'index < i sont rangés par ordre croissant
    if (arr[i-1] > arr[i]) {
        is_increasing_sequence = false;
        break;
    }
}
// Les éléments du tableau sont rangés par ordre croissant
// si et seulement si is_increasing_sequence contient true
```

On prendra garde à l'initialisation de la boucle, qui, en accord avec la propriété que l'on teste, commence avec $i = 1$. Cela permet notamment de garantir que « `arr[i-1]` » fait toujours bien référence à un élément du tableau. On aurait pu également effectuer une boucle sur les indices de `0` à `n-2` inclus, en comparant alors `arr[i]` et `arr[i+1]`.

## 2.6 Plus longue séquence

Une dernière situation qui apparait fréquemment dans les algorithmes demande de trouver la plus longue séquence possible d'éléments vérifiant une propriété. Par exemple, trouver la longueur de plus longue séquence d'éléments pairs dans un tableau.

Un tel problème peut être résolu en itérant une seule fois sur les éléments du tableau, il n'est pas besoin de deux boucles. L'idée est simple : on utilise un compteur `curr_count`, initialisé à 0, que l'on incrémente à chaque fois que l'on rencontre un nombre qui convient (pair) et que l'on remet à zéro lorsque l'on rencontre un nombre qui ne convient pas (impair). Une seconde variable `max_count` mémorise à tout instant la plus grande valeur atteinte par `curr_count`. L'algorithme se déroule donc de la façon suivante :

| | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| arr | 1 | 2 | 4 | -1 | -2 | 1 | 2 | 4 | -8 | -6 | 5 | 6 | -4 | 0 |
| curr_count : | 0 | 0 | 1 | 2 | 0 | 1 | 0 | 1 | 2 | 3 | 4 | 0 | 1 | 2 | 3 |
| max_count : | 0 | 0 | 1 | 2 | 2 | 2 | 2 | 2 | 2 | 3 | 4 | 4 | 4 | 4 | 4 |

itération i=0

L'implémentation en langage C ne pose pas de difficulté particulière :

```c
int curr_count = 0;
int max_count = 0;
for (int i=0; i<n; ++i) {
    // max_count contient la longueur de la plus longue séquence
    // observée d'entiers pairs, curr_count de la séquence en cours
    if (arr[i]%2 != 0) {
        curr_count = 0;
    } else {
        curr_count++;
        if (curr_count > max_count) {
            max_count = curr_count;
        }
    }
}
// max_count contient la longueur d'une plus longue
// séquence d'entiers pairs
```

Il peut sembler inutile de mettre à jour systématiquement `max_count` à chaque fois que l'on augmente `curr_count`. On pourrait ne le faire que lorsqu'une séquence se termine (typiquement lorsque l'on rencontre un entier impair dans le cas présent). C'est parfaitement possible, mais avec un petit problème : il faut *aussi* pouvoir prendre en compte la dernière séquence si elle s'achève par la fin du tableau et non sur un entier impair.

Cela nécessite donc d'ajouter un morceau de code après la boucle. On se retrouve avec deux morceaux de code identiques à deux endroits distincts, ce qui n'est fréquemment pas une bonne idée (lorsque l'on modifie le code, on a tôt fait de modifier un morceau et d'oublier l'autre). Une alternative peut consister à ajouter une valeur supplémentaire en fin de tableau pour garantir que la dernière séquence ne peut pas se terminer par la fin du tableau. Dans le cas présent, cela consisterait simplement à ajouter une valeur impaire quelconque dans une case supplémentaire à la fin du tableau.

Il peut sembler délicat de déterminer où la séquence la plus longue commence, car lorsqu'une séquence commence, on ne sait pas encore si ce sera effectivement la plus longue. Une solution consiste à se mémoriser quand la séquence la plus longue *se termine* !

| | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| arr | 1 | 2 | 4 | -1 | -2 | 1 | 2 | 4 | -8 | -6 | 5 | 6 | -4 | 0 |
| curr_count : | 0 | 0 | 1 | 2 | 0 | 1 | 0 | 1 | 2 | 3 | 4 | 0 | 1 | 2 | 3 |
| max_count : | 0 | 0 | 1 | 2 | 2 | 2 | 2 | 2 | 2 | 3 | 4 | 4 | 4 | 4 | 4 |
| index_end_max : | ? | ? | 1 | 2 | 2 | 2 | 2 | 2 | 2 | 8 | 9 | 9 | 9 | 9 | 9 |

itération i=0

Cela donne donc :

```
int curr_count = 0;
int max_count = 0;
int index_end_max;
for (int i=0; i<n; ++i) {
    // max_count contient la longueur de la plus longue séquence
    // observée d'entiers pairs, nb_courant de la séquence en cours
    if (arr[i]%2 != 0) {
        curr_count = 0;
    } else {
        curr_count++;
        if (curr_count > max_count) {
            max_count = curr_count;
            index_end_max = i;
        }
    }
}
// max_count contient la longueur d'une plus longue séquence
// d'entiers pairs, et index_end_max contient l'index de la
// dernière case d'une telle séquence *s'il en existe*
```

Si l'on souhaite savoir à quel index de tableau la séquence a débuté, il suffit de calculer « `index_end_max - max_count + 1` » (attention de ne pas oublier le `+1` !) Dans notre exemple, la séquence la plus longue, de longueur 4, s'étend donc sur les cases d'index 6 à 9.

Dans le problème précédent, lorsqu'une séquence se termine, la séquence suivante ne débute pas de suite (il y a au moins un entier impair entre deux séquences d'entiers pairs consécutifs). Ce n'est hélas pas toujours aussi facile, par exemple si l'on souhaite déterminer la longueur de la plus longue séquence croissante par exemple. Dans cette situation, les séquences croissantes se suivent immédiatement.

Aussi, lorsque l'on détecte deux éléments consécutifs dans le tableau tels que le second est strictement plus petit que le premier, c'est le signe qu'une séquence croissante s'est terminée, mais le second élément est déjà le début d'une nouvelle séquence. On repart ainsi tout de suite avec une séquence de longueur 1.

| arr | 1 | 2 | 4 | -1 | -2 | 1 | 2 | 4 | -8 | -6 | 5 | 6 | -4 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| curr_count : | 1 | 2 | 3 | 1 | 1 | 2 | 3 | 4 | 1 | 2 | 3 | 4 | 1 | 2 |
| max_count : | 1 | 2 | 3 | 3 | 3 | 3 | 3 | 4 | 4 | 4 | 4 | 4 | 4 | 4 |
| index_end_max : | 0 | 1 | 2 | 2 | 2 | 2 | 2 | 7 | 7 | 7 | 7 | 7 | 7 | 7 |

itération `i=1`

On remarquera que l'itération commence à 1. Ce n'est pas surprenant, car on s'appuie sur le même principe que le programme vérifiant si les éléments d'un tableau sont tous rangés par ordre croissant en les comparant deux à deux.

L'initialisation également est quelque peu différente. En cas de doute, il faut en revenir à notre invariant de boucle : l'initialisation doit permettre que l'invariant soit correct au début de la première itération. Avant de traiter le second élément, on a bien une séquence croissante de 1 entier (`arr[0]`), séquence qui se termine pour l'instant à l'index 0.

L'implémentation peut donc ressembler à ceci :

```
int curr_count = 1;
int max_count = 1;
int index_end_max = 0;
for (int i=1; i<n; ++i) {
    // max_count contient la longueur de la plus longue séquence
    // observée d'entiers croissants, index_fin_max l'index
    // de son dernier élément, et nb_courant la longueur
    // de la séquence croissante en cours
    if (arr[i-1] > arr[i]) {
        curr_count = 1; // arr[i] débute une nouvelle séquence
    } else {
        curr_count++;
        if (curr_count > max_count) {
            max_count = curr_count;
            index_end_max = i;
        }
    }
}
// max_count contient la longueur d'une plus longue
// séquence d'entiers croissants, index_end_max contient
// l'index de la dernière case d'une de ces séquences
```

Terminons-en en remarquant que l'on aurait pu écrire le cœur de la boucle différemment, en effectuant systématiquement l'incrémentation de `curr_count` :

```
    if (arr[i-1] > arr[i]) {
        curr_count = 0; // une nouvelle séquence va débuter
    }
    curr_count++; // arr[i] est un élément de la séquence en cours
    if (curr_count > max_count) {
        max_count = curr_count;
        index_end_max = i;
    }
```

# 3 Quelques exemples plus élaborés

## 3.1 Recherche d'un gagnant d'un vote à la majorité

**Objectif poursuivi**

Penchons-nous à présent sur un problème moins trivial, la question du dépouillement d'un vote. $n$ votants ont exprimé leur choix parmi différentes possibilités, que l'on suppose identifiées par des entiers positifs. Les votes de chacun des votants sont regroupés dans un tableau de $n$ entiers appelé `vote`, de sorte que `vote[i]` soit un entier positif correspondant au choix du votant numéro `i` (avec $0 \leqslant \texttt{i} < n$). Par exemple, un vote avec 16 participants peut être représenté par le tableau suivant :

| vote | 42 | 37 | 29 | 37 | 11 | 4 | 37 | 9 | 37 | 37 | 37 | 42 | 37 | 37 | 1 | 37 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|

On cherche à déterminer si un des choix a obtenu une majorité absolue, c'est-à-dire s'il existe un entier positif $k$ tel que le nombre de cases du tableau contenant la valeur $k$ soit strictement supérieur à $n/2$. Dans cette situation, le choix $k$ est déclaré *gagnant* du vote.

Par exemple, pour le vote précédemment cité en exemple, le choix 17 ressort gagnant car il a obtenu strictement plus de la moitié des votes (9 votes sur 16) :

| vote | 42 | 37 | 29 | 37 | 11 | 4 | 37 | 9 | 37 | 37 | 37 | 42 | 37 | 37 | 1 | 37 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|

En revanche, dans le cas du second vote ci-dessous, il n'y a aucun gagnant (car il est nécessaire de dépasser *strictement* $n/2$) :

| vote | 42 | 37 | 29 | 37 | 11 | 4 | 37 | 9 | 37 | 37 | 37 | 42 | 37 | 54 | 1 | 37 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|

**Solution « naive »**

Une première solution peut consister à compter les votes pour chacun des choix, et voir si l'un de ces décomptes est strictement supérieur à $n/2$. Comme le nombre de choix possibles peut être très grand, on voudra sans doute se restreindre aux choix qui ont été exprimés, c'est-à-dire apparaissant parmi les votes.

Faute de connaître les choix exprimés, on pourra effectuer une itération sur chacun des votes pour les connaître, avant de les dénombrer.

On supposera dans la suite que le tableau contenant le résultat du vote déclaré sous la forme ci-dessous, et qu'un entier `n` contient le nombre de votants :

```c
int vote[] = { 42, 37, 29, 37, 11, 4, 37, 9,
                              37, 37, 37, 42, 37, 37, 1, 37 };
int n = 16;
```

Un programme déterminant l'éventuel vainqueur du vote est proposé ci-dessous. On y introduit une variable entière `winner`, qui contiendra en fin d'algorithme le choix ayant reçu strictement plus de $n/2$ votes, ou $-1$ (qui ne peut correspondre à un choix, lesquels sont positifs) si aucun choix n'a atteint cette condition.

```c
int winner = -1; // Contiendra le choix gagnant, s'il existe

for (int i=0, i<n; ++i) {
    // Les choix des votants < i ont été examinés,
    // aucun n'a remporté strictement plus de n/2 suffrages
    // On dénombre les votants d'index j ⩾ i
    // dont le choix est identique à celui du votant i
    int count = 1;
    for (int j=i+1, j<n; ++j) {
        if (vote[j] == vote[i]) {
            count++;
        }
    }
    // On regarde si ce choix a remporté la majorité absolue
    // des suffrages
    if (count > n/2) {
        winner = vote[i];
        break;
    }
}
// Si winner ≠ -1, alors il contient un choix
// qui a été exprimé strictement plus de n/2 fois
```

Il devient inutile de continuer à décompter les votes dès que l'on a identifié un gagnant (il ne peut, de façon évidente, y en avoir plusieurs). Il est donc tout à fait raisonnable d'interrompre la boucle dès que le gagnant est identifié, c'est la raison pour laquelle on quitte la boucle avec un **`break`**.

Si un choix a fait l'objet de plusieurs votes, on risque fort de décompter plusieurs fois le nombre de votes pour un même choix[23]. On peut s'affranchir de ce problème en « marquant » les votes comptabilisés.

On pourrait utiliser un tableau booléen supplémentaire pour ce faire, mais il est souvent possible d'utiliser une transformation (si possible réversible) des valeurs dans le tableau étudié pour identifier celles déjà traitées. Dans le cas présent, les choix correspondant à des entiers positifs, on peut les convertir en entiers strictement négatifs, et ignorer les

23. S'ils n'ont pas atteint la majorité des suffrages avec la sortie anticipée. Notons que les décomptes ultérieurs pour ce choix ne donneront pas le total correct des votes puisque l'on débute le décompte au votant `i`, mais cela n'a pas de conséquences sur le résultat car on sait déjà que le nombre de votes ne dépassera pas $n/2$.

valeurs strictement négatives dans le tableau, par exemple avec un **continue**, car ayant déjà été traités. Cela donne par exemple :

```
int winner = -1;

for (int i=0, i<n; ++i) {
    // les choix des votants < i ont été examinés
    if (vote[i] < 0) { // déjà comptabilisé
        continue;
    }
    int count = 1;
    for (int j=i+1, j<n; ++j) {
        if (vote[j] == vote[i]) {
            vote[j] = -1-vote[j]; // on le marque
            count++;
        }
    }
    if (count > n/2) {
        winner = vote[i];
        break;
    }
}
```

Ce marquage a évidemment pour conséquence de modifier les valeurs contenues dans le tableau. Mais ce marquage est aisément réversible une fois la recherche du gagnant terminée, en ajoutant :

```
for (int i=0; i<n; ++i) {
    if (vote[i] < 0) {
        vote[i] = -(vote[i]+1);
    }
}
```

Cela étant, quelle que soit la variante que l'on choisisse, le nombre de comparaisons « `vote[j] == vote[i]` » peut, dans le pire des cas (par exemple lorsque tous les votes expriment un choix différent), atteindre $n(n-1)/2$. Pour des tableaux de grande taille, ce nombre d'opérations peut devenir prohibitif. Imaginez par exemple un vote à l'échelle d'un pays !

### Algorithme des votes de Boyer-Moore

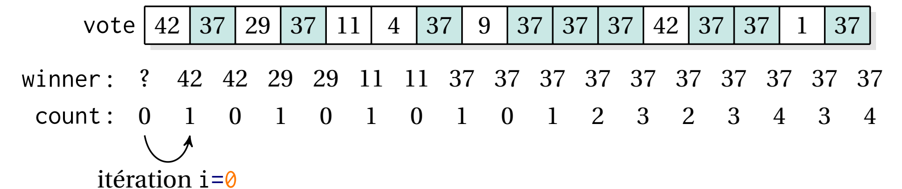
Il est en fait, avec un peu de réflexion, possible de faire bien mieux. Si un choix a fait l'objet de plus de la moitié des suffrages, il existe moins de votes en faveur de l'ensemble des autres choix, quels qu'ils soient, qu'en faveur du choix gagnant. En se basant sur cette constatation, Robert S. Boyer et J Strother Moore ont proposé un algorithme permettant de déterminer le gagnant du vote s'il existe de la façon suivante. On crée une variable entière `count`, initialisée à 0, et une variable `winner` destinée à recevoir un gagnant potentiel. Puis on itère sur les éléments du tableau, en effectuant deux opérations :

- si `count=0`, on considère le choix courant comme gagnant possible en le plaçant dans `winner`
- si le choix courant correspond au contenu de la variable `winner`, on incrémente `count`, et sinon on le décrémente.

L'algorithme s'écrit donc [24] :

```
int count = 0;
int winner;
for (int i=0; i<n; ++i) {
    if (count == 0) {
        winner = vote[i];
    }
    if (vote[i] == winner) {
        count++;
    } else {
        count--;
    }
}
```

L'application de cet algorithme à notre premier exemple donne ainsi :




**Lemme 1.** *À l'issue de la boucle, s'il y a un choix gagnant, alors il correspond à ce qui est mémorisé dans la variable* `winner`.

**Démonstration.** Pour le prouver, supposons que le vote ait un gagnant $g$, et considérerons la quantité $c$ qui vaut, à l'issue de chaque itération,

- `count` lorsque `winner`$= g$ ;
- `-count` dans les autres cas.

Si, lors d'une itération, `vote[i]`$= g$, alors $c$ est incrémentée de 1. Lorsque `vote[i]`$\neq g$, $c$ peut être soit incrémentée, soit décrémentée de 1. Si $g$ est le choix gagnant du vote, alors $c$ est nécessairement incrémentée strictement plus souvent qu'elle n'est décrémentée,

24. On pourrait aussi initialiser `count` à 1, `winner` à `vote[0]` et commencer l'itération à $i = 1$.

donc $c$ est strictement positive à la fin de l'algorithme. Et comme `count` ne peut être que positif, cela signifie que l'on a forcément `winner`$= g$. □

Cependant, `winner` ne contient pas nécessairement un choix qui a remporté la majorité des suffrages [25]. Il convient donc ensuite de compter explicitement le nombre de votes pour ce choix avant de pouvoir conclure :

```
count = 0;
for (int i=0; i<n; ++i) {
    if (vote[i] == winner) {
        count++;
    }
}
if (count <= n/2) {
    winner = -1; // aucun choix n'a obtenu plus de n/2 votes
}
```

Dans la version présentée, l'algorithme n'est pas forcément capable de trouver un choix exprimé $n/2$ fois exactement si $n$ est pair [26]. Cela peut par exemple arriver si tous les choix d'index impair se sont portés sur un même choix, et les autres se sont portés sur divers autres choix. Toutefois, on peut montrer que si un choix a été fait $n/2$ fois (avec $n$ pair), s'il n'est pas dans `winner`, alors nécessairement `vote[n-1]` contient ce choix [27] [28].

## 3.2 Problème du sous-tableau maximal

### Objectif poursuivi

On considère un tableau `arr` contenant `n` valeurs numériques. On cherche à déterminer, parmis tous les « sous-tableaux » de `arr` (des morceaux connexes du tableau `arr`) celui dont la somme des éléments est maximal.

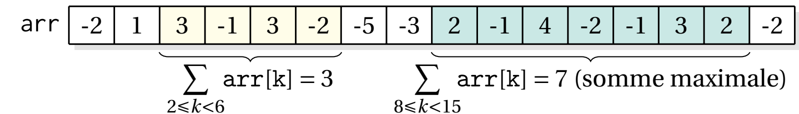
En d'autres termes, on cherche à maximiser l'expression suivante :

$$\max_{0\leqslant i\leqslant j\leqslant n}\left(\sum_{i\leqslant k<j}\mathtt{arr[k]}\right)$$

et, incidemment, les valeurs de `i` et `j` permettant d'obtenir cette somme maximale.

Dans la formulation du problème que l'on a choisi ici, on remarque que l'on considère les situations où `i=j`, c'est-à-dire où la somme ne contient aucun élément (on considérera alors que la somme est nulle). La somme maximale est donc nécessairement positive ou nulle [29].

Par exemple, pour le tableau représenté ci-dessous, la somme maximale que l'on puisse obtenir est 7, atteinte lorsque $i = 8$ et $j = 15$ :




### Solution très naïve

Une première solution, appliquant à la lettre la définition proposée, considère tous les couples $(i, j)$ possibles, et calcule séparément chacune des sommes correspondantes pour en déterminer la plus grande [30]. Cela peut s'écrire par exemple :

```
int largest_sum = 0
for (int i=0; i<n; ++i) {
    for (int j=i+1; j<=n; ++j) {
        // Calcul de ∑_{i≤k<j} arr[k]
        int sum = 0;
        for (int k=i; k<j; ++k) {
            sum += arr[k];
        }
        if (sum > largest_sum) {
            largest_sum = sum;
        }
    }
}
```

Rien de bien complexe ici, si ce n'est une attention particulière à apporter aux indices et à l'initialisation. Les cas $i = j$ étant triviaux (ils donnent une somme nulle), on a initialisé `largest_sum` à 0 et on ne traite donc que les cas $0 \leqslant$ `i<j` $\leqslant n$.

La boucle sur `j` commence donc à `j=i+1` et se termine avec `j=n` *inclus*, comme l'indique la formule. Cela ne provoquera pas d'accès hors du tableau avec « `arr[k]` » car la boucle sur `k` se termine avec `k=j-1`. Enfin, la boucle sur `i` se termine avec `i=n-1` puisque l'on a supposé `i<j`.

Cette solution est parfaitement correcte, mais présente un inconvénient majeur : elle effectue une très grande quantité d'opérations. En effet, la boucle sur `k` effectue `j-i`

25. Ni même le choix le plus souvent exprimé.
26. Ce qui n'a rien de surprenant, car il n'est pas forcément unique.
27. Pour le montrer, on peut considérer le tableau privé de son dernier élément et les résultats précédents.
28. Il existe également des variantes un peu plus élaborées de cet algorithme pour trouver par exemple un choix exprimé plus de $n/3$ fois, en utilisant deux compteurs et deux variables mémorisant des choix.

29. Cette contrainte n'est pas très limitante : si au moins une valeur du tableau est positive ou nulle, la somme maximale sera positive et on pourra trouver un sous-tableau non-vide de somme maximale. Si toutes les valeurs sont strictement négatives, la somme maximale d'un sous-tableau non-vide correspond exactement à la plus grande des valeurs du tableau.
30. On ne se préoccupe pas ici, afin de garder l'algorithme le plus lisible possible, de mémoriser les `i` et `j` donnant la somme maximale, mais cela ne pose pas de difficulté particulière.

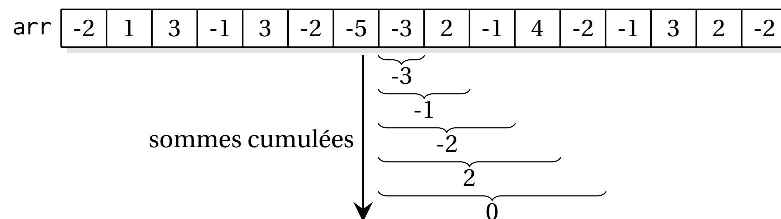
additions. Il y a donc au total

$$\sum_{i=0}^{n-1}\sum_{j=i+1}^{n} j-i = \sum_{i=0}^{n-1}\frac{(n-i)(n-i+1)}{2} = \sum_{i=0}^{n-1}\frac{i^2-(2n+1)i+n^2+n}{2} = \frac{n^3+3n^2+2n}{6}$$

additions effectuées. Un nombre qui croît très vite lorsque la taille `n` du tableau augmente[31].

### Un peu plus intelligent

On remarquera toutefois que, si l'on calcule $\sum_{i\leqslant k<j}$ arr[k], il n'est pas pertinent de repartir de zéro pour calculer $\sum_{i\leqslant k<j+1}$ arr[k]. En fait, toutes les sommes $\sum_{i\leqslant k<j}$ arr[k] pour un $i$ donné et tous les $i \leqslant j \leqslant n$ peuvent être calculées en utilisant la méthode des sommes cumulées présentée tantôt, en commençant à la case d'index $i$ :




Avec cette constatation, le programme se simplifie quelque peu[32] :

```c
int largest_sum = 0
for (int i=0; i<n; ++i) {
    int sum = 0;
    for (int j=i; j<n; ++j) {
        sum += arr[j];
        // sum contient ∑_{i≤k<j+1} arr[k]
        if (sum > largest_sum) {
            largest_sum = sum;
        }
    }
}
```

31. Dans la pratique, nous n'aurons pas besoin du nombre exact d'additions effectuées, mais seulement de la vitesse à laquelle ce nombre évolue lorsque $n$ augmente. Nous n'effectuerons donc généralement pas de calcul aussi précis. Dans le cas présent, il est aisé de voir que le nombre d'additions est majoré par $n^3$, et, en considérant dès que $n \geqslant 5$ les seuls sous-tableaux pour lesquels $0 \leqslant i \leqslant \lceil n/5 \rceil$ et $\lfloor 4n/5 \rfloor \leqslant j \leqslant n$, tableaux tous de taille supérieure à $\lceil n/2 \rceil$, de montrer ainsi que le nombre d'opérations est supérieur à $n^3/50$. L'encadrement est grossier, mais suffisant pour comprendre que l'augmentation de $n$ rendra très rapidement cet algorithme inutilisable.

32. La variable `j` qui y figure n'y désigne pas exactement le `j` de la formule initiale, comme on peut le voir sur l'invariant de boucle avec la présence du $+1$, mais ce décalage de 1 permet d'éviter d'écrire `arr[j-1]`.

Mais surtout, on n'effectue plus que $n(n+1)/2$ additions pour déterminer le résultat recherché, ce qui est déjà, pour de grandes valeurs de $n$, un progrès considérable. Pour un tableau à un millier d'éléments, on est ainsi passé d'un nombre d'additions de l'ordre du milliard au million. Et plus $n$ est grand, plus l'économie est importante.

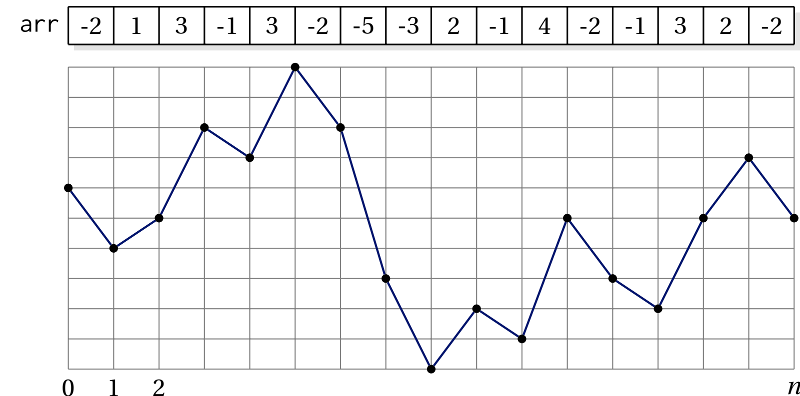
Mais on peut encore faire mieux !

### Algorithme de Kadane

L'algorithme de Kadane visant à déterminer le sous-tableau de somme maximal a parfois la réputation d'être complexe et obscur, mais il n'en est rien si on parvient à donner aux grandeurs calculées une image aussi claire que possible.

Aussi tentons de décrire le problème en des termes plus concrets. Nous pouvons par exemple prendre l'image d'une action boursière dont le prix évolue chaque jour. Chaque case du tableau indique ainsi l'évolution du prix de l'action[33]. Dans notre tableau d'exemple, le prix de celle-ci a baissé de 2 unités le premier jour, puis monté de 1 unité lors du second jour, de 3 unités lors du troisième jour et ainsi de suite.

Le tableau des sommes cumulées correspond donc à l'évolution du cours de l'action au cours du temps. Cela donne par exemple :




Avec cette interprétation, le terme $\sum_{i\leqslant k<j}$ arr[k] prend une signification simple : il s'agit de l'argent que l'on a gagné (ou perdu) en achetant l'action au matin du jour $i$ (avant l'évolution correspondant à `arr[i]`) et en la revendant au matin du jour $j$.

La question de la plus grande somme d'un sous-tableau devient donc le problème du gain maximal que l'on peut faire en achetant, puis en revendant notre action. Puisque le problème impose $i \leqslant j$, on peut vendre l'action immédiatement après l'avoir achetée ($i = j$), le gain étant alors nul, mais il n'est pas permis de la vendre *avant* de l'avoir achetée.

33. Attention, on parle d'évolution en terme de différence de prix, il ne s'agit pas ici de pourcentages de hausse ou de baisse.

On ne peut donc pas simplement résoudre le problème en calculant la différence entre le cours le plus haut et le cours le plus bas.

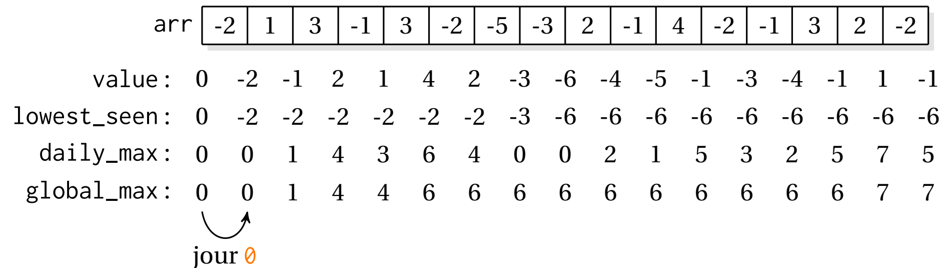
Supposons que l'on vende notre action le matin du jour $j$. Pour faire le plus grand bénéfice possible, il faut l'avoir achetée lorsque le cours était le plus bas possible, parmi tous les jours $i \leqslant j$.

Ainsi, si on la vend le matin du jour $j = 5$ (lorsque le cours était au plus haut), le mieux était de l'avoir achetée le matin du jour $i = 1$ (bénéfice de 6). Si on la vend le matin du jour $j = 11$, il fallait l'acheter le matin du jour 8 pour faire le plus grand bénéfice possible (soit un bénéfice de 5).

Pour résoudre le problème, nous allons donc considérer tour à tour tous les jours $j$ possibles pour la vente (y compris le matin du jour $n$, après la dernière variation enregistrée dans le tableau), et voir quel bénéfice on peut en tirer. Pour ce faire, nous aurons besoin de connaître le cours de la valeur le matin du jour $j$, mais aussi le plus bas cours qu'ait connu l'action entre le premier jour et le jour considéré.

Notons que le cours initial n'a pas d'importance, puisque l'on ne raisonne qu'en terme de différences. On suppose donc que le matin du jour 0, le cours était égal à 0 [34]

Appliqué au tableau pris en exemple, cela donne :



| arr | -2 | 1 | 3 | -1 | 3 | -2 | -5 | -3 | 2 | -1 | 4 | -2 | -1 | 3 | 2 | -2 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|

| | | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| value : | 0 | -2 | -1 | 2 | 1 | 4 | 2 | -3 | -6 | -4 | -5 | -1 | -3 | -4 | -1 | 1 | -1 |
| lowest_seen : | 0 | -2 | -2 | -2 | -2 | -2 | -2 | -3 | -6 | -6 | -6 | -6 | -6 | -6 | -6 | -6 | -6 |
| daily_max : | 0 | 0 | 1 | 4 | 3 | 6 | 4 | 0 | 0 | 2 | 1 | 5 | 3 | 2 | 5 | 7 | 5 |
| global_max : | 0 | 0 | 1 | 4 | 4 | 6 | 6 | 6 | 6 | 6 | 6 | 6 | 6 | 6 | 6 | 7 | 7 |



Ainsi, à tout instant de l'algorithme, avant de traiter la case j du tableau ;

- la variable value contient le cours de la valeur le matin du jour j, son calcul correspond aux sommes cumulées des valeurs du tableau ;
- la variable lowest_seen contient le plus bas cours observé un matin d'un jour $i \leqslant j$, elle est mise à jour chaque fois que le cours descend en-dessous de la valeurs plus-bas ;
- la variable daily_max correspond au gain maximal value-lowest_seen que l'on peut obtenir si l'on vend la valeur boursière le matin du jour considéré ;
- enfin, la variable global_max contient le plus grand gain que l'on ait pu obtenir pour une vente ayant eu lieu le matin d'un jour $j' \leqslant j$, elle est mise à jour chaque fois que daily_max dépasse global_max.

A la fin de l'algorithme, global_max contient le plus grand gain que l'on ait pu obtenir, quel que soit le matin où l'on ait décidé de vendre la valeur (y compris le matin du jour n). Ce résultat correspond exactement au problème de la sous-somme maximale que l'on pouvait obtenir.

34. Cela conduira à manipuler des cours négatifs, mais cela n'a pas d'importance.

Or cette approche est très efficace : pour chaque jour, on effectue une somme (pour mettre à jour value), une soustraction (pour calculer daily_max) et deux comparaisons (pour éventuellement mettre à jour lowest_seen et global_max), soit $4n$ opérations au total pour traiter l'intégralité du problème ! Ce qui est un progrès conséquent par rapport aux deux précédentes approches... Traduit en langage C, cela donne :

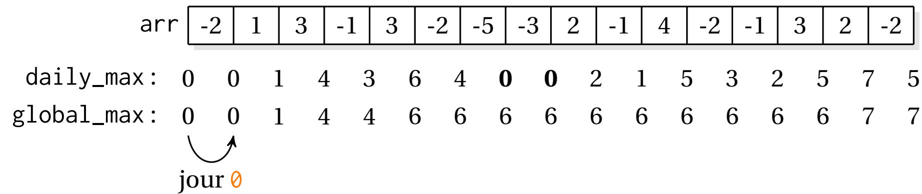
```c
int value = 0;
int lowest_seen = 0;
int global_max = 0; // = plus grande somme

for (int j=0; j<n; ++j) {
    // value contient le cours de l'action le matin du jour j
    // lowest_seen le plus bas cours observé un jour i <= j
    // global_max contient le plus grand bénéfice que l'on ait
    // pu faire en vendant le matin d'un jour j' <= j

    // En préparation du jour suivant, on met à jour le cours
    value += arr[j];
    // On regarde si c'est le nouveau "plus bas cours"
    if (value < lowest_seen) {
        lowest_seen = value;
    }
    // On détermine ce que l'on peut gagner en vendant
    int daily_max = value - lowest_seen;
    // On regarde si c'est un nouveau record
    if (daily_max > global_max) {
        global_max = daily_max;
    }
}
```

Poussons l'analyse un peu plus loin, maintenant que l'on a posé les idées générales. Lorsque l'on traite la valeur arr[i] du tableau, on l'ajoute à value. Deux cas peuvent se produire :

- si la nouvelle valeur de value est plus grande que lowest_seen, alors la valeur de daily_max est également modifiée en ajoutant arr[i] ;
- si la nouvelle valeur de value est plus petite que lowest_seen, alors daily_max est remis à 0.

En outre, on peut remarquer que l'on peut identifier le second cas simplement en comparant daily_max à arr[i] : c'est lorsque daily_max + arr[i] est négatif que l'on rencontre cette situation.

Par conséquent, les variables `value` et `lowest_seen` ne sont pas indispensables. On peut directement travailler avec `daily_max` (et `global_max`). L'évolution, sur notre exemple, est représentée ci-dessous. Les deux valeurs en gras représentent les situations où l'ajout de `arr[i]` à `daily_max` aurait donné un résultat négatif, donc la seconde situation évoquée ci-dessus, qui conduit à une remise à zéro de `daily_max`.



| arr | -2 | 1 | 3 | -1 | 3 | -2 | -5 | -3 | 2 | -1 | 4 | -2 | -1 | 3 | 2 | -2 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| daily_max : | 0 | 0 | 1 | 4 | 3 | 6 | 4 | **0** | **0** | 2 | 1 | 5 | 3 | 2 | 5 | 7 | 5 |
| global_max : | 0 | 0 | 1 | 4 | 4 | 6 | 6 | 6 | 6 | 6 | 6 | 6 | 6 | 6 | 6 | 7 | 7 |



Le programme peut donc encore se simplifier :

```
int daily_max = 0, global_max = 0;
for (int j=0; j<n; ++j) {
    daily_max += arr[j];
    if (daily_max < 0) {
        daily_max = 0;
    } else if (global_max < daily_max) {
        global_max = daily_max;
    }
}
```

Si l'on oublie la nature « financière » utilisée pour les noms de variable (et qui n'avaient qu'un but illustratif), et si l'on dispose d'une fonction `max`, alors l'algorithme plaçant dans la variable `best` la somme maximale d'un sous-tableau prend la forme très simple (mais bien plus obscure en terme de compréhension) suivante :

```
int curr = 0, best = 0;
for (int i=0; i<n; ++i) {
    curr = max(0, curr+arr[j]);
    best = max(best, curr);
}
```

Quant à déterminer les $i$ et $j$ qui correspondent à la somme maximale, il faut pouvoir mémoriser à tout instant les $i$ et $j$ ayant conduit à la valeur mémorisée dans la variable `global_max`. Si c'est à peu près immédiat pour `j` (il s'agit de $j+1$ au moment où `global_max` est mis à jour, puisque l'on se place après la prise en compte de la variation `arr[j]`), c'est un peu plus délicat pour `i`. Mais cela nécessite seulement de connaître le jour correspondant à la mise à jour de `lowest` (ou, de façon équivalente, la remise à zéro de `daily_max`). Avec trois variables supplémentaires, on pourrait donc également obtenir les $i$ et $j$ qui correspondent à la somme maximale déterminée par l'algorithme de Kadane.

# 4 Premiers tris et notion de complexité

## 4.1 Le tri par sélection

Le problème du tri des éléments d'un tableau a été très largement étudié en algorithmique. Une des raisons est l'importance d'un tel traitement considérant son omniprésence dans quantité d'algorithmes : beaucoup supposent en effet que l'on traite les éléments du plus grand au plus petit, du plus petit au plus grand, du plus simple au plus complexe, du plus facile au plus problématique, etc.

Une autre raison tient au nombre considérable de solutions différentes qui ont été apportées à ce problème, solutions qui mettent en lumière beaucoup d'idées et de problèmes courants en algorithmique. Difficile donc d'échapper à ce problème lorsque l'on s'intéresse à l'algorithmique...

Nous allons donc à présent nous intéresser à des tableaux contenant des valeurs entières, et au problème du tri de ces entiers par ordre croissant pour l'ordre usuel[35]. On cherchera à effectuer ce tri *en place*, c'est-à-dire en déplaçant les entiers à l'intérieur du tableau, par opposition à la construction d'un nouveau tableau où on placerait les entiers convenablement ordonnés.

La première méthode de tri que nous allons aborder est qualifiée de « tri par sélection ». Le comportement d'un algorithme informatique peut souvent être décrit en des termes semblables à ceux d'un conte. Cela commence par « il était une fois », les données sur lesquelles notre algorithme va travailler, c'est-à-dire ici le tableau non trié :

| arr | 3 | -4 | 1 | -9 | 5 | 2 | -8 | 9 | 6 | -3 | 4 | -6 | 0 | -7 | -1 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|

Et cela se termine par « ils vécurent heureux », la situation à laquelle on souhaite arriver, à savoir un tableau trié par valeurs croissantes :

| arr | -9 | -8 | -7 | -6 | -4 | -3 | -1 | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 9 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|

La meilleure façon de décrire un algorithme est de considérer un moment en plein milieu de l'histoire, en un chapitre `i`, de décrire la situation au début de ce chapitre (ce qui, ce faisant, nous fournira sans doute un invariant de boucle !) et de décrire ce qu'il se passe pour parvenir au chapitre `i+1`.

Dans le cas de l'algorithme du tri par sélection, on supposera qu'au chapitre `i`, les `i` premiers éléments du tableau se trouvent déjà placés au bon endroit[36]. Il s'agit de notre invariant de boucle. Par exemple, pour $i = 5$, la situation ressemblera à :

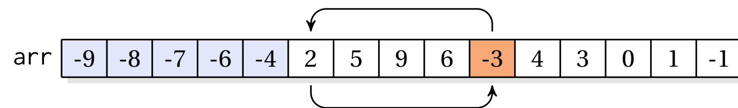
| arr | -9 | -8 | -7 | -6 | -4 | 2 | 5 | 9 | 6 | -3 | 4 | 3 | 0 | 1 | -1 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|

35. Les solutions apportées pourront aisément être ajustées pour toute autre relation d'ordre sur les éléments du tableau, quels que soient les types des éléments.

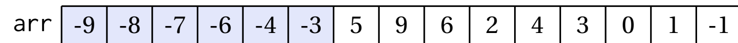
36. Cela ne présume pas du fait que les autres éléments sont à la *mauvaise* place. On ne peut juste pas encore l'affirmer car on ne s'en est pas encore assuré.

Pour avancer dans l'histoire, et passer de « les `i` premiers éléments du tableau sont bien placés » à « les `i+1` premiers éléments du tableau sont bien placés », il faut s'arranger pour que le `i+1`ᵉ élément du tableau, dans la case d'index `i`, soit lui aussi le bon. Comme les éléments à sa gauche sont déjà bien placés, l'élément devant à terme se trouver dans la case d'index `i` est le plus petit des éléments du tableau se trouvant présentement dans les cases d'index `i` à `n-1` (inclus).

Ce plus petit élément identifié, nous n'aurons qu'à échanger sa place avec celui actuellement dans la case d'index `i` :




Une fois la permutation des deux éléments effectués, on se trouve dans la situation suivante, qui correspond bien à celle souhaitée où un élément de plus est bien placé :




Il ne reste qu'à préciser exactement où les choses commencent et où elles se terminent. Initialement, aucun élément n'est identifié comme bien placé, donc on débutera avec `i=0`. C'est un peu plus subtil pour la dernière étape : lorsque l'on aura placé correctement `n-1` éléments, alors l'élément restant est nécessairement aussi bien placé! Avant la toute dernière étape, nous avons donc besoin que `i=n-2` éléments soient bien placés.

Nous avons à présent tous les éléments pour écrire notre tri par sélection :

```
for (int i=0; i<n-1; ++i) {
    // Les i premiers éléments de tab sont bien placés
    // On cherche la position du plus petit élément de arr[i..n-1]
    int index_min = i;
    for (int j=i+1; j<n ++j) {
        if (arr[j] < arr[index_min]) {
            index_min = j;
        }
    }
    // On replace par échange ce plus petit élément en position i
    int tmp = arr[i];
    arr[i] = arr[index_min];
    arr[index_min] = tmp;
}
```

Reste à présent à se poser la question du nombre d'opérations nécessaires pour trier un tableau de taille $n$, et il n'est que trop temps d'aborder la notion de complexité algorithmique d'un programme.

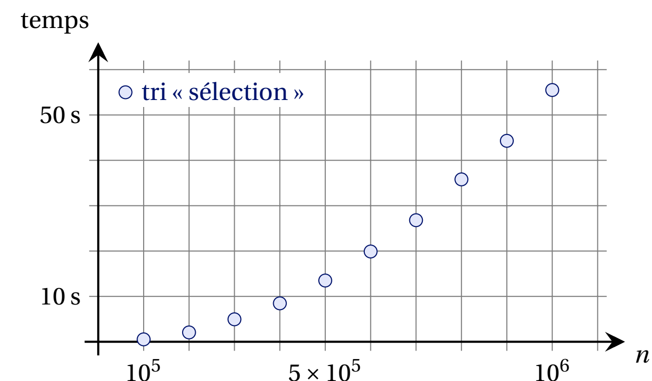
## 4.2 Complexité algorithmique

### Objectifs

Nous l'avons vu sur les exemples précédents, nous ne pouvons nous contenter d'un programme correct. Nous devons aussi nous efforcer de faire en sorte qu'il utilise efficacement les différentes « ressources » qui lui sont confiées : temps processeur, mémoire, bande passante... En effet, ces ressources ne sont pas infinies, aussi s'efforce-t-on généralement d'en faire une consommation aussi raisonnable que possible, et donc d'écrire des programmes qui s'exécutent rapidement, consomment peu de mémoire, etc.

Un pan entier de l'informatique, appelé étude de la *complexité algorithmique*, vise à étudier comment ces besoins évoluent lorsque le nombre $n$ d'éléments à traiter augmente. La *complexité* d'un programme correspond à l'évolution de ses besoins en une ressource en fonction de $n$. Nous nous intéresserons essentiellement à la complexité en temps, ou *complexité temporelle* (l'évolution du temps d'exécution par rapport à la quantité de données à traiter) et à la complexité en mémoire, ou *complexité spatiale* (l'évolution des besoins en mémoire).

Voici par exemple l'évolution du temps nécessaire à l'exécution du programme précédent triant un tableau de taille $n$ par sélection, en fonction de la taille $n$ du tableau [37], suggérant une dépendance en $n^2$ pour le temps de calcul :




### Complexité algorithmique et notations de Landau

Dans cette section, nous allons établir quelques notations et principes qui permettront de faciliter la description des besoins d'un programme. Notons $\mathcal{D}$ l'ensemble des entrées possibles d'un programme (les données qu'il devra traiter). On peut partionner $\mathcal{D}$ en différents ensembles $\mathcal{D}_n$ en regroupant les entrées représentant un volume similaire

37. Deux exécutions ne conduisent pas toujours rigoureusement au même temps d'exécution, mais les fluctuations dans la durée d'exécution sont très faibles pour cet algorithme.

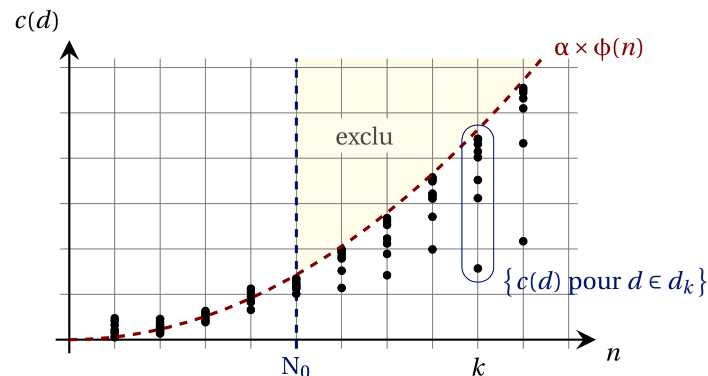
de données à traiter. Par exemple, si le programme s'occupe de trier des tableaux, $\mathcal{D}_n$ correspondra à l'ensemble des tableaux de taille $n$ que l'on pourra confier au programme.

Pour une entrée $d \in \mathcal{D}$, on notera $c(d)$ la consommation de la ressource pour traiter cette entrée (le temps d'exécution si on s'intéresse à la complexité temporelle, le nombre maximal de cellules mémoire occupées simultanément par le programme au cours de son exécution si l'on s'intéresse à la complexité spatiale [38]). Pour jauger sa dépendance avec le volume $n$ de données, on utilise les notations de Landau :

**Définition.** On dit que la consommation, dans le pire des cas, d'une ressource, pour un programme, est en $O\big(\phi(n)\big)$ s'il existe un entier $N_0 \in \mathbb{N}$, un réel $\alpha \in \mathbb{R}$ et une fonction $\phi$ tels que

$$\forall n \geqslant N_0,\ \max_{d \in \mathcal{D}_n} c(d) \leqslant \alpha \times \phi(n)$$

Cette notation s'intéresse donc à ce qui se passe pour de « grandes » valeurs de $n$. Elle affirme que pour $n$ suffisamment grand [39], on peut majorer les coûts de toutes les exécutions par une fonction dont la dépendance en $n$ est connue, sur l'exemple ci-dessous :




Nous utiliserons également deux autres notations :

**Définition.** On dit que la consommation, dans le pire des cas, d'une ressource, pour un programme, est en $\Omega\big(\phi(n)\big)$ s'il existe un entier $N_0 \in \mathbb{N}$, un réel $\beta \in \mathbb{R}$ et une fonction $\phi$ tels que

$$\forall n \geqslant N_0,\ \max_{d \in \mathcal{D}_n} c(d) \geqslant \beta \times \phi(n)$$

38. Il s'agit bien du maximum de cellules mémoire occupées à un instant, car la mémoire libérée peut être réutilisée ensuite pour d'autres opérations.

39. Ce qui signifie que la notion de complexité peut avoir un intérêt pratique limité dans certaines circonstances... Comme elle ne met aucune condition sur le préfacteur $\alpha$ ni sur le $N_0$, la fonction peut avoir une complexité intéressante... mais seulement pour des données dont la taille excède largement les capacités de stockage d'une machine! La complexité seule ne saurait donc totalement trancher entre deux algorithmes, comme nous aurons l'occasion de le vérifier.

**Définition.** On dit que la consommation, dans le pire des cas, d'une ressource, pour un programme, est en $\Theta\big(\phi(n)\big)$ si elle est à la fois en $O\big(\phi(n)\big)$ et $\Omega\big(\phi(n)\big)$, c'est-à-dire s'il existe un entier $N_0 \in \mathbb{N}$, deux réels $\alpha$ et $\beta \in \mathbb{R}$ et une fonction $\phi$ tels que

$$\forall n \geqslant N_0,\ \max_{d \in \mathcal{D}_n} \beta \times \phi(n) \leqslant c(d) \leqslant \alpha \times \phi(n)$$

C'est la notation $O(\bullet)$ que nous manipulerons le plus fréquemment. En effet, c'est généralement une majoration des ressources nécessaires qui est le plus éclairant. Il est fort rare que l'on se plaigne qu'un programme ne prend pas assez de temps ou ne consomme pas suffisamment de mémoire.

Les minorations peuvent cependant également être utiles. Par exemple, une consommation d'une ressource en $O\big(n^2\big)$ est, bien évidemment, également en $O\big(n^3\big)$. On s'efforcera naturellement d'obtenir la majoration la plus « serrée » possible, et on pourra fréquemment justifier que l'on a obtenu la meilleure majoration possible en exhibant une minoration.

Toutes les définitions précédentes font référence au « pire cas ». Là encore, c'est pour s'assurer que l'on ait une majoration de la consommation qui soit valable quelles que soient les données. S'il s'agit de contrôler la phase d'aterrissage d'un avion, on ne peut pas se permettre qu'une fois de temps en temps le calcul prennent plus de temps que prévu et qu'il n'en faut à l'avion pour entrer en collision avec le sol!

On peut cependant également considérer la complexité « *dans le meilleur des cas* ». On remplacera alors $\max_{d \in \mathcal{D}_n}$ dans les définitions par $\min_{d \in \mathcal{D}_n}$. Il est également possible de s'intéresser à la complexité « *en moyenne* », en majorant, minorant ou encadrant la consommation *moyenne* de la ressource. Dans ce dernier cas, toutefois, on peut avoir besoin de connaître la loi de probabilité des $d \in \mathcal{D}_n$, toutes les entrées possibles n'étant pas nécessairement équiprobables. Lorsque l'on parlera de complexité sans préciser, la majoration devra toujours être faite *dans le pire des cas*.

Les complexités que nous rencontrerons le plus fréquemment seront :

- $O(1)$, dite *complexité constante*, où la consommation de la ressource ne dépend pas du volume de données traitées ;
- $O(n)$, dite *complexité linéaire*, où elle en dépend linéairement ;
- $O\big(n^2\big)$, dite *complexité quadratique* ;
- $O\big(\log(n)\big)$, dite *complexité logarithmique* ;
- $O\big(n \times \log(n)\big)$, dite *complexité quasi-linéaire* [40] ;
- $O(k^n)$, dite *complexité exponentielle* [41].

40. Certains ouvrages parlent plutôt que complexité linéarologarithmique, pour réserver le terme quasilinéaire à la complexité $O\big(n \log^*(n)\big)$ où $\log^*$ est le logarithme itéré, correspondant au nombre de fois qu'il faut appliquer log à une valeur avant d'obtenir un résultat négatif.

41. Pour être exact, le terme de « complexité exponentielle » regroupe généralement toutes les complexités de la forme $O(k^{P(n)})$ où P est un polynôme, on parle parfois de « complexité exponentielle avec un exposant linéaire » pour spécifiquement faire référence à $O\big(k^n\big)$.

**Retour sur les algorithmes étudiés**

Si l'on reprends les exemples que l'on a déjà étudiés, les algorithmes élémentaires sur les tableaux que l'on a écrits en langage C, ainsi que les algorithmes de Boyer-Moore et de Kadane, étaient tous de complexité temporelle linéaire ($\Theta(n)$). Le tri par sélection était quant à lui de complexité temporelle quadratique ($\Theta\left(n^2\right)$).

Pour le calcul de la complexité temporelle [42], sauf circonstances très exceptionnelles, on n'essaie pas d'encadrer le temps d'exécution d'un algorithme. On se contente de déterminer la (ou les) opération(s) qui contribuent le plus au temps d'exécution, et on les dénombre.

Par exemple, dans le cas du tri par sélection, on effectue principalement $n(n-1)/2$ comparaisons d'éléments du tableau deux à deux, $n-1$ échanges d'éléments [43]. Ces opérations se font en un temps « constant », c'est-à-dire pouvant être encadré indépendamment de $n$. Cela nous permet de conclure à une complexité $\Theta\left(n^2\right)$.

Notons au passage que, comme c'est le cas en mathématiques, on ne gardera que le terme dominant (on n'écrira pas $\Theta\left(n^2+n\right)$), et il n'y a aucune raison de faire figurer un préfacteur dans la notation (on n'écrira pas non plus $\Theta\left(n^2/2\right)$). Pour la même raison, $\ln(n)$, $\log(n)$ ou $\log_2(n)$ de $n$ sont équivalents et interchangeables.

On peut se convaincre qu'il y aura toujours moins d'échanges que de comparaisons, et donc ne dénombrer que les comparaisons dans un tri tel que le tri sélection pour conclure. Et il n'est pas besoin de dénombrer aussi précisément le nombre de comparaisons effectuées. Il suffit de justifier que l'on en fait moins de $n^2$ (puisque chacune des deux boucles imbriquées ne peut être exécutée plus de $n$ fois), et que les $n/2$ premières itérations sur $i$ conduisent toutes, chacune, à plus de $n/2$ itérations sur $j$, aussi le nombre de comparaisons est d'au moins $n^2/4$. Cela suffit à conclure.

Attention, si les opérations que l'on a vu en C pour le moment (lecture et écriture en mémoire dans des variables ou des cases de tableau, opérations élémentaires sur les types `int`, `double` et `bool`...) sont *toutes* de coût constant [44], ce n'est pas forcément toujours le cas. Lorsque l'on s'intéresse au temps d'exécution, il faut dénombrer des opérations *élémentaires* pour le processeur. Nous aurons l'occasion de voir quelques opérations dont le coût n'est pas élémentaire, nous le signalerons alors en temps utile.

C'est l'occasion également de souligner que l'on parle ici de *complexité algorithmique*, laquelle est liée à une implémentation donnée d'un algorithme, donc dans un langage spécifique, et en lien avec une architecture particulière sur laquelle elle va tourner. Un même algorithme, implanté différemment (par exemple avec une structure de données différente), dans un langage différent voire exécuté sur une machine différente peut conduire à des complexités distinctes [45].

Il est possible de s'affranchir des questions d'implémentation et d'architecture pour réfléchir à la complexité intrinsèque d'un problème, un point qui sera abordé dans une étude plus complète de la théorie de la complexité.

**Importance du problème**

La complexité d'un algorithme est particulièrement importante car c'est, dans une large mesure, ce qui décidera si un algorithme peut être utilisé dans la pratique ou non. Dans le tableau ci-dessous, on a reporté l'ordre de grandeur du temps d'exécution d'un algorithme sur une machine courante effectuant de l'ordre de $10^{10}$ opérations par seconde, pour différents volumes de données $n$ :

| n | $O(1)$ | $O(\log(n))$ | $O(n)$ | $O(n\log(n))$ | $O(n^2)$ | $O(2^n)$ |
|---|---|---|---|---|---|---|
| 10 | 0,1 ns | 0,3 ns | 1 ns | 3 ns | 10 ns | 0,1 µs |
| $10^2$ | 0,1 ns | 0,7 ns | 0,01 µs | 0,07 µs | 1 µs | $> t_{\text{univers}}$ |
| $10^3$ | 0,1 ns | 1 ns | 0,1 µs | 1 µs | 0,1 ms | |
| $10^4$ | 0,1 ns | 1,3 ns | 1 µs | 0,01 ms | 10 ms | |
| $10^5$ | 0,1 ns | 1,7 ns | 0,01 ms | 0,17 ms | 1 s | |
| $10^6$ | 0,1 ns | 2 ns | 0,1 ms | 2 ms | 2 min | |
| $10^7$ | 0,1 ns | 2,3 ns | 1,0 ms | 23 ms | 3 h | |
| $10^8$ | 0,1 ns | 2,7 ns | 0,01 s | 0,3 s | 12 j | |
| $10^9$ | 0,1 ns | 3 ns | 0,1 s | 2 s | 3 a | |

On remarque que les complexités jusque $O\left(n\log(n)\right)$ sont généralement utilisables même sur de gros volumes de données, tandis que $O\left(n^2\right)$ peut poser des problèmes et que $O(2^n)$ est généralement inutilisable en pratique.

Inversement, on peut s'intéresser au volume maximal de données que l'on peut traiter en une minute sur la mêe machine, selon la complexité :

- un algorithme en $O(\log(n))$ peut traiter $\simeq 10^{180000000000}$ données ;
- un algorithme en $O(n)$ peut traiter $\simeq 6\times 10^{11}$ données ;
- un algorithme en $O(n\log(n))$ peut traiter $\simeq 2\times 10^{10}$ données ;
- un algorithme en $O(n^2)$ peut traiter $\simeq 8\times 10^{5}$ données ;
- un algorithme en $O(2^n)$ peut traiter $\simeq 40$ données.

Nous retrouvons naturellement les mêmes conclusions quant aux difficultés pratiques que peuvent présenter l'utilisation d'algorithmes à la complexité « élevée ». Il est par ailleurs aisé ici de voir pourquoi la complexité $O\left(n\log(n)\right)$ est qualifiée de « quasi-linéare » : les performances sont en effet très similaires avec la complexité linéaire.

42. On utilisera la même démarche également pour la complexité spatiale.

43. Ainsi que $n+n(n+1)/2$ incrémentations d'indices de boucle et autant de comparaisons avec la borne supérieure. Toutefois, comme on n'a jamais de boucle dont le corps est vide, les opérations $O(1)$ entre chaque itération d'une boucle coûtent, au plus, aussi cher que les opérations effectuées dans le corps de la boucle, mais jamais davantage, donc il n'est jamais utile de les considérer.

44. En supposant que l'architecture sur laquelle le programme est exécutée dispose de mémoire à accès direct (RAM), ce qui est vrai pour toutes les architectures courantes.

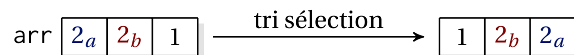
45. Et le compilateur peut parfois voir des optimisations qui vous ont échappé, et le code compilé peut parfois avoir une complexité moindre que le code source !

## 4.3 Le tri « bulle »

Revenons à la question des tris. Le tri sélection n'est pas fréquemment utilisé car, outre le fait qu'il n'est pas particulièrement efficace, il n'a pas non plus de propriétés particulièrement intéressantes. À l'inverse, il a le défaut de ne pas être un tri *stable*.

**Définition.** Un tri est dit *stable* si, pour deux éléments α et β, parmi les éléments triés, vérifiant α = β, leur ordre relatif est préservé par le tri (si α apparaît avant β dans le tableau initial, alors il apparaît avant β dans le tableau trié).

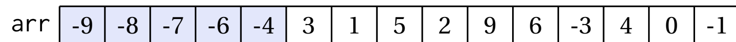
Cette notion suppose que l'on puisse différencier deux éléments considérés comme *égaux* lors du tri, comme si les entiers que l'on trie avaient des couleurs différentes, par exemple, permettant de les identifier de façon unique. Le tri sélection n'est pas stable car il peut inverser l'ordre des deux élements égaux, comme dans l'exemple ci-dessous :



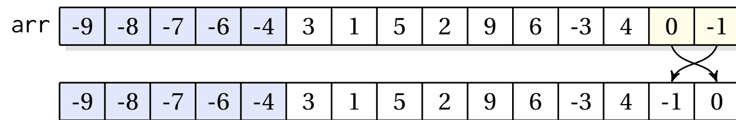

La stabilité est une caractéristique souvent recherchée pour un tri car elle permet de trier des données sur plusieurs critères : pour trier par exemple des personnes selon leur nom, et selon leur prénom lorsque plusieurs d'entre elles portent le même nom, on commence par trier les personnes par leur prénom, puis on les trie à nouveau par leur nom en choisissant un tri stable.

Le tri « bulle » est une variante du tri par sélection (en ce sens qu'il utilise le même invariant de boucle : après $i$ itérations, les $i$ premiers éléments sont bien placés) utilisant une façon différente de ramener en position $i$ le plus petit des éléments du tableau entre la case d'index $i$ et la fin du tableau, afin d'obtenir un tri stable.

Pour illustrer son fonctionnement, reprenons une situation dans laquelle les cinq premiers éléments sont à leur place, et où l'on s'efforce de placer le bon élément dans la case d'index 5 [46] :

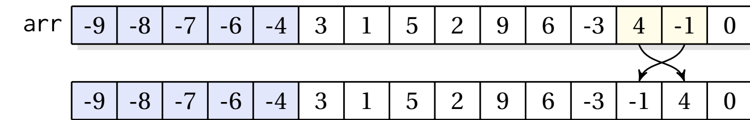



Le tri bulle procède en ne permutant que des éléments voisins dans le tableau. Pour ramener l'élément adéquat dans la case d'index $i$, il commence par regarder quel est le plus petit des deux derniers éléments (ici 0 et −1), et s'arrange pour que le plus petit des deux soit dans la case d'index $n-2$, en les permutant au besoin :

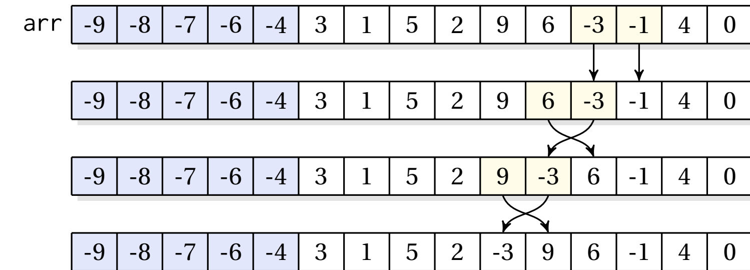



46. Les éléments dans les cases d'index 5 et supérieurs ne sont pas aux mêmes places que dans le cas du tri par sélection, car les cinq premières étapes ont également procédé différemment pour placer les cinq premiers éléments.

Puis on fait de même avec les avant-dernier et antépénulptième éléments, de façon à ce que le plus petit des deux soit en position $n-3$ :




On continue de comparer les éléments par paire de la droite vers la gauche, en ne les permutant que si nécessaire. Les prochaines étapes sont donc :




On voit qu'à tout instant, après l'examen d'une paire d'éléments et la permutation si nécessaire, l'élément situé à gauche est plus petit que tous les éléments à sa droite. La séquence va donc progressivement amener dans la case d'index $i = 5$ le plus petit des éléments qui n'ont pas encore été placés. Le nom de « bulle » vient de l'image d'une bulle qui progresse vers le haut en emportant à tout instant le plus petit élément qu'elle trouve [47] (pour que l'image soit pertinente, il faudrait représenter le tableau verticalement, la bulle progressant vers la gauche dans la représentation du tableau que l'on a choisie).

Une implémentation en langage C peut être :

```c
for (int i=0; i<n-1; ++i) {
    // les i premiers éléments sont bien placés
    for (int j=n-1; j>i; --j) {
        // l'élément dans la case d'index j est plus petit
        // que les éléments dans les cases d'index k>j
        if (arr[j-1] > arr[j]) {
            int tmp = arr[j];
            arr[j] = arr[j-1];
            arr[j-1] = tmp;
        }
    }
}
```

47. Signalons l'existence d'un tri « pierre » qui effectue la même chose que le tri bulle dans l'autre sens, les comparaisons étant effectuées du début du tableau vers la fin, emportant les plus grands éléments, de sorte qu'après $i$ séquences, les $i$ derniers éléments sont bien placés.

On remarque que l'invariant de la boucle indexée par `i` est bien le même que dans le cadre du tri par sélection, et que le second invariant, dans la boucle indexée par `j`, nous permet bien de maintenir ce premier invariant d'une itération à l'autre.

En terme de complexité, il est aisé de voir que l'on effectue $n(n-1)/2$ comparaisons d'éléments du tableau deux à deux et, dans le pire des cas, autant d'échanges (consistant chacun en trois copies). On a donc un tri qui, en terme de complexité temporelle, est quadratique quel que soit l'arrangement initial des données ($\Theta\left(n^2\right)$).

En revanche, le tri est bien stable, cette fois : la position relative de deux éléments $\alpha$ et $\beta$ ne peut changer que s'ils sont échangés deux à deux, et cela ne peut arriver que s'ils ne sont pas égaux !

On peut remarquer que l'on effectue en pratique un peu plus de copies d'éléments que nécessaire. Si par exemple le plus petit élément parmi les éléments restants se trouvait initialement en fin du tableau, on passera notre temps à le placer dans la variable `tmp` avant de le remettre dans le tableau. Cela ne change en rien la complexité, mais c'est quand même maladroit.

On préférera donc conserver le plus petit élément dans la variable à part, comme ci-dessous.

```c
for (int i=0; i<n-1; ++i) {
    // les i premiers éléments sont bien placés
    int smallest = arr[n-1];
    for (int j=n-1; j>i; --j) {
        // smallest contient le plus petit des éléments
        // des cases j à n-1
        if (arr[j-1] <= smallest) {
            arr[j] = smallest;
            smallest = arr[j-1];
        } else {
            arr[j] = arr[j-1];
        }
    }
    arr[i] = smallest;
}
```

La complexité de l'algorithme n'a pas changé (en particulier, le nombre de comparaisons est strictement identique), mais le nombre de copies a été quelque peu réduit. Attention toutefois : dans la version précédente, le contenu de la case d'index $j$ est arbitraire[48]. Les éléments du tableau correspondent aux entiers dans toutes les cases *excepté* la case d'index $j$ ainsi que l'élément dans la variable `smallest`.

48. Il s'agit d'un « duplicata », soit de l'élément dans `smallest`, soit de l'élément dans la case d'index $j+1$.

Reste un point intéressant concernant le tri bulle : supposons que, lors d'une itération $i$, la dernière permutation de deux éléments ait eu lieu entre les éléments placés dans les cases d'index $k$ et $k+1$. On sait avec certitude que l'élément dans la case d'index $k$ est plus petit que tous ceux situés dans les cases à sa droite. Mais puisqu'il n'y a plus eu de permutations ensuite, on sait également que tous les éléments à gauche de celui dans la case d'index $k$ sont rangés par ordre croissant. Par conséquent, tous les éléments dans les cases d'index 0 à $k$ (inclus) sont *tous* bien placés.

Par conséquent, on peut gagner du temps en effectuant le travail de plusieurs itérations sur $i$ d'un seul coup en mémorisant où a été effectuée la dernière permutation ! Pour ce faire, on transforme la boucle **`for`** sur `i` en une boucle **`while`**, en conservant toutefois le même invariant de boucle.

On mémorise, dans une variable `k_last_perm`, pour chaque itération du **`while`**, l'index `k` correspondant à la dernière permutation effectuée, car elle nous permettra de déterminer la valeur suivante de `i` garantissant que l'invariant de boucle est respecté. Si aucune permutation n'est effectuée, c'est que la liste est triée, et on voudra sortir. On initialise donc `k_last_perm` à `n-1` (par exemple) de façon à ce que cela provoque la sortie du **`while`** dans cette situation.

Cela s'écrira par exemple[49] :

```c
int i=0;
while (i<n-1) {
    // les i premiers éléments sont bien placés
    int k_last_perm = n-1;
    int smallest = arr[n-1];
    for (int j=n-1; j>i; --j) {
        // smallest contient le plus petit des éléments
        // des cases j à n-1
        if (arr[j-1] <= smallest) {
            arr[j] = smallest;
            smallest = arr[j-1];
        } else {
            arr[j] = arr[j-1];
            k_last_perm = j-1;
        }
    }
    arr[i] = smallest;
    i = k_last_perm+1;
}
```

49. En toute logique, on préférerait effectuer un décalage d'indices et écrire « `k_last_perm = j` » et « `i = k_last_perm` » afin d'éviter un « `+1` » et un « `-1` » inutiles, on a préféré ici rester aussi près que possible de la description de l'algorithme.

La présence d'une boucle `while` incite à la prudence quant à la terminaison de l'algorithme. Ici, l'algorithme termine bien car `k_last_perm` ne peut prendre qe des valeurs supérieures ou égales à $i$, ce sorte que la mise à jour de `i` à la fin de la boucle met dans `i` une valeur forcément strictement plus grande. `i` peut donc nous servir de variant de boucle, il s'agit d'une séquence d'entiers strictement croissante et majorée, il ne saurait y avoir de boucle infinie.

Cette dernière modification de notre tri bulle ne change pas sa complexité dans le pire des cas : il peut toujours être nécessaire d'effectuer $n-1$ itérations de la boucle `while` pour trier notre tableau [50]. Mais dans certains cas, on peut aller plus vite. Par exemple, lorsque les éléments du tableau sont déjà triés par ordre croissant : la première itération du `while` ne trouvera aucune permutation de deux éléments à effectuer, et l'algorithme s'arrêtera immédiatement.

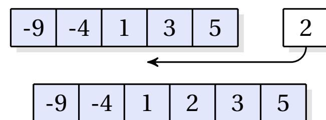
Cette dernière version a donc une complexité linéaire ($\Theta(n)$) dans le meilleur des cas, même si elle reste quadratique ($\Theta\left(n^2\right)$) dans le pire des cas. On conserve une complexité temporelle quadratique ($O\left(n^2\right)$), mais on n'est pas à l'abri d'une bonne surprise de temps en temps. Les tableaux pour lesquels l'algorithme pourrait s'averer inhabituellement efficace sont ceux pour lesquels les éléments sont à peu près triés, exceptés quelques-uns d'entre eux qui se trouvent un peu plus à droite dans le tableau qu'ils ne devraient l'être [51].

## 4.4 Le tri « insertion »

Considérons un tableau où les $n-1$ premiers éléments sont rangés par ordre croissant, mais où le dernier élément n'est pas nécessairement plus grand que tous les autres, comme ci-dessous :

| -9 | -4 | 1 | 3 | 5 | 2 |
|---|---|---|---|---|---|

Appliquons le principe de la « bulle » à ce tableau. On constate que le dernier élément va remonter dans le tableau, en passant devant des éléments plus grands que lui comme un coureur cycliste remonterait un peloton de concurrents, jusqu'à s'arrêter lorsqu'un élément plus petit que lui le précède, ou qu'il se retrouve en tête du tableau. Tout se passe comme si l'élément était « inséré » à la bonne place dans le tableau :

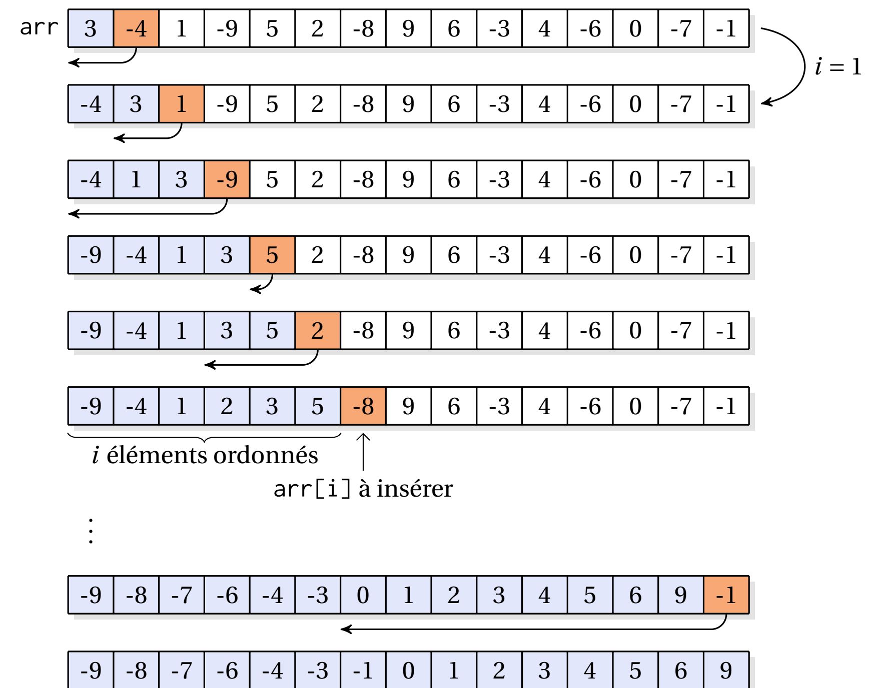



50. Par exemple si, initialement, tous les éléments du tableau ont rangés dans un ordre décroissant.
51. S'ils se situent un peu à gauche, c'est un tri pierre qui conviendrait le mieux. Si on n'a que quelques éléments quelque peu décalés, mais dans les deux sens, on peut s'intéresser au tri « cocktail » qui alterne les passes de « bulle » et de « pierre » pour profiter des deux effets.

Contrairement à ce qui se passait pour le tri bulle, il est inutile de poursuivre dès que deux éléments ne sont pas permutés, car on sait déjà que les éléments restants, à gauche, sont rangés dans un ordre croissant.

Le tri par insertion se base sur cette idée : à l'étape $i$, on suppose que les $i$ premiers éléments du tableau sont convenablement ordonnés (attention, ils ne sont pas nécessairement pour autant déjà à la bonne place). Il s'agira donc de notre invariant de boucle. Pour le faire progresser, il faut obtenir que les $i+1$ premiers éléments soient rangés par ordre croissant, donc on va faire progresser l'élément initialement dans la case d'index $i$ vers la gauche, à l'aide de permutations d'éléments voisins comme décrit précédemment, jusqu'à ce que l'élément qui le précède soit plus petit ou qu'il se trouve en tête du tableau.

Le tri par insertion des éléments d'un tableau se passe donc de la façon suivante :




Comme on le voit ci-dessus, pour la première itération, conformément à notre invariant de boucle, on choisira $i = 1$ : un élément tout seul est nécessairement déjà « trié » par ordre croissant. Le premier élément à « insérer » sera donc l'élément dans la case d'index 1. Puis on insèrera les éléments dans les cases d'index 2, 3, etc. Le dernier élément à insérer sera le dernier élément du tableau, situé dans la case d'index $n-1$.

Chaque insertion se déroule comme dans la seconde version proposée du tri bulle, à ceci près qu'un `while` remplace la boucle `for` : en effet, dès lors que deux éléments ne sont plus à permuter, on en a terminé avec l'insertion de l'élément.

En langage C, cela s'écrira donc par exemple :

```
for (int i=1; i<n; ++i) {
    // les i premiers éléments sont rangés par ordre croissant
    // on insère l'élément situé dans la case d'index i
    int elem = arr[i];
    int j=i;
    while (j>0 && elem<arr[j-1]) {
        arr[j] = arr[j-1];
        --j;
    }
    arr[j] = elem;
}
```

On notera, dans l'exemple précédent, l'utilisation de l'évaluation paresseuse : `arr[j-1]` n'est comparé à `elem` que si `j>0` est vrai, il ne peut donc y avoir d'accès à une case qui n'existe pas !

Dans la pratique, on utilisera souvent le fait que la construction « **for** » en langage C n'est qu'une façon un peu particulière d'écrire une boucle conditionnelle, et on écrira, de façon plus succinte quitte à être un peu moins immédiatement reconnaissable comme une boucle **while** :

```
for (int i=1; i<n; ++i) {
    // les i premiers éléments sont rangés par ordre croissant
    // on insère l'élément situé dans la case d'index i
    int j, elem = arr[i];
    for (j=i; j>0 && elem<arr[j-1]; --j) {
        arr[j] = arr[j-1];
    }
    arr[j] = elem;
}
```

Dans le pire des cas, comme pour les tris précédents, on effectue $n(n-1)/2$ comparaisons [52] (et $(n+1)(n+2)/2$ affectations [53]). Le tri par insertion est donc lui aussi quadratique dans le pire des cas.

Comme on l'a dit précédemment, il n'est pas utile de dénombrer précisément le nombre de comparaisons effectuées. Il suffit de remarquer que les $n/2$ dernières insertions, dans le pire des cas (où l'élément doit arriver en tête du tableau), conduisent chacune au moins à $n/2$ comparaisons. Donc cette seule partie de l'algorithme nécessite déjà $n^2/4$ opérations.

52. Toujours en ne comptant que les comparaisons entre éléments du tableau, sans tenir compte des boucles dont on a dit que le coût pouvait toujours être ignoré devant les autres coûts, ce qui inclue donc aussi la comparaison de j avec 0.

53. Même remarque, il s'agit d'affectations liés aux éléments du tableau.

Inversement, on ne saurait effectuer plus de $n$ fois $n$ comparaisons pour les $n$ insertions à faire, soit $n^2$ comparaisons au total. Cet encadrement permet directement de conclure à une complexité temporelle dans le pire des cas en $\Theta(n^2)$.

En revanche, comme dans la dernière version du tri bulle, dans le meilleur des cas, la complexité est linéaire. Cela arrive par exemple lorsque chaque insertion se termine immédiatement, lorsque le tableau est déjà trié.

Globalement, le tri par insertion est, lui aussi, un algorithme de tri en $O(n^2)$. Si l'on s'intéresse aux préfacteurs et à la probabilité de tomber sur un cas favorable qui réduit le nombre d'opérations à effectuer, on peut vérifier que pour de grands $n$, le tri par insertion est généralement celui qui s'en sort avec un léger avantage. Toutefois, nous verrons que pour de « grands [54] » $n$, il existe des tris plus efficaces que ceux présentés dans le présent chapitre.

Les tris quadratiques restent utiles pour trier des tableaux de très petite taille. Sur de tels tableaux, pour les architectures actuelles, c'est généralement la seconde version proposée ici du tri bulle qui se révèle le plus rapide des tris proposés dans ce chapitre.

# 5 Tableaux multidimensionnels

## 5.1 Principe

Terminons ce tour d'horizon des tableaux en parlant de tableaux multidimensionnels. Il s'agit de tableaux pour lesquels on utilise généralement plusieurs indices pour désigner une case. Par exemple pour des tableaux à deux dimensions, tels que celui ci-dessous :

| | | |
|---|---|---|
| 22 | 42 | 17 |
| 37 | 54 | 11 |

Dans un tel tableau, on peut repérer une case en indiquant la ligne et la colonne de l'élément auquel on veut faire référence. Dans la suite, on supposera que le numéro de ligne est désigné par `i` et le numéro de colonne par `j`, et que la numérotation débute à zéro. Ainsi, la case contenant la valeur 37 correspond à `i=1` et `j=0`.

La première solution pour représenter un tel tableau en langage C est d'utiliser un tableau « normal ». Il suffit en effet de choisir une numérotation quelconque des cases et de s'en servir comme index dans un tableau classique. La numérotation la plus courante considère les cases ligne par ligne :

| | | |
|---|---|---|
| ⓪ | ① | ② |
| ③ | ④ | ⑤ |

54. La notion de « grand » $n$ est à relativiser, dans le cas présent, pour $n = 10$, le tri par insertion sera déjà moins bon que les tris que nous verrons ultérieurement.

Le tableau à deux dimensions ci-dessus pourra donc être représenté par le tableau C suivant :

| 22 | 42 | 17 | 37 | 54 | 11 |
|---|---|---|---|---|---|

Un élément sur la ligne `i` et la colonne `j` du tableau sera donc placé dans la case d'index `i*nbcols+j` où `nbcols` désigne le nombre de colonnes du tableau à deux dimensions.

On peut également numéroter les cases colonne par colonne :

| ⓪ | ② | ④ |
|---|---|---|
| ① | ③ | ⑤ |

Cette numérotation conduit au tableau C suivant :

| 22 | 37 | 42 | 54 | 17 | 11 |
|---|---|---|---|---|---|

L'élément sur la ligne `i` et la colonne `j` du tableau sera alors placé dans la case d'index `i+j*nbrows` où `nbrows` désigne le nombre de lignes du tableau à deux dimensions.

Le principe s'étend à des tableaux à plus de deux dimensions. C'est un principe très fréquemment utilisé car la représentation par un tableau à une dimension est très simple et souple. Le seul inconvénient est qu'il faut travailler quelque peu pour calculer l'index de la case dans le tableau C correspondant à la case qui nous intéresse dans le tableau à plusieurs dimensions.

En fait, on peut même adopter une formule très générale pour le calcul de l'index : « `start + i*s_i + j*s_j` ». Si on utilise par défaut la représentation ligne par ligne, on choisira donc `start = 0`, `s_i = nbcol` et `s_j = 1`.

Avec cette représentation générale, si on a besoin d'obtenir la transposée du tableau (les lignes deviennent des colonnes et inversement), il n'est pas besoin de déplacer les données, il suffit de redéfinir les valeurs de `s_i` et `s_j`, en prenant `s_i = 1` et `s_j = nbrows`. Cela se généralise à d'autres opérations. En partant du tableau initial, on peut :

- inverser l'ordre des colonnes en prenant `start = nbcols-1`, `s_i = nbcols` et `s_j = -1`;
- inverser l'ordre des lignes avec `start = (nbrows-1)*nbcols`, `s_i = -nbcols` et `s_j = 1`;
- réduire le tableau à ses seules colonnes d'index pair[55] en choisissant `start = 0`, `s_i = nbcols` et `s_j = 2`...

Si vous avez eu l'occasion d'utiliser le module `numpy` de Python, vous retrouvez ici quelques-unes des opérations sur les `numpy.ndarray` qui, de façon un peu surprenante, ont une complexité constante. La raison est simplement que `numpy` utilise cette représentation pour ses tableaux en interne, même s'il la cache à ses utilisateurs en s'occupant du calcul des index.

55. L'ensemble des données du tableau original reste conservé en mémoire, mais les numéros de ligne et de colonne font référence au tableau « réduit ».

## 5.2 Tableaux C à plusieurs dimensions

Il peut toutefois être intéressant de manipuler des tableaux à plusieurs dimensions. Notre tableau d'exemple peut être vu comme un tableau contenant deux lignes, chaque ligne étant elle-même un tableau contenant trois cases. Autrement dit, le tableau à deux dimensions peut être vu comme un tableau de tableaux de trois entiers.

Pour définir un tel objet en C, on utilise la déclaration suivante :

```c
int arr[2][3];
```

Cette déclaration doit se lire comme le fait que `arr` est un tableau de taille `2` contenant des `int[3]`, soit des tableaux de trois entiers, ce qui correspond bien à ce que l'on veut. L'initialisation peut se faire en utilisant des accolades imbriquées[56] :

```c
int arr[2][3] = { {22, 42, 17},
                  {37, 54, 11} };
```

Il est possible, comme pour des tableaux à une dimension, d'utiliser des initialisations incomplètes, telles que :

```c
int arr[2][3] = { {22, 42}, {37} };
```

Dans le cas précédent, le dernier élément de la première ligne et les deux derniers éléments de la seconde ligne, manquants dans l'initialisation, sont simplement initialisés à 0. On peut également ne pas spécifier toutes les lignes, comme ci-dessous, auquel cas elles seront intégralement initialisées à 0 :

```c
int arr[2][3] = { {22, 42} };
```

La manière la plus rapide d'initialiser un tableau à deux dimensions intégralement avec des zéros est donc :

```c
int arr[2][3] = { {0} };
```

Il est possible, si on fournit une initialisation de ne pas spécifier la *première* dimension du tableau, elle sera déduite des données d'initialisation. Mais cela ne fonctionne *que* pour la première. La déclaration suivante construit donc bien un tableau de taille $2 \times 3$ :

```c
int arr[][3] = { {22, 42}, {37} };
```

Pour accéder à une case du tableau, on va utiliser la même notation que pour les tableaux à une seule dimension, mais en renseignant l'index dans chaque dimension. Par exemple,

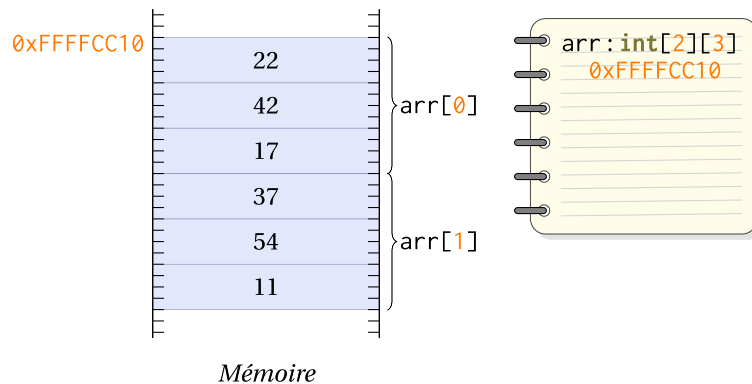
56. Il n'est pas besoin d'aller à la ligne dans l'initialisation, ce choix n'a été fait ici que pour faciliter la lecture du tableau que l'on crée, toujours dans un but de lisibilité du code.

pour afficher l'élément 37 sur la seconde ligne (index 1) et la première colonne (index 0), on utilisera :

```c
printf("L'élément ligne 1 colonne 0 est %d\n", arr[1][0]);
```

On pourrait envisager ne renseigner qu'un seul des index. « `arr[1]` » est un objet de type `int[3]`, donc un tableau à une dimension représentant la seconde ligne. Seulement, comme il s'agit d'un tableau, le langage C ne nous fournit guère de façon d'en faire quelque chose pour le moment (il est impossible, notamment, de s'en servir dans une affectation).

Dans la mémoire, le compilateur va donc créer des « blocs » de `int[3]` consécutifs. Le tableau va donc être stocké de la façon suivante :




*Mémoire*

Lorsque l'on tente d'accéder à une case du tableau, les mêmes restrictions s'imposent que pour un tableau à une dimension : les index fournis doivent être valides (c'est-à-dire positifs et strictement inférieurs au nombre de cases dans la dimension correspondante), sinon le comportement du programme sera indéfini. Ainsi, pour le tableau précédent, il est illégal d'écrire « `arr[2][1]` », « `arr[2][-2]` » ou « `arr[0][5]` » [57]

Terminons en précisant que si, pour un tableau à deux dimensions, le premier index désigne *généralement* le numéro de ligne et le second le numéro de colonne, il ne s'agit que d'une convention. C'est au programmeur de choisir la représentation qui lui convient. On peut parfaitement déclarer le tableau d'exemple avec

```c
int arr[3][2] = { {22, 37}, {42, 54}, {17, 11} };
```

Dans ce cas, pour accéder à un élément ligne `i`, colonne `j`, on écrira alors :

```c
printf("L'élément ligne i colonne j est %d\n", arr[j][i]);
```

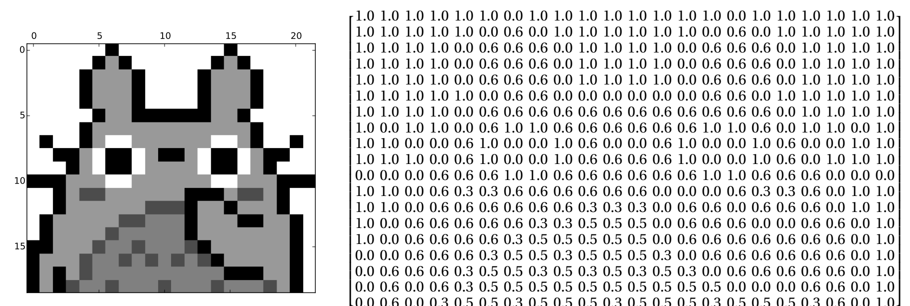
57. Même si, dans les deux derniers cas, le calcul de la position en mémoire à partir des deux index fournis pourrait faire référence à une case du tableau. Comme le comportement du programme est « indéfini », il est possible que ces deux dernières écritures fonctionnent, mais cela reste illégal.

## 5.3 Représenter des images

Les images sont des applications importantes des tableaux à plusieurs dimensions. Dans le domaine du numérique, les photographies et images sont en fait des mosaïques, car il n'est possible de mémoriser qu'un nombre fini d'informations [58]. Elles sont en général constituées d'un agencement régulier de zones de couleur et d'intensité uniforme, toutes rectangulaires et de même taille. On donne à ces éléments le nom de *pixels* (pour *picture element*).

Il est donc naturel de représenter une image en noir et blanc par simple tableau à deux dimensions, contenant dans chaque case une intensité. On représente généralement cette intensité soit par un nombre flottant (0.0 représentant généralement du noir, 1.0 du blanc), soit par un entier entre 0 et $2^p - 1$ (correspondant généralement, là encore, respectivement à du noir et à du blanc). La valeur de $p$ la plus fréquente est $p = 8$ (ce qui donne 256 valeurs distinctes pour l'intensité), mais cette valeur peut changer en fonction des utilisations.

Un exemple d'image noire et blanc et le tableau (contenant des flottants) la représentant :




$$
\begin{bmatrix}
1.0 & 1.0 & 1.0 & 1.0 & 1.0 & 1.0 & 0.0 & 1.0 & 1.0 & 1.0 & 1.0 & 1.0 & 1.0 & 1.0 & 1.0 & 0.0 & 1.0 & 1.0 & 1.0 & 1.0 & 1.0 & 1.0 \\
1.0 & 1.0 & 1.0 & 1.0 & 1.0 & 0.0 & 0.6 & 0.0 & 1.0 & 1.0 & 1.0 & 1.0 & 1.0 & 1.0 & 0.0 & 0.6 & 0.0 & 1.0 & 1.0 & 1.0 & 1.0 & 1.0 \\
1.0 & 1.0 & 1.0 & 1.0 & 0.0 & 0.6 & 0.6 & 0.6 & 0.0 & 1.0 & 1.0 & 1.0 & 1.0 & 0.0 & 0.6 & 0.6 & 0.6 & 0.0 & 1.0 & 1.0 & 1.0 & 1.0 \\
1.0 & 1.0 & 1.0 & 1.0 & 0.0 & 0.6 & 0.6 & 0.6 & 0.0 & 1.0 & 1.0 & 1.0 & 1.0 & 0.0 & 0.6 & 0.6 & 0.6 & 0.0 & 1.0 & 1.0 & 1.0 & 1.0 \\
1.0 & 1.0 & 1.0 & 1.0 & 0.0 & 0.6 & 0.6 & 0.6 & 0.0 & 1.0 & 1.0 & 1.0 & 1.0 & 0.0 & 0.6 & 0.6 & 0.6 & 0.0 & 1.0 & 1.0 & 1.0 & 1.0 \\
1.0 & 1.0 & 1.0 & 1.0 & 1.0 & 0.0 & 0.6 & 0.6 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.6 & 0.6 & 0.0 & 1.0 & 1.0 & 1.0 & 1.0 & 1.0 \\
1.0 & 1.0 & 1.0 & 1.0 & 0.0 & 0.6 & 0.6 & 0.6 & 0.6 & 0.6 & 0.6 & 0.6 & 0.6 & 0.6 & 0.6 & 0.6 & 0.6 & 0.0 & 1.0 & 1.0 & 1.0 & 1.0 \\
1.0 & 0.0 & 1.0 & 1.0 & 0.0 & 0.6 & 1.0 & 1.0 & 0.6 & 0.6 & 0.6 & 0.6 & 0.6 & 0.6 & 1.0 & 1.0 & 0.6 & 0.0 & 1.0 & 1.0 & 0.0 & 1.0 \\
1.0 & 1.0 & 0.0 & 0.0 & 0.6 & 1.0 & 0.0 & 0.0 & 1.0 & 0.6 & 0.0 & 0.0 & 0.6 & 1.0 & 0.0 & 0.0 & 1.0 & 0.6 & 0.0 & 0.0 & 1.0 & 1.0 \\
1.0 & 1.0 & 1.0 & 0.0 & 0.6 & 1.0 & 0.0 & 0.0 & 1.0 & 0.6 & 0.6 & 0.6 & 0.6 & 1.0 & 0.0 & 0.0 & 1.0 & 0.6 & 0.0 & 1.0 & 1.0 & 1.0 \\
0.0 & 0.0 & 0.0 & 0.6 & 0.6 & 0.6 & 1.0 & 1.0 & 0.6 & 0.6 & 0.6 & 0.6 & 0.6 & 0.6 & 1.0 & 1.0 & 0.6 & 0.6 & 0.6 & 0.0 & 0.0 & 0.0 \\
1.0 & 1.0 & 0.0 & 0.6 & 0.3 & 0.3 & 0.6 & 0.6 & 0.6 & 0.6 & 0.6 & 0.6 & 0.0 & 0.0 & 0.0 & 0.6 & 0.3 & 0.3 & 0.6 & 0.0 & 1.0 & 1.0 \\
1.0 & 1.0 & 0.0 & 0.6 & 0.6 & 0.6 & 0.6 & 0.6 & 0.6 & 0.3 & 0.3 & 0.3 & 0.0 & 0.6 & 0.6 & 0.0 & 0.6 & 0.6 & 0.6 & 0.0 & 1.0 & 1.0 \\
1.0 & 0.0 & 0.6 & 0.6 & 0.6 & 0.6 & 0.6 & 0.3 & 0.3 & 0.5 & 0.5 & 0.5 & 0.0 & 0.6 & 0.6 & 0.6 & 0.0 & 0.0 & 0.6 & 0.6 & 0.0 & 1.0 \\
1.0 & 0.0 & 0.6 & 0.6 & 0.6 & 0.6 & 0.3 & 0.5 & 0.5 & 0.5 & 0.5 & 0.5 & 0.0 & 0.6 & 0.6 & 0.6 & 0.6 & 0.6 & 0.6 & 0.6 & 0.0 & 1.0 \\
0.0 & 0.0 & 0.6 & 0.6 & 0.6 & 0.3 & 0.5 & 0.5 & 0.3 & 0.5 & 0.5 & 0.5 & 0.3 & 0.0 & 0.6 & 0.6 & 0.6 & 0.6 & 0.6 & 0.6 & 0.0 & 1.0 \\
0.0 & 0.6 & 0.6 & 0.6 & 0.3 & 0.5 & 0.5 & 0.3 & 0.5 & 0.3 & 0.5 & 0.3 & 0.5 & 0.3 & 0.0 & 0.6 & 0.6 & 0.6 & 0.6 & 0.6 & 0.0 & 1.0 \\
0.0 & 0.6 & 0.0 & 0.6 & 0.3 & 0.5 & 0.5 & 0.5 & 0.5 & 0.5 & 0.5 & 0.5 & 0.5 & 0.5 & 0.5 & 0.0 & 0.0 & 0.0 & 0.6 & 0.6 & 0.0 & 1.0 \\
0.0 & 0.6 & 0.0 & 0.3 & 0.5 & 0.5 & 0.3 & 0.5 & 0.5 & 0.5 & 0.3 & 0.5 & 0.5 & 0.5 & 0.3 & 0.5 & 0.5 & 0.5 & 0.3 & 0.6 & 0.0 & 1.0
\end{bmatrix}
$$

La plupart des formats de fichiers enregistrent une image sous la forme de tableaux d'entiers. Cependant, lorsque l'on modifie une image, il n'est pas rare d'utiliser des flottants, car ils permettent plus de précision dans les calculs intermédiaires. Lors de l'enregistrement du résultat, une fois les modifications terminées, chaque intensité est simplement « arrondie » à l'intensité la plus proche représentable par le format de fichier choisi.

Pour les images couleurs, ce n'est pas beaucoup plus compliqué. En théorie, il faudrait, pour représenter parfaitement une image en couleur, indiquer, pour chaque pixel, l'intensité de chacune des longueurs d'ondes visibles. Le problème, c'est qu'il y en a une infinité...

Cependant, les cellules photosensibles situés dans la rétine de l'œil humain sont de deux types, appelés batonnets et cônes. Les bâtonnets sont essentiellement sensibles à l'intensité et non à la couleur, et sont surtout présents en nombre au niveau de la vision

58. Les photographies argentiques sont en fait aussi des mosaïques, constituées de grains de chlorure d'argent plus ou moins noircis par la lumière reçue, même si leur répartition n'est pas régulière.

périphérique. Au centre de la rétine, ce sont les cônes, qui eux sont sensibles à la couleur, qui sont les plus nombreux.

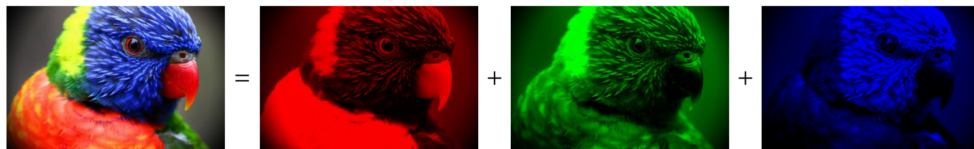
Or on ne trouve sur la rétine que trois variétés de cônes (erytholabes, chlorolabes et cyanolabes), chacun ayant sa propre courbe de sensibilité aux différentes longueurs d'ondes, centrées respectivement sur le rouge, le vert et le bleu [59].

Lorsque la rétine reçoit un rayonnement dont la longueur d'onde correspond au « jaune » du spectre, les cônes erytholabes et chlorolabes réagissent, et le cerveau a pour habitude d'associer cela à la couleur jaune. Cependant, il est incapable de faire la différence entre une longueur d'onde jaune, et un rayonnement mélangeant des longueurs d'onde verte et rouge, puisque cela fait réagir les mêmes cônes.

Cela a une conséquence intéressante : pour « simuler » n'importe quelle couleur que le cerveau peut identifier, il suffit de pouvoir générer un mélange de trois longeurs d'onde, rouge, verte et bleue, associées aux différentes variétés de cônes. On parle de synthèse additive.

C'est le principe utilisé par la quasi-totalité des dispositifs d'affichage, où chaque pixel coloré est en fait une association d'une source émettant de la lumière rouge, d'une source émettant de la lumière verte, et d'une source émettant de la lumière bleue, dans des proportions bien choisies pour produire n'importe quelle couleur.

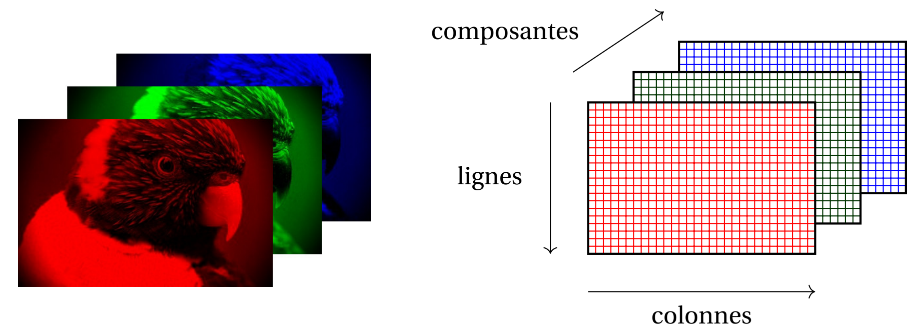
Une image en couleur est donc la combinaison d'une image rouge, d'une image verte, et d'une image bleue. L'exemple ci-dessous illustre la décomposition d'une image en couleur en trois images rouge, verte et bleue.




Le bec rouge est essentiellement visible sur la première des trois images, tandis que le plumage à l'avant du crâne, d'un bleu tirant légèrement sur le vert, apparaît tout naturellement principalement sur la troisième. Le plumage à l'arrière du crâne, de couleur jaune, est obtenue en faisant parvenir à l'œil le mélange de rouge et de vert, et se trouve donc visible sur les deux premières images.

Les images en couleurs sont donc enregistrées sur le même principe : pour chaque pixel (chaque « case »), on mémorise *trois* valeurs d'intensité, pour chacune des trois couleurs (ou *composantes*). On travaillera donc généralement avec des tableaux à trois dimensions, de taille `nbrows` × `nbcols` × 3.

59. Les « défauts » de vision des couleurs, très fréquents, consiste en une sensibilité moindre, ou plus rarement nulle, de l'une des trois variétés de cônes, ou en un déplacement du maximum de sensibilité de l'une d'elles.




Ainsi, pour un tableau `img` à trois dimensions [60] représentant une image en couleur, `img[17][42][0]` indiquera par exemple la quantité de rouge dans le pixel à la 18e ligne et la 43e colonne, tandis que `img[17][42][1]` correspondra à la quantité de vert et `img[17][42][2]` à celle de bleu.

Il existe d'autres systèmes de représentation des couleurs, plus ou moins équivalents, qui permettent de représenter des espaces colorimétriques un peu plus étendus, ou corrigent certains défauts du RGB (dans lequel des valeurs différentes peuvent représenter des teintes fort peu discernables, quand elles pourraient être utilisées plus efficacement ailleurs). Certains sont plus spécifiquement destinés à d'autres tâches, en vidéo par exemple.

Des applications dont l'observation à l'œil n'est pas le but peuvent également utiliser plus de trois composantes de couleurs (par exemple en astrophotographie, où l'on peut considérer un grand nombre de longueurs d'ondes, y compris en dehors du visible).

En imprimerie, on procède de la même façon, mais en synthèse soustractive, avec des encres qui *absorbent* chacune des couleurs élémentaires : le rouge (avec de l'encre cyan), le vert (avec de l'encre magenta) et le bleu (avec de l'encre jaune).

Il peut d'ailleurs arriver que des images destinées à l'imprimerie soient stockées sous la forme de tableau « CMY » où sont mémorisées les quantité de chacune des encres primaires pour chaque pixel (ou souvent « CMYK » où l'on précise aussi une quantité d'encre noire, le mélange des trois autres encres ne donnant généralement pas un noir assez sombre).

60. Les trois dimensions font respectivement référence à l'indice de la ligne, l'indice de la colonne et l'indice de la composante de couleur (0, 1 et 2 respectivement pour rouge, vert et bleu), et non au fait qu'on représente les images en mélangeant trois couleurs primaires !

# Exercices

## Ex. 3.1 – Fond de caisse

Le prix d'entrée d'une manifestation est de cinq euros. Un groupe de `n` personnes, placés dans une file d'attente, souhaite assister à cette manifestation. Cependant, tous n'ont pas l'appoint pour payer l'entrée : chacune de ces personnes dispose d'un unique billet, de cinq, de dix ou de vingt euros. Ces informations sont regroupées dans un tableau `arr` de taille `n`, tel que :

| arr | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 5 | 10 | 5 | 20 | 5 | 5 | 10 | 20 | 5 | 5 | 10 | 10 | 20 | 5 |

Dans cet exemple, la première des personnes dans la file possède un billet de cinq euros, la seconde un billet de dix euros, etc. On souhaite savoir si la personne au guichet (dont la caisse initialement est vide) pourra rendre la monnaie à tout le monde. Dans l'exemple proposé, on rendra la monnaie à la seconde personne avec le billet de la première entrée, et à la quatrième personne avec les billets des seconde et troisième entrées, mais il n'est pas possible de rendre la monnaie à l'avant-dernier visiteur.

**1.** Écrire un programme C déterminant s'il sera possible de faire entrer tous les visiteurs dans l'ordre où ils se présentent, ou si la personne au guichet se retrouvera dans l'impossibilité de rendre la monnaie.

**2.** Même question si l'on se permet de modifier l'ordre des personnes dans la file.

## Ex. 3.2 – Problème du char d'assault allemand

Connaître les capacités de production ennemies est d'un grand intérêt en temps de guerre. Durant la seconde guerre mondiale, espions et mathématiciens se sont disputés quant au nombre de chars d'assaut que l'Allemagne nazie produisait chaque mois (plus d'un millier d'après les espions, quelques centaines d'après les mathématiciens). Il s'est avéré après guerre que les mathématiciens avaient des chiffres quasiment exacts.

Pour estimer la production, ils se sont basés sur les numéros de série des boîtes de vitesses des chars détruits sur le champ de bataille. Dans le cas (un peu simplifié) d'une production de N boîtes de vitesses numérotées de 1 à N, on peut estimer N avec la formule suivante :

$$\hat{\mathrm{N}} = m + \frac{m-k}{k}$$

où $m$ est le plus grand numéro de série observé, et $k$ le nombre de numéros de série dont on dispose. Cela peut se comprendre : il s'agit d'ajouter au plus haut numéro de série une estimation de l'écart moyen entre deux numéros observés.

Cette technique a été utilisée avec succès à de nombreuses reprises ensuite, depuis l'estimation du nombre de V2 produits par les nazis jusqu'au nombre d'iPhones produits par Apple.

**1.** En supposant que `serials` est un tableau d'entiers contenant des numéros de série, et `k` un entier correspondant à sa taille, écrire un programme C estimant le nombre de chars d'assaut produits.

On peut obtenir un nombre entier choisi aléatoirement dans $[\![1\,..\,\mathrm{N}]\!]$ en langage C en écrivant « `rand()%N + 1` » (après avoir inclus le fichier d'en-tête `stdlib.h`).

**2.** Pour plusieurs valeurs de N et de $k$, construire un tableau `serials` (en évitant les doublons dans le tableau !) et évaluer le bon fonctionnement de l'estimateur.

## Ex. 3.3 – Échanges successifs

On considère un tableau `arr` de taille `n` contenant exactement les entiers de 0 à $n-1$, dans un ordre quelconque. On se proposer d'appliquer l'algorithme suivant : tant que la première case du tableau ne contient pas 0, on échange le contenu de la case d'index `0` et de la case d'index `arr[0]`.

**1.** Qu'obtient-on à l'issue de cet algorithme avec le tableau ci-dessous ?

| arr | | | | | |
|---|---|---|---|---|---|
| 5 | 4 | 2 | 1 | 0 | 3 |

**2.** L'algorithme termine-t-il toujours ? Le montrer ou proposer un contre-exemple.

**3.** Proposer une implémentation de cet algorithme en langage C.

## Ex. 3.4 – Sommes non divisibles

Soit une partie finie $\mathcal{P}$ de l'ensemble des entiers positifs $\mathbb{N}$ de cardinal $n$, et $k$ un entier positif.

On souhaite déterminer le cardinal du plus grand sous-ensemble $\mathcal{P}'$ de $\mathcal{P}$ tel que pour toute paire $(x, y)$ d'éléments distincts de $\mathcal{P}'$, $x + y$ n'est pas divisible par $k$.

On suppose $\mathcal{P}$ représenté par un tableau `P` de taille `n`. Par exemple :

| P | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 2 | 4 | 5 | 6 | 8 | 9 | 10 | 11 | 12 | 14 | 15 |

Les entiers de `P` sont supposés tous distincts, mais pas nécessairement rangés par ordre croissant. Pour ce tableau `P`, et pour `k=6`, on peut trouver des sous-ensembles de $\mathcal{P}$ avec 7 éléments et vérifiant cette propriété (par exemple, $\{2, 5, 6, 8, 9, 11, 14\}$) mais aucun avec davantage d'éléments.

Proposer un programme qui, à partir de `arr`, `n` et `d` détermine le cardinal du (des) plus grand(s) sous-ensemble(s) de $\mathcal{P}$ vérifiant la propriété.

Note : il n'est pas nécessaire de tester toutes les sommes, et encore moins tous les sous-ensembles !

## Ex. 3.5 – Suite d'Oldenbuger-Kolakoski

La suite d'Oldenbuger-Kolakoski [61] est une suite autoréférente infinie de 1 et 2, débutant par 1, qui est son propre codage par plages, c'est-à-dire que le nombre de répétitions successives d'un même chiffre est décrit par les chiffres eux-mêmes.

Les 25 premiers chiffres de cette suite sont :

$$\underbrace{1,}_{1}\ \underbrace{2,2,}_{2}\ \underbrace{1,1,}_{2}\ \underbrace{2,}_{1}\ \underbrace{1,}_{1}\ \underbrace{2,2,}_{2}\ \underbrace{1,}_{1}\ \underbrace{2,2,}_{2}\ \underbrace{1,1,}_{2}\ \underbrace{2,}_{1}\ \underbrace{1,1,}_{2}\ \underbrace{2,2,}_{2}\ \underbrace{1,}_{1}\ \underbrace{2,}_{1}\ \underbrace{1,1,}_{2}\ \underbrace{2,}_{1}\ \underbrace{1,}_{1}\ \ldots$$

**1.** Proposer un programme affichant les 1000 premiers chiffres de cette suite.

**2.** Faire de même avec d'autres chiffres, par exemple 1 et 3, ou 2 et 3.

## Ex. 3.6 – Nombres anti-Fibonacci

La suite des nombres anti-Fibonacci [62] est une suite croissante d'entiers strictement positifs. Chaque nombre de la suite est la somme de deux entiers strictement positifs *ne figurant pas dans la suite*, chaque entier absent ne servant qu'une seule fois. Les entiers constituant la suite doivent être les plus petits possibles.

Ses douze premiers termes sont : 3, 9, 13, 18, 23, 29, 33, 39, 43, 49, 53, 58. En effet, 3 est la somme $1+2$, $9=4+5$, $13=6+7$, $18=8+10$, $23=11+12$, etc.

Proposer un programme C affichant les entiers de cette suite inférieurs ou égaux à 1000.

Note : on pourra vérifier que dans la liste, on trouve tous les entiers congrus à 3 modulo 10, et que tous les autres sont congrus à 8 ou 9 modulo 10.

## Ex. 3.7 – Complexité et escape game

Vous vous trouvez dans face à un mur dans l'obscurité totale. Vous savez qu'il existe une sortie, mais vous ne savez pas si elle se trouve à droite ou à gauche. Pour toutes considérations pratiques, le mur est peut être considéré comme infini de part et d'autre du point de départ.

Proposer un algorithme permettant de trouver la sortie aussi efficacement que possible, et estimer sa complexité en fonction de la distance, en mètres, de la sortie par rapport au point de départ.

61. Suite A000002 de l'OEIS.
62. Suite A075326 de l'OEIS.

## Ex. 3.8 – Le juste prix

Un produit a un prix entier $p \in [\![1\,..\,n]\!]$ que l'on ne connaît pas. On se propose de déterminer ce prix en effectuant des propositions successives. Si l'on propose le bon prix, on gagne. Si l'on propose un prix strictement inférieur à $p$, on continue. Si l'on propose un prix strictement supérieur à $p$, la *première fois*, on peut continuer, mais la seconde, on perd la partie.

Proposer une stratégie garantissant que l'on va gagner la partie (on ne fait jamais deux propositions strictement supérieures à $p$) de manière à effectuer le minimum de propositions dans le pire des cas. Estimer ce nombre de propositions.

## Ex. 3.9 – Sous-tableau de somme maximale

On dispose d'un tableau d'entiers `arr` de taille `n`, ainsi qu'un entier $k \in [\![1\,..\,n]\!]$. Proposer un programme C déterminant l'index $i \in [\![0\,..\,n-k]\!]$ qui maximise la quantité

$$\sum_{i \leqslant j < i+k} \texttt{arr}[j]$$

On s'efforcera de trouver une solution de complexité temporelle linéaire en `n` et ne dépendant pas de `k` (en particulier, si `k` est de l'ordre de `n/2`, on souhaite que la complexité reste linéaire).

## Ex. 3.10 – Réordonnancement

On considère un tableau `arr` de longueur $n$ contenant des entiers.

**1.** Proposer un algorithme de complexité linéaire modifiant (en place) l'ordre des éléments de la liste de sorte que, pour tout $0 \leqslant i \leqslant n-2$,

- `arr[i] <= arr[i+1]` si $i$ est pair ;
- `arr[i] >= arr[i+1]` si $i$ est impair.

**2.** Proposer un invariant de boucle permettant de justifier sa correction.

**3.** Écrire un programme C réordonnant de la sorte les éléments d'un tableau `arr` de taille `n`.

## Ex. 3.11 – Liste d'entiers

**1.** Proposer un programme remplissant un tableau `arr` de taille $n$ avec les $n$ plus petits entiers de la forme $2^p \times 3^q$ ordonnés par ordre croissant. Quelle est sa complexité ?

**2.** Si ce n'est pas encore le cas, modifier le programme de façon à obtenir une complexité linéaire en temps en la taille $n$ du tableau (on pourra supposer disposer d'un tableau supplémentaire de taille $n$ pour les calculs).

### Ex. 3.12 – Produit maximal

On dispose d'un tableau `arr` de taille `n` contenant des entiers relatifs.

**1.** Soit un entier 0 <`k`⩽ $n$. Proposer un programme, le plus efficace possible, déterminant le plus grand entier qu'il est possible de construire en multipliant `k` éléments distincts de `arr` (on supposera qu'il n'y aura pas de débordement lors des calculs).

**2.** Estimer le nombre d'opérations élémentaires effectuées par le programme.

**3.** Proposer un programme *en temps linéaire* déterminant le plus grand entier qu'il est possible de construire en multipliant un nombre quelconque d'éléments **consécutifs** de `arr`.

### Ex. 3.13 – Permutations et ordre lexicographique

Un tableau `arr` de taille `n` représente une *permutation* de $[\![0\,..\,n-1]\!]$ s'il contient, dans un ordre quelconque, exactement les entiers de 0 à $n-1$.

On définit un ordre $\preceq_L$ dit *ordre lexicographique* sur les permutations de $[\![0\,..\,n-1]\!]$ et les tableaux les représentant défini par :

$$\texttt{t1} \prec_L \texttt{t2} \;\equiv\; \exists j \in [\![0\,..\,n-1]\!] \mid \forall i < j,\ \texttt{t1[i]} = \texttt{t2[i]} \text{ et } \texttt{t1[j]} < \texttt{t2[j]}$$

**1.** Quelle est la plus grande permutation possible pour $\preceq_L$ ?

Pour toute autre permutation que la précédente, on peut vouloir chercher celle qui la suit immédiatement dans l'ordre lexicographique.

**2.** Déterminer les huit permutations suivant immédiatement la permutation ci-dessous :

| arr | 1 | 2 | 5 | 4 | 0 | 3 |
|---|---|---|---|---|---|---|

Pour déterminer la permutation suivante, on propose l'algorithme suivant :
- déterminer le plus grand index $j$ tel que `arr[j]<arr[j+1]`;
- déterminer l'indice $k > j$ tel que `arr[k]>arr[j]` et `arr[k]` soit le plus petit possible;
- échanger le contenu des cases d'index $j$ et $k$;
- renverser le contenu des cases d'index $j+1$ à $n-1$.

**3.** Proposer un programme C appliquant l'algorithme précédent à un tableau `arr` de taille `n` (on pourra supposer que `arr` ne contient pas la plus grande permutation).

**4.** Quelle est sa complexité temporelle dans le pire des cas ?

**5.** Quelle est sa complexité temporelle *en moyenne* (on cherchera à calculer le temps nécessaire à l'algorithme pour déterminer l'ensemble des permutations, et on le rapportera au nombre total de permutations) ?

Indice : chercher à déterminer la probabilité que l'on ait besoin de considérer la case en position $n-k$ pour $k>2$.

### Ex. 3.14 – Minimisation

Proposer un programme qui, à partir d'un tableau `arr` de longueur `n`> 0, détermine (aussi efficacement que possible) un entier $p$ minimisant

$$\sum_{i=0}^{n-1} \left|\texttt{arr[i]} - p\right|^2$$

### Ex. 3.15 – Alignement

On considère deux tableaux `x` et `y` d'entiers, de même longueur `n` non nulle, contenant respectivement les abscisses et ordonnées de points du plan.

**1.** Proposer une fonction « `bool aligned(int x[], int y[], int n)` » renvoyant un booléen indiquant si les $n$ points sont alignés. Celle-ci doit gérer correctement les cas particuliers (points confondus...), et ne pas risquer d'éventuelles erreurs d'arrondi.

**2.** Proposer de même une fonction « `bool cocyclic(int x[], int y[], int n)` » déterminant s'ils sont cocycliques, avec les mêmes précautions que pour la fonction précédente. On pourra s'intéresser au théorème de Ptolémée.

### Ex. 3.16 – Un tri ou pas ?

On propose le programme suivant pour trier un tableau `arr` de taille `n` :

```c
for (int i=0; i<n; ++i) {
    for (int j=0; j<n; ++j) {
        if (arr[i] < arr[j]) {
            int tmp = arr[i];
            arr[i] = arr[j];
            arr[j] = tmp;
        }
    }
}
```

**1.** Essayer l'algorithme sur le tableau suivant :

| arr | 3 | -4 | 1 | -9 | 5 | 2 |
|---|---|---|---|---|---|---|

**2.** Déterminer une propriété sur `arr[i]` à l'*issue* d'une itération d'une boucle sur $i$, et prouver cette propriété.

**3.** Déterminer une propriété sur les éléments du tableau aux positions d'index 0 à $i$ à l'*issue* d'une itération d'une boucle sur $i$, et prouver cette propriété.

**4.** En déduire que l'algorithme proposé constitue bien un tri. Quelle est sa complexité ?

## Ex. 3.17 – Équilibre

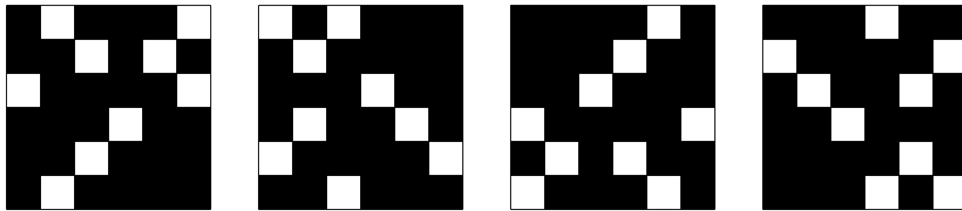
On dispose d'un tableau de booléens `arr` de taille `n`. On cherche à déterminer la longueur $k$ du plus grand sous-tableau ($k$ cases consécutives) contenant le même nombre de `true` et de `false`.

Proposer une fonction `int longuest(bool arr[], int n)` renvoyant $k$ aussi efficacement que possible (il existe une solution linéaire en $n$).

## Ex. 3.18 – Masques

Un « masque » de taille $2n$ est un tableau de booléens à $2n$ lignes et $2n$ colonnes tel que, si l'on considère le tableau et ses trois rotations (de 90, 180 et 270 degrés), chaque case contienne une seule fois **`True`** et trois fois **`False`**.

Par exemple, si les `true` sont représentés par des cases blanches et les `false` par des cases noires, le tableau ci-dessous (et ses quatres orientations) est un masque :



Proposer un programme déterminant, connaissant `n`, si un tableau `arr` de taille $2n \times 2n$ est un masque.

## Ex. 3.19 – Hidoku

Un *hidoku* est un tableau de taille $n \times n$ contenant les entiers de 1 à $n^2$ de sorte que pour tout $i \in [\![1\,..\,n^2-1]\!]$, les cases contenant $i$ et $i+1$ sont voisines (deux cases sont considérées voisines si elles ont un côté en commun).

| | | | |
|---|---|---|---|
| 16 | 3 | 4 | 5 |
| 15 | 2 | 1 | 6 |
| 14 | 11 | 10 | 7 |
| 13 | 12 | 9 | 8 |

On dispose d'un tableau bidimentionnel `arr` d'entiers, de taille $n \times n$. Proposer un programme déterminant si `arr` correspond à un hidoku. On demande une complexité quadratique en $n$ ($O(n^2)$).

## Ex. 3.20 – Problème de Furstenberg et Weiss

On s'intéresse ici à un problème proposé par les mathématiciens israélo-américains Hillel Furstenberg et Benjamin Weiss, qui se place dans le cadre plus général de la théorie de Ramsey. On souhaite savoir s'il existe, dans un tableau rectangulaire contenant des entiers de 1 à $r$, quatre cases, constituant les sommets d'un carré dont les côtés sont horizontaux et verticaux, ayant toutes les quatre le même contenu.

Par exemple, c'est le cas dans le tableau ci-dessous :

| | | | | |
|---|---|---|---|---|
| 1 | 2• | 1 | 3 | 2• |
| 1 | 1 | 2 | 1 | 3 |
| 3 | 3 | 2 | 3 | 1 |
| 1 | 2• | 1 | 3 | 2• |

Proposer un programme déterminant s'il existe un tel « carré » dans un tableau `arr` à deux dimensions (dont on suppose que toutes les cases contiennent bien des entiers dans $[\![1\,..\,r]\!]$), pour lequel le premier index varie de 0 à `nbl`$-1$ et le second de 0 à `nbc`$-1$. On prendra garde aux limites des boucles pour ne pas déborder du tableau tout en testant bien toutes les possibilités.

## Ex. 3.21 – Double bulle

Une variante du tri bulle[63] consiste à faire remonter les bulles deux par deux. Par exemple, pour un tableau tel que celui ci-dessous :

| `arr` | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | 3 | -4 | 1 | -9 | 5 | 2 | -8 | 9 | 6 | -3 | 4 | -6 | 0 | -7 | -1 |

la première passe remonte les *deux* plus petits éléments aux deux premières places avec la méthode classique du tri bulle, ce qui conduit à

| `arr` | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | -9 | -8 | 3 | -4 | 1 | -7 | 5 | 2 | -6 | 9 | 6 | -3 | 4 | -1 | 0 |

puis on s'occupe des éléments en position 2 et 3 :

| `arr` | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | -9 | -8 | -7 | -6 | 3 | -4 | 1 | -3 | 5 | 2 | -1 | 9 | 6 | 0 | 4 |

et ainsi de suite...

Proposer une implémentation de ce tri (on réfléchira particulièrement à la condition d'arrêt de la boucle principale, selon la parité de la longueur du tableau).

63. Qui sous certaines conditions peut présenter des quelques avantages en terme d'efficacité sur de petits tableaux.

# 4 Adresses et pointeurs

« All problems in CS can be solved by another level of indirection, except for the problem of too many layers of indirection. »

— *David Wheeler*

## 1 Adressage

### 1.1 Adresse d'une variable

L'une des spécificités du langage C, et une des raisons de son choix comme langage enseigné en MP2I, est qu'il fournit au programmeur moult possibilités pour intéragir directement avec la mémoire.

Comme évoqué précédemment, les données manipulées par le programme sont [1] stockées dans la mémoire de la machine entre deux calculs, cette mémoire étant constituées de cases permettant généralement de stocker un octet.

Beaucoup de données ne tiennent pas dans un seul octet, et sont mémorisées dans des cases contiguës de la mémoire. Nous avons déjà vu que nous pouvions obtenir la taille réservée en mémoire (nombre de « cases ») pour une variable par le biais de l'opérateur **sizeof**.

Il est également possible de savoir *où* cette réservation débute dans la mémoire, c'est-à-dire l'*adresse* où se situe la variable dans la mémoire (qui correspond, pour faire simple, au « numéro » de la première des cases utilisées), en utilisant l'opérateur &, placé devant le nom de variable dont on souhaite connaître l'adresse. Le programme ci-dessous affiche par exemple l'adresse de la variable mango [2] :

```c
int mango = 42;

printf("L'adresse de la variable mango est %p\n", &mango);
```

1. Pour la plupart au moins.
2. L'instruction printf risque de provoquer un avertissement lors de la compilation liée au type de &mango, nous y reviendrons.

Si nous avons précédemment utilisé « %d » pour indiquer un entier dans une chaîne de formattage printf, c'est « %p » qu'il conviendra d'employer pour obtenir une représentation correcte d'une adresse. Celle-ci sera affichée dans un format hexadécimal, comme c'est l'usage.

Une telle adresse peut être mémorisée dans une variable pour un usage ultérieur, mais il nous faut pour cela disposer d'un type adéquat pour cette variable [3]. Le type utilisé pour une adresse dépend de l'objet situé à l'adresse en question : on n'utilisera pas le même type pour l'adresse d'un entier de type **int** ou l'adresse d'un flottant de type **double**.

Les types correspondant à des adresses se construisent en faisant suivre le type de l'objet rangé à l'adresse en question par le symbole « * ». Par exemple, le type « **int***» permet de mémoriser l'adresse d'un **int**, le type « **double***» l'adresse d'un **double**. On peut donc déclarer une variable p_mango destinée à recevoir l'adresse de la variable mango, et l'initialiser, en écrivant :

```c
int* p_mango = &mango;
```

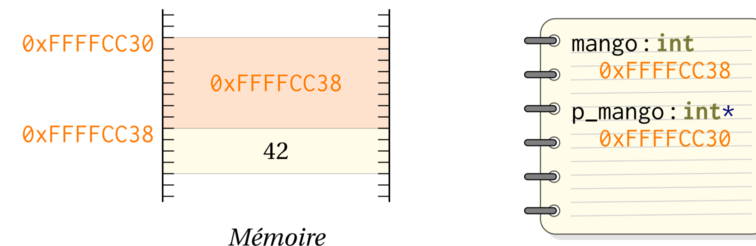
L'adresse de la variable mango, mémorisée dans p_mango, peut être utilisée ultérieurement, par exemple en vue de l'afficher avec :

```c
printf("L'adresse de la variable mango est %p\n", p_mango);
```

En langage C, une variable, telle que p_mango, contenant une adresse est appelée *pointeur*. On dira fréquemment que p_mango « *pointe* » vers mango. Cela signifie que ce qui se trouve à l'addresse mémorisée dans p_mango correspond à l'emplacement mémoire réservé pour la variable mango.

Il n'y a aucune règle particulière à respecter pour les noms des variables contenant une adresse (autrement dit les noms de pointeurs). Toutefois, afin d'éviter tout malentendu, nous choisirons, *pour un temps* dans la suite, des noms débutant par « p_ » (suivis généralement par une indication de l'objet pointé).

Dans la mémoire, les choses se passent donc de la manière suivante :

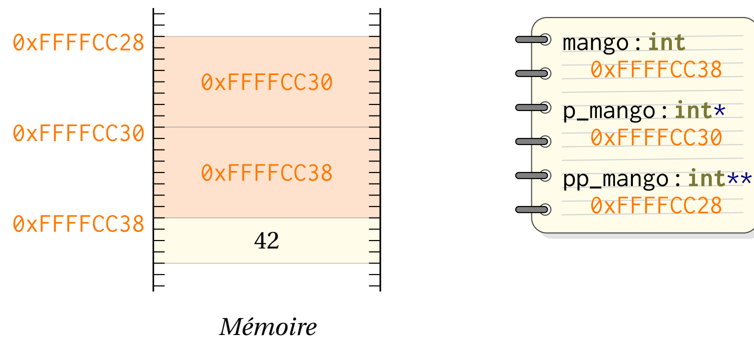



3. Même si une adresse s'apparente techniquement à un entier, il peut y avoir des subtilités et il est préférable d'avoir un type permettant de mémoriser spécifiquement une adresse.

Les pointeurs étant, sous beaucoup d'aspects, des variables comme les autres, il est parfaitement possible d'obtenir l'adresse à laquelle ils se trouvent en mémoire, et si besoin de mémoriser cette adresse. L'adresse de p_mango serait de type int** (c'est en effet l'adresse d'une variable de type int*), et on pourrait donc écrire :

```
int** pp_mango = &p_mango;
```

Ce qui donnerait, en mémoire, quelque chose comme :




*Mémoire*

## 1.2 Utilisation de l'adresse

Ce mécanisme n'a évidemment d'intérêt que si l'on peut, ensuite, *directement* intéragir avec le contenu de la mémoire situé à l'adresse que l'on a mémorisée. Pour ce faire, on dispose de l'opérateur « * » qui, placé à gauche d'une adresse, permet de faire référence à la zone mémoire correspondant à cette adresse. On parle de *déréférencement*.

Ainsi, « *p_mango » désigne le contenu de la mémoire à l'adresse contenue dans la variable p_mango. Pour savoir s'il doit y lire un int, un bool, un double ou un int*, le compilateur se base sur le type de p_mango : puisqu'il s'agit d'une variable de type int*, alors *p_mango sera considéré comme étant de type int, et le compilateur sait ainsi combien de cases contiguës il doit lire et comment interpréter leurs valeurs.

Dans le programme, « *p_mango » et « mango » désignent donc très exactement la même chose : tous deux font référence à la zone mémoire apte à contenir un int désignée par le nom mango et rangée à l'adresse mémorisée dans p_mango. Ainsi, l'instruction

```
*p_mango = *p_mango + 1;
```

est parfaitement équivalente (elle incrémentate la valeur contenue dans la mémoire à l'adresse contenue dans p_mango) à l'instruction

```
mango = mango + 1;
```

À quoi cela peut-il nous servir? Un des principaux usages des pointeurs est de permettre de lever les limitations que le mécanisme de passage des paramètres (par valeur) dans une fonction nous impose.

Supposons par exemple que l'on souhaite une fonction effectuant une permutation entre deux variables. On pourrait être tenté d'écrire :

```
void swap(int a, int b) { // Attention : Fonction inutile !
    int tmp = a;
    a = b;
    b = tmp;
}
```

Malheureusement, cette fonction bien que formellement correcte, ne sert rigoureusement à rien dans la pratique. En effet, si l'on tentait de s'en servir de la sorte

```
int mango = 37, cherry = 42;

swap(mango, cherry); // <- Instruction sans effet
// mango contient toujours 37, et cherry toujours 42
```

l'appel à la fonction swap n'aurait strictement aucun effet sur les variables mango et cherry. Ce qui serait échangé, ce sont les contenus des variables a et b *locales* à la fonction swap, ce qui est d'un intérêt plus que limité...

En revanche, si l'on définit une fonction swap de la sorte :

```
void swap(int* p_a, int* p_b) {
    int tmp = *p_a;
    *p_a = *p_b;
    *p_b = tmp;
}
```

et que l'on s'en sert de la sorte :

```
int mango = 37, cherry = 42;

swap(&mango, &cherry);
// mango contient désormais 42, et cherry 37
```

alors cette fois ci, les contenus des deux variables sont bien échangés. En effet, les variables locales p_a et p_b de la fonction swap ont été initialisées avec les adresses des variables mango et cherry. Ensuite, la fonction swap n'agit pas sur le contenu de ces variables, mais sur ce qui se trouve, en mémoire, aux adresses mémorisées dans p_a et p_b. Elle agit donc bien directemet sur le contenu des variables mango et cherry.

## 1.3 Retour sur les déclarations

Lorsque l'on déclare un pointeur, on dispose d'un peu de souplesse dans l'écriture. On peut aussi bien écrire

```
int* p_mango = &mango;
```

qu'écrire

```
int *p_mango = &mango;
```

On peut interpréter la première écriture comme le fait que `p_mango` est une variable de type `int*`, soit un pointeur vers un `int`, et interpréter la seconde comme le fait que ce que pointe `p_mango`, autrement dit `*p_mango` est un `int`, ce qui revient exactement au même.

La première écriture peut paraître un peu plus logique, en particulier lorsqu'il y a une initialisation en même temps que la déclaration, comme ci-dessus : c'est bien la variable `p_mango` (de type `int*`) qui est initialisée dans les *deux* cas, avec l'adresse de `mango`.

Malheureusement, lorsque le compilateur C interprète les déclarations, il considère que le « `*` » colle à l'objet à sa droite, en ne tenant pas compte des espacements. Cela peut jouer de mauvais tours si l'on effectue des déclarations multiples. Ainsi, si l'on écrit :

```
int* p_mango, cherry;
```

le compilateur interprète la déclaration comme

```
int *p_mango, cherry;
```

ce qui se traduit donc en

```
int* p_mango;
int cherry;
```

Autrement dit, `p_mango` est une variable de type `int*`, mais `cherry` est une variable de type `int`[4] ! Donc, lorsque l'on rencontrera des initialisations multiples, nous préférerons souvent placer le « `*` » de l'autre côté pour éviter tout malentendu et écrire[5] :

```
int *p_mango, *p_cherry;
```

4. Rappelons que la présence de « `p_` » n'est qu'une convention d'écriture, on aurait eu le même résultat évidemment en écrivant « `p_cherry` », mais c'eût été encore plus délicat à comprendre.

5. Il n'y a simplement pas de « bonne » façon de trancher ce débat, on place traditionnellement plutôt le symbole « `*` » du côté droit en C, mais du côté gauche en C++, et c'est aussi ce que suggère le programme de MP2I. C'est accessoirement un argument pour simplement éviter les déclarations multiples !

## 1.4 Validité des adresses

Il existe de nombreuses restrictions à l'usage de l'opérateur « `*` ». La principale étant qu'il n'est possible de déréférencer que des adresses qui correspondent effectivement à une adresse « correcte » (qui, d'une façon ou d'une autre, a été réservée par le programme et n'a pas encore été libérée, et contient une donnée d'un type compatible avec le type de pointeur utilisé). Accéder au contenu d'une adresse que l'on n'a pas obtenue par des moyens « licites » (par exemple via l'opérateur `&`) est un cas de comportement indéfini.

On ne peut en particulier absolument pas espérer étudier le contenu de la mémoire de l'ordinateur en affichant le contenu d'adresses que l'on a librement choisies. D'autant que les adresses manipulées par le programme ne correspondent pas à des adresses réelles dans la mémoire : si l'on fait tourner deux programmes en même temps, il est fort possible qu'ils se retrouvent à utiliser des adresses identiques.

En fait, chaque programme travaille avec sa propre mémoire virtuelle, et les adresses vues par un programme sont traduites en adresses « physiques » par le système d'exploitation et le processeur[6]. En plus de la souplesse que ce mécanisme procure, c'est très utile pour des questions de sécurité : un programme ne doit pouvoir accéder qu'à ses propres données et certainement pas aux données d'autres programmes tournant simultanément sur la machine, même si le compilateur décidait de permettre au programme la lecture d'adresses aléatoires.

Il convient tout particulièrement de prendre garde au fait qu'une variable de type pointeur a pu conserver l'adresse d'une variable qui n'existe plus, par exemple parce qu'elle a atteint la limite de sa portée, comme dans l'exemple ci-dessous :

```
int* ptr;
{
    int mango=42;
    ptr = &mango;
}
int cherry = *ptr; // <- Illégal (UB) car mango n'existe plus
```

Une telle situation conduit à un comportement indéfini du programme[7], et c'est une des raisons pour lesquelles les pointeurs sont une source très fréquente de bugs dans les programmes C (et une des raisons à la fois de la mauvaise réputation des pointeurs et du fait que la quasi-totalité des langages essaient de s'en passer, ou de proposer des alternatives plus sûres).

6. Ces adresses peuvent d'ailleurs changer au cours du temps, le système d'exploitation pouvant par exemple enregistrer une partie de ces informations dans un fichier d'échange (swap) sur le disque dur lorsqu'elles sont peu utilisées afin de libérer la mémoire pour d'autres usages, et les restaurer plus tard.

7. Dans l'exemple précédent, il est fort possible que, même si `mango` n'est théoriquement plus accessible, son contenu n'ait pas encore été effacé de la mémoire, et que le comportement (indéfini) du programme consiste à mettre dans `cherry` ce qui subsiste à cet emplacement mémoire. En d'autres termes, il y a toutes les chances que cela « marche » encore si le but était de récupérer la valeur de `mango`, mais cela reste illégal.

## 1.5 Pointeurs NULL

Dans un algorithme, il peut arriver que l'on ait besoin, à certains moments, qu'une variable de type pointeur ne désigne aucun élément en mémoire. Pour permettre ceci, le langage C garantit que l'adresse désignée par 0 n'est *jamais* utilisée pour mémoriser quelque chose. Il est donc d'usage d'assigner cette valeur 0 à une variable de type pointeur pour signifier qu'elle ne pointe vers aucune donnée.

Bien entendu, tenter de déréférencer un tel pointeur pour en faire quoi que ce soit conduit à un comportement indéfini du programme, puisque la mémoire située à l'adresse désignée par 0 n'est accessible ni en lecture, ni en écriture.

0 se trouve ainsi être la seule et unique valeur que l'on puisse directement assigner à une variable de type pointeur. Afin d'apporter un peu plus de clarté au code source, on peut écrire [8] « `NULL` » à la place de 0, `NULL` n'étant simplement qu'une façon alternative d'écrire 0 dans un programme lorsque cela est en lien avec des pointeurs.

Si l'on ne connaît pas, lors de la déclaration d'un pointeur, l'adresse d'un objet vers lequel il doit pointer, pour ne pas laisser le pointeur indéfini, il est fréquent de l'initialiser à `NULL` :

```
int* ptr = NULL;
```

Afin de savoir si deux pointeurs `ptr1` et `ptr2` pointent vers le même objet, on peut comparer les adresses qu'ils contiennent avec `==` (ou `!=`). Le résultat de la comparaison est « `true` » si et seulement si les objets pointés sont les mêmes. Il est également possible de comparer une variable de type pointeur avec la valeur particulière `NULL` (ou la valeur 0), ce qui nous offre un moyen pour tester si une variable pointe bien vers un objet, sous réserve que l'on ait utilisé la valeur `NULL` dans ce but [9] « `if (ptr != NULL)` ».

Signalons que si un pointeur est utilisé seul comme expression dans un test, le compilateur interprétera ça comme un test le confrontant à `NULL` : « `if (ptr)` » est équivalent à « `if (ptr != NULL)` » [10]. De manière générale, nous éviterons dans ce cours, en accord avec le programme, de laisser au compilateur la charge de donner une interprétation booléenne d'une expression qui ne l'est pas [11], mais il s'agit d'un cas d'usage suffisamment courant pour qu'il nous paraisse utile d'en faire mention.

8. Afin que cette constante soit bien définie, il est nécessaire d'inclure un fichier d'en-tête qui s'en charge, typiquement `stddef.h`. Mais l'inclusion d'un fichier d'en-tête tel que `stdio.h`, `stdlib.h` ou bien encore `string.h` suffit à s'assurer que `NULL` soit défini.

9. En effet, la seule chose qui est contrôlée est si l'adresse rangée dans la variable de type pointeur est « non nulle ». Il n'existe aucun moyen de contrôler que cette adresse désigne bien un contenu valide et encore accessible !

10. De même, « `if (!ptr)` » est équivalent à « `if (ptr == NULL)` ».

11. De manière générale, est considéré « vrai » ce qui est « non nul ».

## 1.6 Pointeurs et mémoire adressable

Comme pour n'importe quelle variable, on peut se renseigner sur la taille occupée par un pointeur en mémoire avec l'opérateur `sizeof`, le résultat étant généralement 4 ou 8 bytes [12]. Sans doute avez-vous déjà remarqué que de nombreux programmes ou systèmes d'exploitation sont qualifiés de « 32 bits » ou « 64 bits ». Ces termes font justement référence à la taille des pointeurs en mémoire. Cette taille a des conséquences très concrètes. En effet, un pointeur dont la taille est « 32 bits » ne permet de référencer, au mieux, que $2^{32}$ « cases » mémoire, soit un peu plus de quatre milliards de bytes.

La mémoire vive disponible sur les ordinateurs n'ayant cessé de croître, elle dépasse à présent très fréquemment les quatre gigaoctets. Un programme utilisant des pointeurs de taille 32 bits ne peut donc plus utiliser l'intégralité de la mémoire disponible [13]. Il devient donc plus qu'utile de passer à des pointeurs sur 64 bits, même s'ils occupent un peu plus de place en mémoire [15].

## 1.7 Allocation dynamique

À travers les pointeurs, les programmeurs ont accès à de nouvelles possibilités dans la gestion de la mémoire. Nous avons vu que la déclaration d'une variable réserve de la place en mémoire pour son stockage, et que cette place en mémoire est libérée lorsque la variable atteint la limite de sa portée. Mais il est également possible de réserver explicitement de la place en mémoire, en utilisant la fonction `malloc` (**m**emory **alloc**ation) qui prend en argument la taille (en bytes, donc en nombre de cases mémoires) de la zone à réserver, et renvoie l'adresse du début de la zone réservée.

C'est à cette occasion que `sizeof` est particulièrement utile, afin de connaître la taille mémoire qu'il est nécessaire de réserver pour stocker un objet d'un type donné. Par exemple, « `malloc(sizeof(int))` » réserve un emplacement mémoire dont la taille permettra de ranger exactement un objet de type `int`, et renvoie l'adresse de la première case de la zone mémoire ainsi réservée.

Une difficulté apparaît ici : comme la fonction peut réserver une zone mémoire destinée au stockage de différents objets, elle ne sait si elle doit renvoyer un `int*`, un `double*`… Elle renvoie donc un objet de type particulier, un `void*`. C'est le type utilisé pour une adresse permettant de ranger un objet dont le type n'est pas connu du compilateur.

12. Le programme de MP2I ne considère que des pointeurs sur 8 bytes.

13. Si plusieurs programmes s'exécutent en même temps sur la machine, ils peuvent cependant chacun utiliser des parties différentes de la mémoire [14], et donc *conjointement* utiliser plus de quatre gigaoctets de mémoire vive. Rappellons en effet que les adresses vues par un programme ne correspondent pas directement aux adresses physiques de la mémoire, le processeur effectuant une traduction des adresses.

15. S'il peut sembler étrange que deux tailles différentes de pointeurs peuvent coexister sur une même architecture, dans la mesure où cela complique le design du processeur, c'était un mal nécessaire pour préserver la compatibilité des logiciel existants. La disparition de la compatibilité des nouvelles machines avec les programmes 32 bits se pose, et certains systèmes d'exploitation ont décidé de franchir le pas et ne supportent plus que des programmes dits « 64 bits ».

Avant toute utilisation de cette adresse, il faut la convertir au bon type, afin que lors du déréférencement il soit possible de savoir quel objet il faut lire en mémoire. Cela peut se faire en utilisant une conversion de type sur le résultat de l'appel à `malloc`, et en écrivant par exemple « `(int*)malloc(sizeof(int))` ». On peut donc réserver de la mémoire destinée à conserver un entier, et mémoriser l'adresse correspondante dans une variable de type pointeur, en écrivant :

```
int* ptr = (int*)malloc(sizeof(int));
```

La conversion explicite de type est cependant inutile ici, on pourrait simplement écrire :

```
int* ptr = malloc(sizeof(int));
```

En effet, la conversion aura de toute façon lieu lorsque l'on rangera le résultat du malloc dans la variable `ptr` de type `int*`. En fait, on recommande en général, en langage C, d'utiliser cette seconde écriture [16]

Une des raisons poussant à préférer la seconde écriture, implicite, est que si l'on souhaite modifier le type pointé, cela nécessite moins d'altérations du programme (on aurait en effet tôt fait de modifier la déclaration de type de `ptr` et d'oublier de reporter ce changement dans le `malloc`). En fait, pour éviter toute répétition, on préfère même en général indiquer dans le `sizeof` que la taille requise est celle de l'objet pointé, en écrivant :

```
int* ptr = malloc(sizeof(*ptr));
```

Dans cette dernière écriture, il n'y a plus qu'une unique mention de `int`, qui peut donc être modifiée rapidement sans aucun risque. On perd naturellement cette facilité avec une conversion explicite du type du pointeur.

Il reste un point à préciser : il est possible que la réservation de la mémoire se passe mal, et que `malloc` ne parvienne pas à réserver la zone mémoire demandée [17]. Dans ce cas, `malloc` renvoie la valeur `NULL`. Il faudrait en théorie [18] systématiquement tester le résultat de l'appel à `malloc` et de vérifier que l'adresse obtenue n'est pas `NULL` avant de s'en servir !

En demandant la réservation d'une zone de la mémoire avec `malloc`, le programme s'en voit confier la responsabilité. Cette zone mémoire lui est réservée jusqu'à la fin du programme, ou bien jusqu'à ce qu'il décide de la libérer, n'en ayant plus besoin. La libération d'une zone mémoire obtenue avec `malloc` se fait avec la fonction `free`, qui prend en argument une adresse ayant été renvoyée lors d'un précédent appel à `malloc` [19] :

```
int* ptr = malloc(sizeof(int)); // Allocation

if (ptr != NULL) { // Si l'allocation s'est bien passée :
    ...            // Utilisation de la mémoire allouée
    free(ptr);     // Libération de la mémoire allouée
}
```

En toure rigueur, chaque réservation de mémoire nécessite une libération explicite lorsque l'on n'en a plus besoin [20]. Seulement, les variables utilisées pour mémoriser les adresses des zones réservées ayant une portée bien précise, il est parfaitement possible de « perdre » l'adresse d'une zone mémoire réservée, auquel cas il devient impossible de libérer la mémoire réservée. C'est une situation qualifiée de « *fuite mémoire* ». Si cela arrive régulièrement, le programme occupera de plus en plus de place en mémoire même s'il ne peut utiliser des zones entières de cette mémoire, en ayant perdu les clés, au point parfois de conduire l'ordinateur à ne plus avoir de mémoire vive disponible. Il faut donc être particulièrement prudent avec ces réservations [21].

Comme `malloc` et `free`, la fonction `printf` attend que les pointeurs qu'on lui soumet soient de type « `void*` ». Si l'on veut afficher l'adresse d'un pointeur avec « `%p` », il faut en principe explicitement convertir ce pointeur en `void*` (ce que `printf`, ne connaissant pas le type de ses arguments en avance, ne sait faire spontanément) en écrivant :

```
printf("L'adresse contenue dans ptr est %p\n", (void*)ptr);
```

Profitons de l'occasion pour signaler que toutes les conversions d'un type de pointeur à un autre ne sont pas permises. Il est toujours possible de convertir un pointeur quelconque en `void*`, mais l'inverse n'est pas forcément vrai : si un pointeur a été initialement obtenu avec l'opérateur `&` par exemple, et est destiné initialement à désigner des entiers (pointeur `int*`), il est illégal [22] de le convertir en un pointeur de type `double*`, même en passant par la case `void*`. Nous verrons ultérieurement que la même restriction s'applique au type `char*` : tout pointeur peut être converti en `char*`, mais la conversion inverse n'est possible que si elle respecte le type original du pointeur.

---

16. Tandis que le langage C++ préfère généralement que la conversion de type soit explicite.
17. Lorsqu'il s'agit de quelques bytes, c'est extrêmement peu probable, mais nous verrons tantôt que nous pouvons demander la réservation de centaines de millions de bytes lors d'un seul appel à `malloc`, et dans ce cas, il est beaucoup moins évitent que la réservation soit toujours possible.
18. Toutefois, si une allocation échoue pour un objet de taille raisonnable, c'est que le système est à court de mémoire, et il est peu probable qu'il reste grand-chose à sauver, aussi est-il parfois suggéré d'ignorer cette vérification dans certains cas.
19. Pour une raison similaire à ce qui se passe pour le retour de la fonction `malloc`, la fonction `free` attend en principe un pointeur de type `void*`, mais il n'est pas nécessaire d'effectuer explicitement la conversion : tout pointeur placé en argument sera automatiquement converti en pointeur de type `void*`.
20. Excepté si cette libération doit avoir lieu en fin de programme, où l'appel à `free` n'est pas indispensable car la libération de la mémoire allouée dynamiquement est alors automatique.
21. Et là encore, beaucoup de langages ont cherché des solutions pour éviter ces dangers.
22. Les raisons sont diverses, mais l'une d'entre elles est liée à un souci d'*alignement* des données en mémoire. Le processeur n'est pas forcément capable de lire un `int` à n'importe quel emplacement de la mémoire, il peut par exemple avoir besoin que l'adresse soit divisible par 4. Une conversion entre des pointeurs de type différent pourrait casser ces règles d'alignement.

# 2 Tableaux

## 2.1 Allocation dynamique de tableaux

De la même façon que l'on peut allouer de la place en mémoire pour *une* donnée quelconque, il est possible de réserver de la place pour en ranger *plusieurs*, sur le principe des tableaux (les données étant rangées en mémoire les unes derrière les autres).

Il suffit pour cela, très simplement, de déterminer la taille nécessaire au stockage de l'ensemble de ces données. Ainsi, pour réserver de la place en mémoire pour un tableau de dix entiers, on écrira :

```
int* p_arr = malloc(10*sizeof(int));
```

On parle d'allocation *dynamique* de tableau car la réservation de la mémoire se fait lors de l'exécution, au moment précis où le déroulement du programme atteint l'appel à la fonction `malloc`. Cela présente de nombreux avantages, par exemple de pouvoir créer un tableau dont la taille n'est pas connue lors de la compilation.

Après l'allocation, « `*p_arr` » fera référence à la première des cases du tableau. Mais il nous faut pouvoir accéder également aux autres cases ! Fort heureusement, ce n'est pas bien compliqué car on peut utiliser la même syntaxe que pour les tableaux [23]. Ainsi, pour accéder à la dernière case de notre tableau, on pourra simplement utiliser « `p_arr[9]` ». Pour accéder à la première case, on utilisera généralement de même « `p_arr[0]` » et non `*p_arr` même s'ils sont équivalents. Cela permet de lire [24] et d'écrire dans chacune des « cases » de la zone mémoire allouée dynamiquement :

```
int tmp = p_arr[0];
p_arr[1] = tmp;
```

Lors de l'utilisation, la différence entre un tableau « usuel » et ce que l'on qualifiera de « tableau alloué dynamiquement » n'est pas immédiatement visible, et on oubliera généralement vite que l'on manipule des pointeurs. Par conséquent, nous omettrons dorénavant le préfixe « `p_` » dans les noms de ces objets, et écrirons simplement :

```
int* arr = malloc(10*sizeof(int));
```

23. Nous verrons que ce n'est pas un hasard.
24. Contrairement à ce qui se passe pour une variable, une partie de la zone mémoire allouée par un appel à `malloc` peut être éventuellement être lue avant d'avoir fait l'objet d'une écriture. Cela n'implique pas nécessairement de comportement indéfini du programme, excepté lorsque certaines représentations binaires ne correspondent pas à une valeur possible pour le type concerné (cela peut par exemple arriver pour un booléen). Toutefois, la valeur qui serait lue serait *indéfinie*, ce qui signifie en C que sa valeur peut être quelconque, et peut même changer entre deux lectures. Toute expression faisant intervenir une valeur indéfinie sera par ailleurs également indéfinie. On prendra donc garde, comme pour les variables usuelles, à ne pas accéder à des zones mémoires qui n'ont pas été initialisées.

Puisque « `*arr` » pointe vers l'entier rangé dans la première case du tableau, l'autre écriture pour la déclaration est donc ici aussi valable :

```
int* arr = malloc(10*sizeof(*arr));
```

La libération de la mémoire se fera, avec les mêmes précautions que précédemment, via un appel à la fonction `free`, en écrivant donc [25] :

```
free(arr); // Libération de la mémoire allouée
```

## 2.2 Différences avec les tableaux

Si en apparence le fonctionnement des tableaux usuels et des tableaux dynamiques est similaire, il y a néanmoins des différences majeures liées au fait que dans le second cas, le nom désigne un pointeur.

Ainsi, pour un tableau alloué dynamiquement, « `sizeof(arr)|` » (ou « `sizeof arr` ») renvoie la taille d'un « `int*` » en mémoire, soit la taille occupée par un pointeur. Il n'est *pas possible* d'utiliser `sizeof` pour obtenir des informations sur la taille de la zone mémoire qui a été réservée, donc sur la taille du tableau. Il faut donc toujours mémoriser à part la taille du tableau, par exemple dans une seconde variable.

En outre, dans le cas d'un tableau alloué dynamiquement, `arr` contient une adresse mémoire qui peut très bien être modifiée par une allocation. Ainsi, on peut théoriquement écrire :

```
int* arr1 = malloc(2*sizeof(*arr1));
int* arr2 = malloc(2*sizeof(*arr2));

arr1 = arr2;
```

Après cette affectation, `arr1` et `arr2` font référence à la même zone mémoire (celle initialement réservée lors de l'initialisation de `arr2`. En particulier, « `arr1[i]` » et « `arr2[i]` » désignent le même emplacement mémoire. Dorénavant, modifier le contenu d'un tableau modifie également le contenu de l'autre tableau [26].

Plus gênant, on a ici perdu tout accès à la zone mémoire réservée lors de l'initialisation de `arr1`. Il n'est plus possible d'y accéder, ni même de la libérer. Nous avons ici un problème de fuite de mémoire.

Une telle affectation est impossible si `arr1` est un tableau usuel.

25. Précisons qu'il est impossible de libérer une *partie* de la mémoire allouée avec un appel à `malloc`.
26. Il y a ici un comportement assez similaire avec le cas d'une affectation « `L1 = L2` » écrit en Python lorsque `L1` et `L2` sont des listes. Ce qui peut aisément s'expliquer par la façon dont Python gère pratiquement ses objets en mémoire.

## 2.3 Tableaux et fonctions

Nous l'avons dit précédemment, il est impossible de passer un tableau en argument à une fonction. En revanche, il est tout à fait possible d'utiliser un pointeur. Par exemple, on peut écrire une fonction prenant en argument un pointeur arr vers un tableau d'entiers de taille n, ainsi que sa taille [27] et renvoyant le plus grand de ses élément :

```
int max_array(int* arr, int n) {
    // renvoie le plus grand élément du tableau arr de taille n>0
    int maximum = arr[0];
    for (int i=1; i<n; ++i) {
        if (arr[i] > maximum) {
            maximum = arr[i];
        }
    }
    return maximum;
}
```

Pour utiliser cette fonction sur un tableau arr alloué dynamiquement dont on connaît la taille n, on peut simplement écrire :

```
    int maxi = max_array(arr, n);
```

Comment faire avec un tableau usuel, déclaré avec « int arr[5] »? Une solution est d'obtenir l'adresse du début du tableau avec l'opérateur « & » et d'écrire

```
    int arr[5] = { 11, 37, 22, 54, 42 };

    int maxi = max_array(&arr[0], 5);
```

Mais dans le but de simplifier la tâche des utilisateurs, puisqu'il n'est virtuellement pas possible de faire quelque calcul que ce soit avec un tableau, dans quasiment toutes les situations [28], **lorsque l'on écrit le nom d'un tableau usuel dans un programme, il est traduit en l'adresse de son premier élément.** Ainsi, on peut parfaitement écrire, et cela revient strictement au même :

```
    int arr[5] = { 11, 37, 22, 54, 42 };

    int maxi = max_array(arr, 5);
```

Comme pour l'indexation, il n'apparaît aucune différence notable à l'usage, lors d'un appel de fonction, entre un tableau usuel et un tableau dynamique.

27. Très important puisque cette taille ne peut pas être obtenue autrement.
28. Avec quelques exceptions, par exemple dans le cas d'un **sizeof**.

Reste un dernier point gênant. Lorsque l'on écrit une fonction prenant en argument un tableau et que l'on utilise cette notation :

```
int max_array(int* arr, int n) {
    ...
}
```

il n'est pas évident (surtout si l'on fait abstraction du nom) que arr est un pointeur vers un tableau et non un pointeur vers un simple entier. Si cela n'a pas d'importance pour le compilateur (pour qui un entier, finalement, n'est pas très différent [29] d'un tableau d'entiers de taille 1), cela en a pour celui qui essaiera de lire le programme.

On dispose donc d'une autre écriture, parfaitement équivalente mais qui peut, pour un lecteur du programme, être plus facile à comprendre [30], pour indiquer que le pointeur arr fait référence à un tableau d'entiers :

```
int max_array(int arr[], int n) {
    ...
}
```

Même si cette seconde écriture ressemble à la déclaration d'un tableau usuel, ce n'est qu'une notation facilitant la compréhension. arr reste une variable locale de type « int* ». Il n'est, répétons-le, pas possible d'utiliser un tableau comme argument d'un fonction, on ne peut transmettre que son adresse.

## 2.4 Effets de bords des fonctions

Lors d'un appel à une fonction, les arguments de la fonction sont copiés dans des variables locales. Nous avons déjà vu que la modification de ces variables locales, tout naturellement, pas d'effet sur les variables définies à l'extérieur de la fonction.

Il existait cependant un cas où des effets peuvent être visibles sur les variables existant en-dehors de la fonction : lorsque ces arguments sont des pointeurs, contenant l'adresse de ces mêmes variables situées à l'extérieur de la fonction, et que l'on déréférence ces pointeurs pour modifier la mémoire située à l'adresse pointée (comme c'était le cas de notre fonction swap).

Dans le cas où un ou plusieurs des paramètres d'une fonction est l'adresse d'un tableau, il se passe la même chose : l'*adresse* du tableau est copiée dans une variable locale arr. Si cette variable est modifiée, cela n'a pas d'influence à l'extérieur de la fonction. En revanche, si l'on modifie le contenu d'une *case* du tableau (ce qui se trouve à une adresse mémoire

29. Et oui, si p_mango est un pointeur vers un entier, on *pourrait* utiliser « p_mango[0] » à la place de « *p_mango » pour faire référence à l'entier pointé, mais en programmeur responsable, on ne le fera pas !
30. On parle de *sucre syntaxique* pour désigner de telles notations fournies par le langage visant à faciliter l'écriture et la lecture de programme.

construite à partir de celle du tableau), alors cette modification affecte bien le tableau extérieur à la fonction.

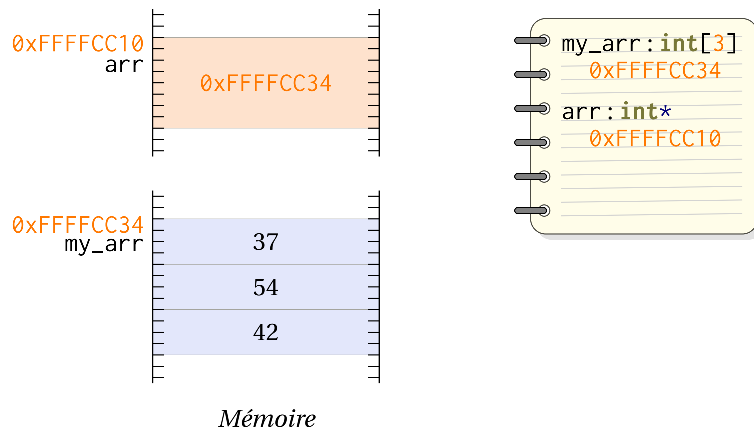
Afin d'illustrer cet « effet de bord » lorsque l'on utilise l'adresse d'un tableau comme paramètre à une fonction, considérons un exemple concret :

```
void cumsum(int arr[], int n) {
    for (int i=1; i<n; ++i) {
        arr[i] += arr[i-1];
    }
}
```

Si l'on appelle cette fonction avec pour argument un tableau [31] et sa taille, en écrivant

```
    int my_arr[3] = { 37, 54, 42 };

    cumsum(my_arr, 3);
```

alors le contenu du tableau `my_arr` est bien modifié par la fonction ! Car si `arr` est une variable locale à la fonction `cumsum`, il ne s'agit d'un pointeur vers le tableau `my_arr`, comme on le voit sur la représentation de la mémoire ci-dessous. Par conséquent `arr[i]` fait directement référence à la case d'index `i` du tableau `my_arr`, qui lui n'est pas local à la fonction. Et toute mention de `arr[i]` fait donc directement référence à `my_arr[i]`.




*Mémoire*

Après l'appel « `cumsum(my_arr, 3);` », le tableau `my_arr` contiendra donc les valeurs 37, 91 et 133, la fonction `cumsum` a donc bien eu un effet sur le tableau `my_arr` dont l'adresse a été passé en argument lors de l'appel à `cumsum`.

31. Il s'agit ici un tableau usuel, mais les effets de bord seraient les mêmes avec un tableau qui aurait été alloué dynamiquement.

Supposons à présent que l'on souhaite échanger le contenu de deux cases d'index `i` et `j` d'un tableau `arr`. Une solution pourrait être d'utiliser la fonction `swap` que l'on a défini un peu plus tôt dans ce chapitre, en utilisant des pointeurs vers les deux cases en question :

```
    swap(&arr[i], &arr[j]);
```

Mais on peut préférer écrire une fonction explicite pour deux cases d'un même tableau d'entiers, par exemple :

```
void array_swap(int arr[], int i, int j) {
    int tmp = arr[i];
    arr[i] = arr[j];
    arr[j] = tmp;
}
```

On pourra donc échanger le contenu des deux cases en écrivant [32] :

```
    array_swap(arr, i, j);
```

## 2.5 Tableaux et retours de fonctions

Une autre limitation évoquée précédemment pour les fonctions concernait l'impossibilité, pour une fonction, de renvoyer un tableau. Toutefois, rien n'empêche une fonction de renvoyer un *pointeur* vers un tableau. Ainsi, il est tout à fait possible d'écrire une fonction renvoyant un tableau d'entiers de taille `len` initialisé avec `val` dans toutes les cases :

```
int* gen_array(int len, int val) {
    int* arr = malloc(len*sizeof(*arr));
    if (arr != NULL) {
        for (int i=0; i<len; ++i) {
            arr[i] = val;
        }
    }
    return arr;
}
```

On utiliserait ensuite cette fonction de la sorte :

```
    int* my_arr = gen_array(10, 42); // Tableau de taille 10
```

32. Notons qu'exceptionnellement, il n'est pas utile ici de mentionner la taille du tableau parmi les paramètres de la fonction car elle n'intervient pas lors de l'échange. On fera juste attention que les paramètres `i` et `j` correspondent bien à deux index valides du tableau.

Cela appelle plusieurs remarques toutefois :

- le résultat ne peut être qu'un tableau dynamique, et donc ne peut être rangé que dans un pointeur, il n'est pas possible de ranger le résultat dans un tableau usuel ;
- la fonction a appelé `malloc`, et a transmis le résultat à l'appelant, c'est donc à ce dernier qu'est confiée la responsabilité de la mémoire allouée, et notamment de sa libération avec un appel à `free` lorsqu'elle ne sera plus utile ;
- le `malloc` pouvant échouer (auquel cas la fonction `gen_array` renvoie `NULL`), l'appelant devrait en principe vérifier que le pointeur qu'il a obtenu n'est pas `NULL` avant d'utiliser ce qu'il pense être un tableau d'entiers.

Attention, techniquement, il ne serait pas illégal d'écrire :

```
int* bad_idea(void) {
    int arr[5] = { 11, 37, 22, 54, 42 };
    return arr; // A NE PAS FAIRE !
}
```

En effet, la fonction crée un tableau usuel, et renvoie un pointeur vers ce tableau (dans le return, `arr` fait référence à l'adresse de la première case). Le hic, c'est que la fin de la fonction sonne le glas de la variable locale `arr`. Autrement dit, on se retrouve avec un pointeur... qui pointe vers des données qui n'existent plus ! Toute tentative de déréférencer le pointeur renvoyé par la fonction provoque donc un comportement indéfini [33].

## 2.6 Tableaux et opérations sur les pointeurs

Comme pour tout pointeur, il est possible de comparer des pointeurs pointant vers des éléments d'un tableau avec `==` et `!=` afin de savoir s'ils désignent la même case ou non. Mais il est également possible de comparer deux pointeurs pointant vers des cases du même tableau [34] avec les opérateurs de comparaison `<`, `<=`, `>` et `>=`, le résultat faisant référence à leur position relative dans le tableau. Par exemple, après les déclarations ci-dessous où `arr` désigne un tableau [35] d'entiers de taille au moins égale à 6 :

```
    int* ptr1 = &arr[2];
    int* ptr2 = &arr[5];
```

si l'on effectue la comparaison « `ptr1 <= ptr2` », on obtiendra le résultat `true` car `arr[2]` est situé avant `arr[5]` dans le tableau.

33. Même si la mémoire a été marquée comme libérée, il est possible que le tableau subsiste encore dans la mémoire, et comme le comportement est indéfini, le compilateur n'est pas tenu de provoquer une erreur si l'on accède à une adresse qui n'est plus valide. Il est donc parfaitement courant d'avoir l'*impression* que cela fonctionne, et que l'on puisse accéder, du moins dans un temps bref suivant le retour de la fonction, aux données du tableau `arr` depuis l'extérieur de la fonction. Le programme n'en est pas moins rigoureusement incorrect !

34. Si on tente de comparer deux pointeurs pointant vers des éléments ne provenant pas d'un même tableau, le comportement du programme est indéfini.

35. Tableau usuel ou tableau dynamique.

Il existe également un certain nombre d'opérations sur les pointeurs regroupées sous l'appellation d'*arithmétique pointeur*. Ces opérations sont explicitement hors programme, nous éviterons donc de nous en servir dans la suite. Ce qui suit ne vise donc qu'à fournir un éclairage rapide sur des choses que vous pourriez voir fréquemment dans des ouvrages ou des ressources en ligne.

Il est ainsi possible, tout d'abord, de soustraire deux pointeurs pointant vers des éléments d'un même tableau (statique ou dynamique), le résultat indiquant la différence d'index entre les deux « cases ». Par exemple, suite aux déclarations précédentes, « `ptr2 - ptr1` » renverra 3 (car la différence des index est 5 – 2). On remarquera donc que la soustraction de deux pointeurs ne soustrait pas directement les adresses (il y aurait, pour des entiers stockés sur 4 bytes, une différence de 12 cases entre les adresses contenues dans les deux pointeurs) mais tient compte de la taille occupée par les éléments.

Il n'est en revanche pas possible d'additionner deux pointeurs, car cela n'aurait aucun sens. En revanche, on peut additionner (ou soustraire) un pointeur et un entier $k$, le résultat étant un pointeur correspondant à la case située $k$ emplacements plus loin (ou plus en amont) dans le tableau que celle désignée par le pointeur. Ainsi, l'expression « `ptr1 + 2` » a pour résultat un pointeur de type `int*` dont l'adresse désigne la case `arr[4]` (située deux cases après celle désignée par `ptr1`) du tableau `arr`. Le résultat doit impérativement désigner une case du tableau [36], sinon le comportement du programme est indéfini.

On peut donc également incrémenter et décrémenter un pointeur, pour passer d'une case d'un tableau à la suivante ou à la précédente (y compris avec les opérateurs `++` et `--`). Ces opérations permettent d'écrire certains algorithmes sur les tableaux de façon parfois plus succinte. On peut, par exemple, afficher tous les entiers d'un tableau `arr` de taille `n` en écrivant [37] :

```
    for (int* ptr=arr; ptr<arr+n; ++ptr) {
        printf("%d ", *ptr);
    }
```

Puisque l'on s'interdira tout usage de l'arithmétique pointeur, nous continuerons dans la suite d'utiliser explicitement des index :

```
    for (int i=0; i<n; ++i) {
        printf("%d ", arr[i]);
    }
```

On peut toutefois en profiter pour expliquer par quelle magie l'écriture `arr[i]` fonctionne aussi bien sur des tableaux statiques que sur des pointeurs. En fait, dans toutes

36. Par commodité, il est permis d'obtenir de la sorte l'adresse suivant *immédiatement* le tableau, mais évidemment, il sera illégal de tenter de déréférencer cette adresse.

37. On remarquera `arr+n` ne désigne pas une case du tableau mais l'adresse suivant immédiatement celui-ci, le calcul est donc toléré et cette adresse peut être utilisée tant qu'elle n'est pas déréférencée.

les situations, `arr[i]` n'est qu'un *sucre syntaxique* (un raccourci d'écriture facilitant la vie au programmeur) pour `*(arr+i)`. Autrement dit, on calcule l'adresse obtenue en se décalant de `i` cases dans le tableau pointé par `arr`, puis on déréférence cette adresse, ce qui permet d'accéder à la case souhaitée [38]. Comme dans le cas d'un tableau statique le nom se dégrade en un pointeur vers la première case, il est donc tout naturel que l'écriture `arr[i]` fonctionne de la même façon dans les deux situations.

# 3 Quelques exemples concrets

## 3.1 Recherche dichotomique dans un tableau trié

### Principe

Supposons que l'on souhaite savoir si un élément est présent dans un tableau, et si oui à quelle position. Il est aisé d'écrire une fonction pour ce faire, comparant les éléments du tableau un par un à celui recherché (la fonction renverra l'index `-1` si l'élément n'est pas présent dans le tableau) :

```
int index(int arr[], int n, int val) {
    for (int i=0, i<n; ++i) {
        if (arr[i] == val) {
            return i;
        }
    }
    return -1;
}
```

De façon évidente, la fonction a une complexité linéaire ($O(n)$) dans le pire des cas, par exemple si l'élément recherché n'est pas présent dans le tableau. On ne peut pas faire mieux si l'on n'a pas d'informations supplémentaires sur le contenu du tableau.

Mais quiconque ayant déjà utilisé un annuaire ou un dictionnaire sait qu'il n'est pas nécessaire de lire la totalité de l'ouvrage pour trouver un numéro ou une définition. Ceux-ci profitent du fait que les données qu'il contient sont ordonnées. Si l'on consulte un endroit quelconque de l'ouvrage, on sait immédiatement, par une simple comparaison, si l'élément que l'on recherche se trouve avant ou après.

Si le tableau est trié, par exemple par ordre croissant, on peut donc exploiter cette possibilité pour localiser un élément particulier ou exclure sa présence dans le tableau. On commence par regarder au *milieu* du tableau, puis, s'il ne s'agit pas de l'élément recherché, on restreint la recherche à la première moitié du tableau (si l'élément recherché est plus petit que celui situé au milieu du tableau) ou à la seconde moitié (s'il est plus grand que celui-ci). On parle alors de *recherche dichotomique*, car le principe est de « couper en deux » les données.

### Une première implémentation récursive

Cela peut s'écrire simplement, en utilisant une fonction récursive et en examinant les différents cas :

```
int index(int arr[], int n, int val) {
    int m = n/2;                      // Dans [0..n[ si n>0

    if (n == 0) {                     // Absent si le tableau est vide
        return -1;
    } else if (arr[m] == val) {       // Trouvé !
        return m;
    } else if (arr[m] > val) {        // Pas dans la seconde moitié
        return index(arr, m, val);
    } else {                          // Pas dans la première moitié
        int res = index(&arr[m+1], n-m-1, val);
        if (idx == -1) {
            return -1;                // Pas non plus dans la seconde
        } else {
            return m+1+idx;           // Trouvé dans la seconde,
        }                             // on calcule l'index correct
    }
}
```

Cette fonction appelle plusieurs remarques.

- La variable `m` initialisée à `n/2` désigne l'index de la case « centrale ». Il s'agit ici d'une division entière, on calcule donc $\lfloor n/2 \rfloor$. Il s'agit bien exactement de la case centrale si $n$ est impair, et de la case juste à droite du milieu si $n$ est pair. On peut aisément vérifier que $0 \leqslant m < n$ dès lors que $n > 0$, aussi `arr[m]` désigne bien toujours une case du tableau.
- On profite ici de la possibilité d'utiliser un pointeur comme argument à la fonction pour désigner une *partie* du tableau initial (en $O(1)$!). Ainsi, en écrivant `index(arr, m-1, val)`, on continue la recherche dans les $m-1$ premières cases du tableau, et en écrivant `index(&arr[m+1], n-m-1, val)` on la poursuit dans les $n-m-1$ cases d'index $m+1$ à $n-1$ (inclus).
- le `m+1+idx` dans le dernier cas est nécessaire car si l'appel à `index` indique par exemple que l'élément se trouve à la position 4 dans le sous-tableau commençant à la case $m+1$, c'est qu'il se trouve à la position $m+1+4$ dans le tableau initial.

38. Il y aura toujours quelqu'un pour se demander, puisque l'opérateur `+` est commutatif, si écrire `i[arr]` fonctionne... De fait, oui, cela fonctionne, mais on oubliera *très* vite cette curiosité !

- La taille $n$, lors des appels récursifs, reste toujours positive (car $0 \leqslant m < n$) donc il n'est pas nécessaire de considérer le cas $n < 0$.
- La fonction est partiellement correcte si le tableau est trié, et termine car à chaque appel récursif le second argument décroit strictement ($m - 1 < m = \lfloor n/2 \rfloor < n$ si $n > 0$, et $n - m - 1 < n$ également), en restant un entier toujours positif ou nul. Il ne peut donc y avoir de séquence infinie d'appels récursifs. Vu différemment, l'élément `arr[m]`, *a minima*, ne sera plus considéré, et chaque appel élimine donc au moins[39] un élément.

### Une implémentation purement itérative

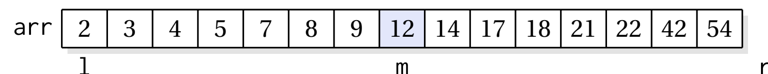
À titre d'illustration, on peut également chercher à écrire une version purement itérative, sans appel récursif, à la fonction. Méfiez-vous de son apparente et trompeuse simplicité, le diable se cache dans les détails et on a tôt fait de commettre une petite erreur (comparaison stricte dans l'expression booléenne du while, mauvaise mise à jour des indices, division flottante et non pas entière, etc.)

Pour la petite histoire, une expérience fut menée chez Bell, dans laquelle on a laissé plusieurs heures à des programmeurs professionnels pour programmer une recherche dichotomique. Seulement 10% d'entre eux ont réussi à proposer un code sans aucune erreur. Donald Knuth signale d'ailleurs que si l'algorithme a été proposé en 1946, il a fallu attendre 1962 pour voir la première publication d'une implémentation correcte.

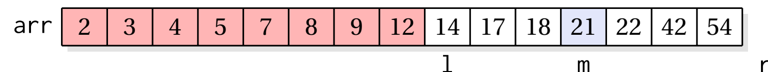
On pourrait continuer à restreindre le tableau par la méthode précédente en mettant à jour `arr` et `n`, mais il est plus simple ici de travailler avec deux indices `l` et `r` permettant de désigner les éléments, dans le tableau, qui restent à considérer. L'invariant de boucle est ici d'une importance cruciale pour ne pas se tromper dans la mise à jour de ces indices, dans leur initialisation et dans la condition d'arrêt de la boucle.

On suppose ici que les éléments restant à examiner sont ceux dont l'index $i$ vérifie $l \leqslant i < r$. On débutera donc la recherche avec $l = 0$ et $r = n$.

Pour illustrer le déroulement, supposons que l'on recherche l'élément 18 dans le tableau suivant. On commence par examiner l'élément au milieu de ce tableau :

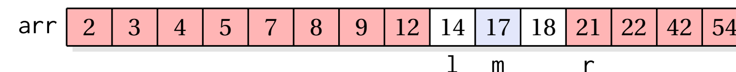



Comme cet élémentest plus petit que l'élément recherché, on élimine la moitié gauche du tableau (élément d'index $m$ compris) et on recommence :

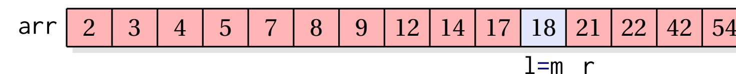



39. Généralement bien plus.

Cette fois l'élément à la position d'index $m$ est trop grand, on élimine la moitié droite des éléments restants :




Puis à nouveau la moitié gauche :




On localise enfin l'élément recherché. Si l'on avait recherché 19, le déroulement aurait été le même, on aurait effecté une itération supplémentaire, éliminant le 18, avant de pouvoir conclure en son absence du tableau.

Une implémentation possible de cet algorithme serait donc :

```c
int index(int arr[], int n, int val) {
    int l = 0, r = n;

    while (l < r) {
        // Les éléments restant à considérer sont deux
        // dans les cases d'index i où i se trouve dans [g..d[
        int m = (l+r)/2;      // Nécessairement dans [g..d[
        if (arr[m] == val) {
            return m;         // Trouvé !
        } else if (arr[m] > val) {
            r = m;            // On poursuit avec [l..m[
        } else {
            l = m+1;          // On poursuit avec ]m..r[ soit [m+1..r[
        }
    }
    return -1;                // Plus de possibilités
}
```

### Évaluation du nombre maximal de tests

Si le tableau contient 1 élément, il faudra évidemment tester un seul élément pour conclure. Si le tableau contient 3 éléments ou moins, la méthode dichotomique nécessitera de tester au plus deux éléments[40].

Si avec $k$ tests on peut traiter un tableau de longueur au plus $s_k$, alors avec $k+1$ tests, on pourra traiter un tableau de longueur au plus $s_{k+1} = 2s_k + 1$. On peut aisément montrer,

40. Par « tests », on entends à la fois la comparaison avec `val` pour l'égalité, et s'il n'y a pas égalité, la comparaison avec `<`. De toute façon, si l'on s'intéresse à la complexité, le facteur 2 n'a aucune importance.

puique $s_1 = 1$ (et $s_2 = 3$), que cela correspond à $s_k = 2^k - 1$.

Ainsi, pour un tableau de longueur $n$, il suffira de $k$ tests où $k$ vérifie $n \leq 2^k - 1$, soit $\ln(n+1) \leq k \ln(2)$, donc $k \geq \ln_2(n+1)$. Le nombre minimal de tests à effectuer pour déterminer si une valeur est présente dans un tableau *trié* de longueur $n$ est donc $\lceil \ln_2(n+1) \rceil$. Comme toutes les opérations à chaque itération (ou à chaque appel récursif) sont effectuées en temps constant, on a donc un algorithme de complexité logarithmique ($O\big(\log(n)\big)$).

Pour donner une idée de l'efficacité d'un tel algorithme, pour une recherche dans un dictionnaire français courant (qui contient de l'ordre de 60000 mots), il suffirait donc, en principe, de vérifier seulement dix-huit mots pour pouvoir conclure quant à la présence d'un mot donné dans ledit dictionnaire avec l'algorithme de la recherche dichotomique.

#### Pourquoi couper au milieu ?

On peut se demander pourquoi couper au milieu plutôt qu'ailleurs. Supposons que le tableau contient initialement $n$ éléments, pour lesquels on ne sait rien, et examinons la première coupure.

La chance de tomber « par hasard » sur la bonne valeur ne dépend pas de l'élément choisi, donc on ne s'intéressera qu'à ce qui se passe si l'élément choisi n'est pas celui recherché.

Les index vont de 0 à $n-1$. Choisissons l'élément d'index $k$ pour couper le tableau en deux, pas nécessairement au milieu. On suppose donc que l'élément d'index $k$ n'est pas l'élément recherché. On se retrouve avec deux tableaux, l'une contenant $k$ éléments, l'autre contenant $n-k-1$ éléments.

L'élément recherché pouvant être n'importe où, la probabilité de le trouver dans chacun de ces deux sous-tableaux est directement proportionnelle au nombre d'éléments dans ces derniers [41].

L'espérance $\mathbb{E}$ du nombre d'éléments qui restent à examiner ensuite peut donc s'écrire

$$\mathbb{E} = k \times \frac{k}{n-1} + (n-k-1) \times \frac{n-k-1}{n-1} = \frac{k^2 + (n-k-1)^2}{n-1}$$

On souhaite évidemment que le tableau qui reste à examiner soit la plus court possible, donc on veut minimiser cette espérance, et ainsi choisir le $k$ qui minimise $k^2 + (n-k-1)^2$.

La dérivée par rapport à $k$ de cette expression est $2 \times k - 2 \times (n-k-1)$, ce qui donne $4k - 2(n-1)$. Cette expression s'annule pour $k = \frac{n-1}{2}$. Puisque la dérivée seconde est positive, cette valeur correspond bien à un minimum de l'espérance. $k$ devant être entier, on effectuera au besoin un arrondi à un des deux entiers les plus proches (ce que fait la division entière dans l'algorithme).

41. Toujours en ne faisant aucune hypothèses sur les éléments. Il est possible, dans certains cas, que la connaissance de `arr[k]` nous informe sur la position probable de l'élément recherché, ce qui peut éventuellement se traduire en un algorithme légèrement plus efficace.

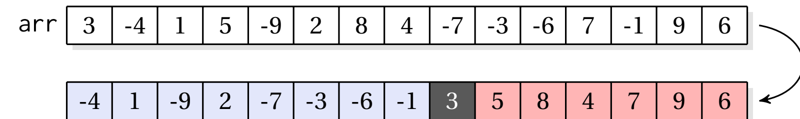
## 3.2 Tri rapide

#### Principe

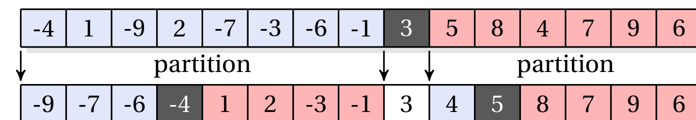
Revenons un temps sur la problématique du tri des éléments d'un tableau d'entiers. Les tris de complexité quadratique vus au chapitre précédent ne permettent pas de trier dans un temps raisonnable des tableaux contenant des centaines, voire des millions d'éléments. Nous allons nous intéresser ici à un algorithme dit du « *tri rapide* » (*quicksort* en anglais) offrant généralement de bien meilleures performances.

Ce tri est basé sur une opération élémentaire de *partition* de ses éléments en deux groupes distincts. On choisit un élément du tableau, que l'on appelle `pivot`. Les éléments restants sont répartis en deux groupes disticts. Les éléments strictement inférieurs au pivot sont placés dans le premier groupe, les éléments strictement plus grands dans le second. Les éléments égaux au pivot peuvent être placés indifféremment dans un groupe ou l'autre, l'important étant que chacun des éléments se retrouve dans un groupe et un seul.

On suppose donc disposer d'une fonction `int partition(int arr[], int n)` prenant en argument un pointeur `arr` vers un tableau de `n` entiers, considère le premier élément comme pivot [42], et réorganise les éléments du tableau en place de façon à ce que les éléments à gauche du pivot soient inférieur ou égaux à celui-ci, et ceux à droite lui soient supérieurs, comme ci-dessous, et renvoie l'index auquel se trouve le pivot à l'issue du traitement [43] [44].

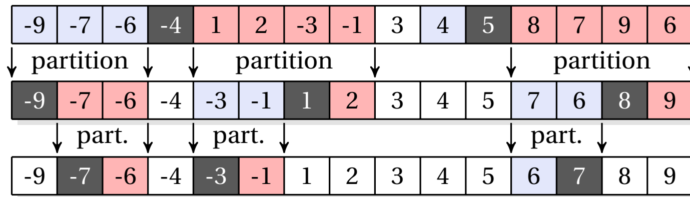



Lorsque l'on partitionne ainsi un tableau, cela ne suffit pas à trier les éléments. La plupart ne sont pas à la bonne place. Mais un élément est nécessairement à la bonne place : il s'agit du pivot. Pour placer correctement les autres éléments, nous allons commencer par appliquer, à nouveau, le principe du partitionnement sur chacun des deux groupes séparément, ce qui nous conduit, pour notre exemple, à trois élements supplémentaires bien placés, les deux pivots de chacune des deux partitions, mais également la valeur 4 qui, « coincée » entre deux éléments bien placés ne peut que l'être également :




42. Le choix du pivot est ici arbitraire, nous en discuterons un peu plus tard.
43. Nous proposerons des implémentations de cette fonction `partition` dans la section suivante.
44. Sur l'exemple proposé, la partition représentée est stable, c'est-à-dire que l'ordre relatif des éléments plus petits que le pivot et celui des éléments plus grand que le pivot sont préservés. Ce n'est virtuellement jamais le cas dans les fonctions de partition que l'on écrira.

Tant qu'il reste des groupes avec plusieurs éléments, on continue à appeler la fonction `partition` sur chacun d'entre eux, jusqu'à parvenir à des morceaux de tableau de longueur au plus 1, et donc à un tableau intégralement trié :




#### Implémentation

Si l'on dispose de la fonction `partition`, le tri rapide est simple à implémenter de façon récursive, puisqu'il se limite à un appel à `partition` suivi de deux appels récursifs, pour les seuls tableau de taille supérieure ou égale à deux :

```c
void quicksort(int arr[], int n) {
    if (n >= 2) { // Si n=0 ou 1, il n'y a rien à faire !
        int idx = partition(arr, n);
        quicksort(arr, idx);
        quicksort(&arr[idx+1], n-1-idx);
    }
}
```

La terminaison de l'algorithme est assurée par le fait que la position `idx` du pivot après la partition vérifie nécessairement $0 \leqslant \texttt{idx} < n$, ce qui assure que les tailles des deux tableaux dans les deux appels récursifs sont bien strictement inférieures[45] à celle du tableau passé en argument de la fonction `quicksort`.

#### Complexité

Si, à chaque partition, tous les éléments se retrouvent dans le même groupe, alors la taille des tableaux à partitionner n'est réduite que de 1 élément à chaque étape. On appellera successivement la fonction `partition` sur des tableaux de taille $n$, puis $n-1$, $n-2$, etc. Elle sera donc appelée en particulier au moins $n/2$ fois sur des tableaux de taille supérieure ou égale à $n/2$. Comme le partitionnement, on l'a dit, a un coût linéaire (a minima, il doit examiner tous les éléments), la complexité du tri dans ce cas est quadratique ($O(n^2)$).

On peut aisément voir qu'il s'agit du cas le plus défavorable : quoi qu'il arrive, un élément ne peut pas être comparé à un pivot plus de $n-1$ fois (à chaque fois, s'il n'est pas choisi comme pivot, il se retrouve dans un morceau de tableau plus petit, et le processus s'arrête dès lors qu'il se retrouve pivot ou isolé).

45. Et positives ou nulles !

En revanche, si le pivot est tel que les deux groupes ont un cardinal comparable, la taille des tableaux est typiquement réduite de moitié à chaque appel récursif. Chaque élément se retrouve comparé au plus un nombre de fois de l'ordre de $\log_2(n)$, ce qui donne un nombre d'opérations pour la totalité du tri de l'ordre de $n\log_2(n)$.

En soi, cela ne paraît pas si intéressant : une complexité quadratique ($O(n^2)$) dans le pire des cas, et $O(n\log(n))$ dans des cas favorables. Nous avons vu que le tri par insertion était lui aussi quadratique dans le pire des cas, mais devenait linéaire dans les cas les plus favorables. Pour comprendre l'intérêt du tri rapide, il nous faut réfléchir à une complexité *en moyenne*. Les cas favorables sont rares pour le tri par insertion, mais dans le cas du tri rapide, ce sont les cas *défavorables* qui sont rares.

Si l'on note $u_n$ le nombre de comparaisons effectuées en moyenne pour un tableau de taille $n$, en supposant l'équirépartition du pivot, on a

$$u_n = n-1+\frac{1}{n}\sum_{k=0}^{n-1} u_k + u_{n-1-k} = n-1+\frac{2}{n}\sum_{k=0}^{n-1} u_k.$$

Pour tout $n \geqslant 1$, on peut écrire

$$\begin{aligned} nu_n-(n+1)u_{n-1} &= nu_n-(n-1)u_{n-1}-2u_{n-1} \\ &= \left(n(n-1)+2\sum_{k=0}^{n-1}u_k\right)-\left((n-1)(n-2)+2\sum_{k=0}^{n-2}u_k\right)-2u_{n-1} \\ &= 2(n-1). \end{aligned}$$

Par conséquent,

$$\frac{u_n}{n+1}-\frac{u_{n-1}}{n} = \frac{2(n-1)}{n(n+1)}.$$

Ce qui conduit, par télescopage, à

$$\begin{aligned} \frac{u_n}{n+1} &= \frac{2(n-1)}{n(n+1)}+\frac{2(n-2)}{(n-1)n}+\ldots+u_0 \\ &= 2\left(\frac{1}{n}-\frac{2}{n(n+1)}\right)+2\left(\frac{1}{n-1}-\frac{2}{(n-1)n}\right)+\ldots+u_0 \\ &= 2\left(\sum_{k=1}^{n}\frac{1}{k}-\frac{2n}{n+1}\right)+u_0. \end{aligned}$$

Puisque $\log(n+1)-1 < \sum_1^n 1/k < \log(n+1)$, on a donc un nombre de comparaison $u_n$ en $\Theta(n\log(n))$. En *moyenne*, le tri rapide a un comportement quasilinéaire, ce qui en fait un tri très efficace.

#### Meilleur choix du pivot

On voit donc que le pivot a une importance considérable si l'on souhaite que le tri rapide fonctionne bien. Idéalement, il faut qu'il y ait un nombre comparable d'éléments dans chacun des deux groupes construits par `partition`. Il existe donc des choix plus intéressants que le premier élément du tableau : en effet si le tableau est trié par exemple, c'est un des plus mauvais choix que l'on puisse faire.

Une meilleure solution consiste à choisir le pivot au hasard parmi les éléments du tableau [46] [47]. Cela ne garantit cependant pas que le hasard ne fasse pas, ponctuellement, mal les choses, même si c'est extrêmement peu probable que le hasard fasse systématiquement de mauvais choix. Un tel tri faisant intervenir le hasard est qualifié de *stochastique*.

Une autre stratégie intéressante consiste à choisir la médiane entre le premier élément, le dernier, et celui se trouvant au milieu. Il existe des solutions de coût linéaire permettant de sélectionner un pivot qui *garantisse* une complexité en $\Theta(n\log(n))$ (méthode de la médiane des médianes par exemple), mais cette solution n'est virtuellement jamais mise en place car elle est complexe pour un gain que l'on n'observera (statistiquement) jamais, et un surcoût qui, lui, sera toujours présent. Si l'on a besoin d'une garantie sur la complexité du tri, on préférera généralement se tourner vers d'autres solutions (tri fusion, tri par tas...)

#### Méthode de tri mixte

Si la méthode du tri rapide fonctionne très bien sur les grands tableaux, pour des tableaux de petite taille (généralement moins de dix éléments), des tris quadratiques, notamment le tri bulle, peuvent se révéler plus performants. Afin d'avoir l'algorithme de tri le plus efficace possible, il est donc raisonnable de commencer par réduire la taille des zones à trier à quelques éléments en utilisant un tri rapide, puis de se servir d'un algorithme différent pour les petits tableaux. On écrira par exemple quelque chose de ce genre :

```c
void quicksort(int arr[], int n) {
    if (n > 10) {
        int idx = partition(arr, n);
        quicksort(arr, idx);
        quicksort(&arr[idx+1], n-1-idx);
    } else {                    // Pour les petits tableaux,
        bubblesort(arr, n);     // on change de stratégie de tri
    }
}
```

46. Ce qui ne nécessite pas de modifier la fonction `partition`, on ajoute simplement une permutation entre le premier élément du tableau et un élément choisi au hasard avant l'appel à `partition`.
47. Alternativement, on peut garder l'algorithme en l'état et simplement mélanger le tableau avant de le trier, ce qui peut être fait aisément en temps linéaire, donc négligeable devant le temps nécessaire au tri.

## 3.3 Partitionner un tableau

Si l'on veut pouvoir implémenter un tri rapide, il nous faut disposer d'une fonction qui prend un tableau et le partitionne comme décrit précédemment. Il existe de nombreuses manières d'obtenir le résultat souhaité. Une seule suffit, mais nous allons en présenter plusieurs, afin d'illustrer comment différents choix d'invariants de boucle conduisent à des algorithmes subtilement différents.

#### Un premier exemple

On souhaite partitionner un tableau en considérant le premier élément du tableau comme pivot [48], autrement dit passer de cette situation :

| arr | 3 | -4 | 1 | 5 | -9 | 2 | 8 | 4 | -7 | -3 | -6 | 7 | -1 | 9 | 6 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|

à celle-ci, correspondant au résultat souhaité [49] :

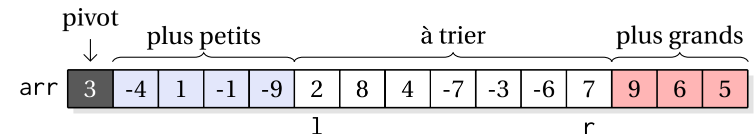
| arr | -7 | -4 | 1 | -1 | -9 | 2 | -6 | -3 | 3 | 4 | 7 | 8 | 9 | 6 | 5 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|

Plutôt que de chercher à obtenir directement ce résultat, on peut, dans un premier temps, s'abstenir de déplacer le pivot, et chercher tout d'abord à parvenir à cette situation :

| arr | 3 | -4 | 1 | -1 | -9 | 2 | -6 | -3 | -7 | 4 | 7 | 8 | 9 | 6 | 5 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|

Il ne restera alors qu'un échange à faire entre le pivot et le dernier des éléments qui lui sont inférieurs pour obtenir le résultat attendu.

Pour voir comment nous pouvons parvenir à cette situation, imaginons une étape au cœur de l'algorithme, lorsqu'une partie des éléments à partitionner a été traitée, mais pas encore la totalité. La situation pourrait par exemple ressembler au schéma ci-dessous. Ce schéma nous fera office d'invariant de boucle [50].

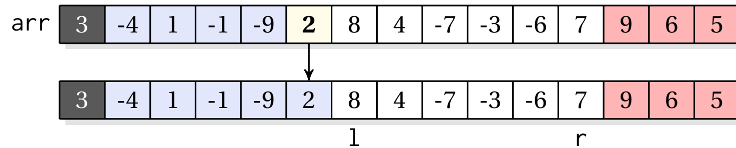



Le pivot est demeuré dans la première case du tableau, divers éléments du tableau, identifiés comme inférieurs au pivot, ont été regroupés sur la gauche, et d'autres, identifiés comme supérieurs au pivot, ont été regroupés sur la droite. Entre ces deux groupes subsiste un ensemble d'éléments qui n'ont pas encore été examinés.

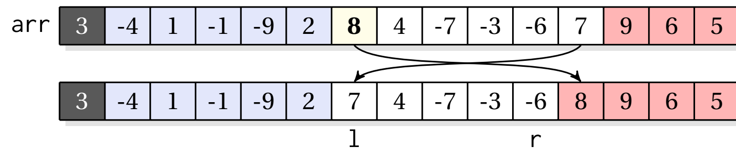
48. Si le premier élément du tableau n'est pas un bon choix, on aura pu préalablement à l'appel à la fonction partition l'échanger avec un autre élément du tableau, donc ce n'est pas ici une limitation.
49. On remarquera que l'ordre des éléments dans chacun des deux groupes a été altéré, l'ordre indiqué ici correspond à celui qui sera obtenu avec l'algorithme présenté ci-après
50. Un schéma peut faire un excellent invariant de boucle, en l'accompagnant éventuellement de quelques explications.

Afin de pouvoir, à tout instant, savoir à quel groupe appartiennent chaque éléments, il nous faut mémoriser où se situent les frontières les séparant. Plusieurs solutions sont possibles, nous avons choisi ici d'identifier par `l` l'index de la première des cases contenant des éléments restant à examiner, et par `r` l'index de la dernière d'entre elles. C'est encore une fois l'invariant de boucle qui nous permet de communiquer, à la personne qui lira l'algorithme, les choix qui ont été effectués.

Pour progresser dans le partitionnement, il faut considérer un des éléments restant à examiner, et le placer dans l'un des deux groupes. On peut par exemple s'intéresser à l'élément à la position `l` dans le tableau. S'il est plus petit que le pivot, il est inutile de le déplacer. En revanche, on incrémente `l` pour indiquer qu'un élément de plus a été traité :

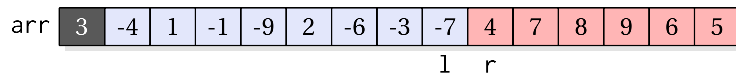



En revanche, s'il est plus grand que le pivot, il faut le déplacer à proximité des éléments déjà identifiés comme état plus grand que le pivot, en l'échangeant avec le dernier des éléments restant à trier, autrement dit l'élément dans la case d'index `r`. On peut ensuite décrémenter `r` pour indiquer qu'un élément de plus a été traité :

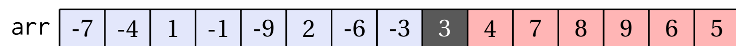



Lorsque l'on rencontre un élément égal au pivot, on peut indifféremment le placer dans l'un ou l'autre des deux groupes [51].

L'algorithme se poursuit tant qu'il reste des éléments à examiner, c'est-à-dire tant que [52] `l <= r`, jusqu'à parvenir à la situation suivante :




Il reste à placer correctement le pivot, ce qui peut se faire, d'après le schéma précédent, en échangeant les cases d'index `0` et `r`, ce qui conduit bien à l'état final attendu :




La fonction termine car la quantité `r-l+1` (qui représente le nombre d'éléments restant à examiner) peut nous servir de variant : à chaque itération de la boucle **`while`**, cette valeur, entière est diminuée d'une unité, et si elle devient négative (donc si `r<l`), on sort de la boucle. il ne peut donc pas y avoir de boucle s'exécutant indéfiniment ici.

En terme de complexité, nous avons bien une complexité linéaire puisque le contenu de la boucle a une complexité constante ($O(1)$), et que la boucle est exécutée $n-1$ fois.

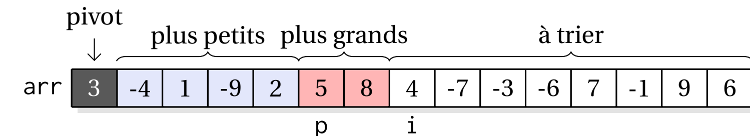
Initialement, les éléments à examiner se trouvent dans les cases d'index 1 à $n-1$, on initialise donc `l` à 1 et `r` à $n-1$. Pour le reste, il ne reste plus qu'à traduire le tout en langage C, ce qui ne présente plus guère de difficultés. Cela donne par exemple :

```c
int partition(int arr[], int n) {
    int pivot = arr[0];
    int l = 1, r = n-1;
    while (l <= r) {
        // Invariant : se reporter au schéma
        if (arr[l] <= pivot) {
            l++;
        } else {
            array_swap(arr, l, r);
            r--;
        }
    }
    array_swap(arr, 0, r);  // On replace le pivot
    return r;               // et on renvoie sa position
}
```

### Partitionnement de Lomuto

La précédente approche n'est pas fréquemment utilisée en pratique [53]. Un partitionnement plus fréquemment utilisé est celui proposé par Nico Lomuto.

Ce qui change, c'est l'ordre dans lequel se trouvent les différents groupes lors du partitionnement. L'invariant de boucle correspond à la situation suivante :



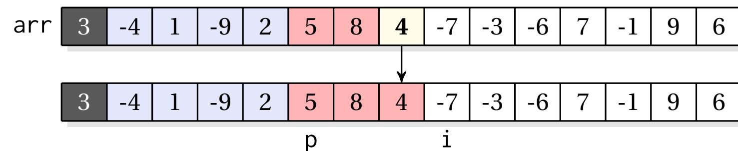

51. S'ils risquent d'être nombreux, il peut être intéressant de chercher à les répartir dans les deux groupes, afin d'équilibrer lesdits groupes, mais cela peut rendre l'algorithme plus complexe. Nous verrons que la méthode de Hoare pour le partitionnement réparti naturellement de façon équitable les éléments égaux au pivot dans les deux groupes.

52. Attention à la condition ! Ici, lorsque `l == r`, il reste encore un élément à examiner.

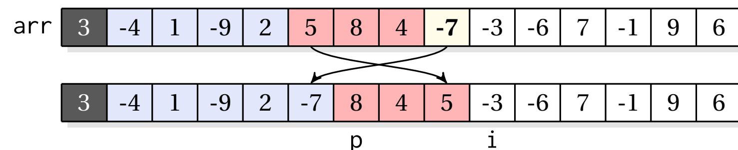
53. Une raison peut se trouver dans le fait que l'élément examiné lors de l'étape $k+1$ dépend de ce qui se passe dans l'étape $k$, ce qui peut avoir un impact négatif sur sa vitesse d'exécution en comparaison du partitionnement qui va suivre. Ce dernier conduit par ailleurs à une implémentation légèrement plus simple, même s'il peut être un tantinet plus subtil.

Comme précédemment, pour identifier les frontières entre les différents groupes, on utilise deux variables : `i` est l'index de la première des cases contenant un élément non considéré, `p` la case qui suit immédiatement les éléments identifiés comme plus petits que le pivot.

Pour progresser, lorsque l'élément dans la case d'index `i` est plus grand que le pivot, il n'y a rien d'autre à faire que d'incrémenter `i` :




En revanche, si l'élément dans la case d'index `i` est plus petit que le pivot, il convient d'échanger les contenus des cases d'index `i` et `p`, et d'incrémenter les *deux* variables `i` et `p` :




La variable `i` augmente donc à chaque itération, évoluant de 1 jusque $n-1$, de sorte qu'il est possible d'écrire la partition comme une boucle `for`. La valeur initiale de `p` sera 1, puisque l'on n'a initialement aucun élément identifié comme plus petit que le pivot.

À l'issue de la boucle, le replacement du pivot entre les deux groupes se fera, comme précédemment, en échangeant le contenu de la case d'index 0 et la dernière case contenant un entier plus petit que le pivot, donc d'index `p-1`.

Une implémentation possible de cet algorithme sera donc :

```c
int partition_Lomuto(int arr[], int n) {
    int pivot = arr[0];
    int p = 1;
    for (int i=1; i<n; ++i) {
        // Invariant : se reporter au schéma
        if (arr[i] <= pivot) {
            array_swap(arr, i, p);
            p++;
        }
    }
    array_swap(arr, 0, p-1);
    return p-1;
}
```

Si l'on regarde de plus près le fonctionnement de l'algorithme, on constate que tant que l'on n'a identifié aucun élément plus petit que le pivot, les contenus des variables `i` et `p` sont égaux, et conduit donc à des échanges d'une case avec elle-même. Ce n'est pas un souci tant que la fonction `array_swap` le permet. Il serait plus pénalisant d'essayer de traiter ces cas comme des exceptions [54].

Comme tous les algorithmes de partition que nous étudierons, la partition de Lomuto a une complexité évidemment linéaire [55]

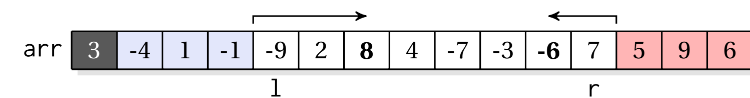
**Réduire les échanges avec la partition de Hoare**

On peut s'attendre à ce que les deux algorithmes précédents effectuent, en moyenne, $n/2$ permutations environ. On peut vouloir chercher à limiter le nombre de ces permutations.

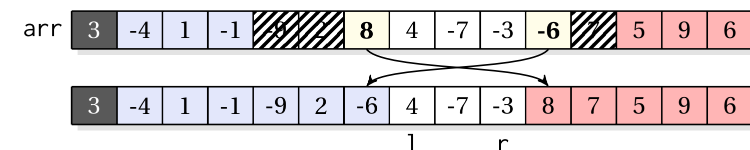
Si l'on reprend le premier exemple, on se déplace dans le tableau tant que les valeurs sont plus petites que le pivot, en augmentant la valeur de la variable `g`, et lorsque ce n'est plus le cas, on échange la valeur problématique avec une valeur dont on ne sait rien. Il pourrait être intéressant de décrémenter également la valeur de `d` tant que les cases correspondantes contiennent des valeurs plus grandes que le pivot jusqu'à en trouver une qui soit plus petite.

On pourrait alors échanger le contenu de la case d'index `g` (contenant une valeur plus grande que le pivot) et de la case d'index `d` (contenant une valeur plus petite que le pivot), ce qui permettrait de placer correctement deux valeurs d'un coup en un seul échange !

Par exemple, dans la situation suivante, pour progresser dans la partition, on peut voir que les éléments $-9$ et 2 sont convenablement placés mais que 8, supérieur au pivot, devra être déplacé, et de même que 7 est bien placé, mais que $-6$ devra lui aussi être déplacé :




On peut donc échanger ces deux éléments, et après cet échange, cinq éléments de plus sont identifiés comme bien placés :




54. Sauf à allonger l'algorithme en divisant la boucle en deux boucles distinctes, selon que l'on ait ou non déjà identifié un élément plus petit que le pivot.

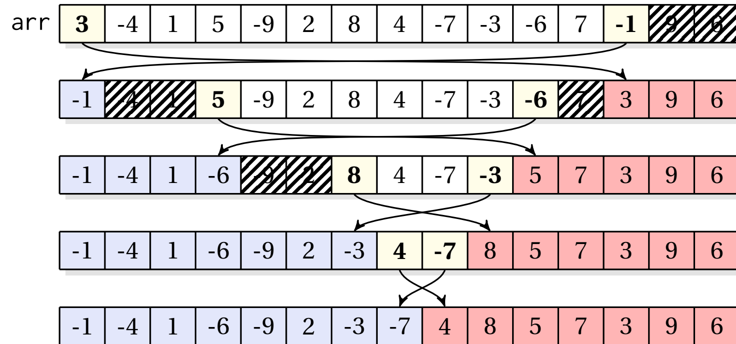
55. Toutefois, du point de vue de la machine, cet algorithme a l'avantage d'être très prévisible (en particulier, le prochain élément qui sera considéré n'est pas affecté par le traitement de l'élément précédent), ce qui en fait un algorithme que le processeur pourra exécuter efficacement.

Une difficulté cependant apparaît : il nous faut éviter que les deux recherches d'éléments mal placés ne « sortent » du tableau. Cela peut être en grande partie évité en utilisant une comparaison *stricte* avec le pivot. En effet, un élément ne pouvant pas être à la fois strictement inférieur et strictement supérieur au pivot, l'élément dans la case d'index `r+1` a été identifié comme supérieur ou égal au pivot, `l` s'arrêtera nécessairement au plus tard en `r+1`. De même pour `r` qui s'arrêtera au plus tard en `l-1` si l'élément `l-1` a été identifié comme inférieur ou égal au pivot. Cela garantit donc que, après qu'ait eu lieu un premier échange, les deux recherches ne peuvent pas progresser au-delà des limites du tableau.

L'ennui, c'est la toute première recherche. Si celle partant de l'extrémité droite en direction de la gauche s'arrêtera dans le pire des cas sur le pivot, celle partant de l'extrémité gauche peut très bien sortir si tous les éléments sont strictement inférieurs au pivot.

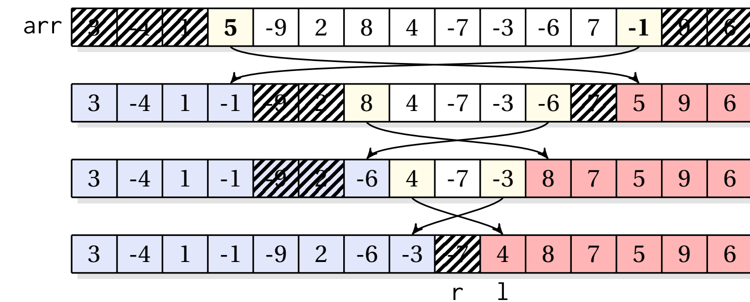
Dans l'algorithme de partitionnement proposé par Antony Hoare, l'idée est d'inclure le pivot dans la zone de recherche (en partant donc de `l=0`). Ainsi, dans le pire des cas, les deux recherches « buteront » sur le pivot [56]. Après le premier échange, si tout le tableau n'est pas encore partitionné, on aura au moins un élément strictement plus petit que le pivot dans la première case, et un élément strictement plus grand dans la dernière case, ce qui garantit que les recherches ultérieures ne pourront plus sortir du tableau.

Appliqué au tableau d'exemple, avec toujours pour pivot l'élement 3 dans la première case, cela donne donc :




L'« ennui » avec cette approche est que le pivot se retrouve déplacé [57] et ne finit pas nécessairement coincé entre les deux groupes, ce dont il nous faudra tenir compte lorsque nous utiliserons ce partitionnement.

Comme le pivot sera déplacé comme les autres éléments, il n'est, par ailleurs, plus guère utile de le choisir dans la première case du tableau. On choisira généralement l'élément au *milieu* du tableau comme pivot, car puisque les recherches « butent » sur le pivot, autant faire en sorte qu'il soit le plus loin possible des extrémités [58]. Voici donc le déroulement de la partition si le pivot choisi est la valeur située au milieu du tableau [59] (ici 4) :




On n'effectuera d'échange que si `l<r`, et, de même, l'opération de partitionnement s'arrête dès que `l>=r` [60]. Une implémentation possible de cet algorithme est donc :

```c
int partition_Hoare(int arr[], int n) {
    int pivot = arr[n/2];  // Ne peut être le dernier si n>1
    int l = 0, r = n-1;
    while (true) {
        // Les éléments d'index <g sont inférieurs ou égaux au pivot
        // Les éléments d'index >d sont supérieurs ou égaux au pivot
        // Les éléments restant à examiner sont ceux
        // dont l'index est dans l'intervalle [l..r]
        while (tab[l] < pivot) { l++; }
        while (tab[r] > pivot) { r--; }
        if (l <= r) {
            return r; // Le groupe des "plus petits" concerne les
        }             // éléments d'index 0 à r inclus
        array_swap(arr, l, r);
        l++; r--;
    }
}
```

On peut aisément montrer que `l` et `r` restent toujours des index valides (ils ne peuvent sortir du tableau lors des deux boucles `while` internes, comme nous l'avons dit, et en cas de permutation de deux éléments, on a nécessairement `l<r` avant la permutation, donc `l`

56. Dès le premier test dans la recherche progressant vers la droite si le pivot se trouve dans la première case.
57. On pourrait mémoriser son placement, surtout s'il se trouve initialement dans la première case est est donc déplacé dès le premier échange et ne bouge plus ensuite, mais nous verrons que ce n'est pas utile.

58. Il s'agit par ailleurs souvent d'un bon choix de pivot : par exemple, si la liste est déjà triée, le partitionnement se déroulera sans aucun échange! La permutation avec un élément choisi aléatoirement est nettement moins utile dans le cas de la partition présentée ici.
59. Le pivot se retrouve sur cet exemple au bord d'un des groupes, c'est relativement courant s'il est choisi initialement au milieu, mais ça n'a rien de systématique, et il peut se retrouver dans n'importe lequel des deux groupes.
60. Dans la pratique, on aura nécessairement `l=r` ou `l=r-1`.

et r restent des index valides après l'incrémentation de l et la décrémentation de r dans la dernière ligne de la boucle principale [61].

À l'issue de l'algorithme, les éléments dans les cases d'index 0 à r inclus sont tous inférieurs ou égaux au pivot, et ceux dans les cases d'index r+1 à n-1 sont tous supérieurs ou égaux au pivot. L'algorithme renvoie r pour indiquer où se trouve la frontière entre les deux groupes. Par ailleurs, et c'est important, si n> 1, on a r< $n-1$ à l'issue de l'algorithme [62]. Par conséquent, **aucun des deux groupes ne peut être vide** puisque $0 \leqslant r < n-1$.

Attention cependant, après la partition de Hoare précédente, il n'y a donc *aucun* élément qui soit bien placé avec certitude. L'appel récursif sur les deux groupes doit donc être subtilement différent pour tenir compte du fait que l'élément dans la case dont l'index est renvoyé par la fonction partition_Hoare ne doit plus être ignoré. Plus précisément, le premier groupe concerne dorénavant les éléments dans les cases d'index 0 à idx=r *inclus*. Il faut donc réécrire le tri rapide de la sorte :

```
void quicksort_Hoare(int arr[], int n) {
    if (n >= 10) {
        int idx = partition_Hoare(arr, n);
        quicksort_Hoare(arr, idx+1);          // <- Attention au +1 !
        quicksort_Hoare(&arr[idx+1], n-idx-1);
    } else {
        bubblesort(arr, n);
    }
}
```

La terminaison de l'algorithme reste garantie, car aucun des deux groupes n'étant vide, les tableaux, dans les appels récursifs, sont tous deux strictement plus petits.

Le partionnement de Hoare présente plusieurs avantages. Tout d'abord, il effectue en moyenne (et dans le pire des cas) moins d'échanges d'éléments [63]. De façon plus intéressante, s'il y a un grand nombre d'éléments égaux au pivot, ils sont naturellement répartis de façon équitable entre les deux groupes, ce qui n'est pas le cas de la plupart des autres partitions. Or, si tous les éléments se retrouvent systématiquement du même côté, la complexité du tri devient quadratique. L'approche proposée par A. Hoare est donc quasi-linéaire pour un tableau où tous les éléments sont égaux!

61. Ces incrémentation/décrémentation peuvent ne pas apparaître dans les implémentations proposés dans les ouvrages que vous pourriez consulter, où des boucles un peu différentes des boucles while peuvent être utilisées (avec une exécution systématique du corps de la boucle avant la première vérification de la condition). L'initialisation de l et r sera alors plutôt $-1$ et $n$.

62. même si le dernier élément est plus petit que le pivot, il subira forcément une permutation, au plus tard avec l'élément choisi au milieu du tableau, et qui ne peut être le dernier élément, comme pivot.

63. Cependant, en raison d'un comportement un peu moins prédictible de l'algorithme au niveau du processeur à cause des deux boucles while, il peut se révéler quand même moins rapide, sous certaines conditions que le partitionnent de Lomuto, quand bien même ce dernier effectue davantage d'échanges. Encore une fois, le décompte exact du nombre d'opérations n'est pas toujours le critère le plus pertinent pour juger de l'efficacité d'un algorithme, utilisation du cache et prédiction de branchement jouent un rôle important dans ces questions.

# 4 Retour sur les tableaux multidimensionnels

## 4.1 Pointeurs vers les tableaux multidimensionnels

Un tableau à deux dimensions n'est, rappelons-le, qu'un tableau contenant des tableaux de tailles identiques. Ainsi, un tableau défini par

```
int arr[2][3];
```

est un tableau contenant deux éléments de type int[3], soit deux tableaux de trois entiers.

Lorsque l'on a besoin d'un pointeur vers un tableau, on manipule, on l'a vu, simplement un pointeur vers son premier élément. Ainsi, un pointeur vers un tableau unidimensionnel contenant des int sera simplement un int*.

On peut de la même façon utiliser des pointeurs vers des tableaux multidimensionnels [64], mais le premier élément étant alors un tableau, le pointeur doit pointer vers un objet de type tableau. Par exemple, pour le tableau tab défini ci-dessus, il nous faut un pointeur pointant vers des tableaux de trois entiers.

On définira un tel pointeur de la sorte [65] :

```
int (*ptr)[3] = &arr[0];
```

La syntaxe C a été pensée pour simplifier la tâche au compilateur, ce qui rend certaines de ces déclarations délicates à lire. On peut néanmoins interpréter l'écriture précédente comme « l'objet pointé par ptr est un tableau de trois entiers ». Les parenthèses sont indispensables car « int *ptr[3] » serait interprété par « ptr est un tableau de trois pointeurs vers des entiers », du fait de la précédence de [] sur *.

De tels pointeurs sont par exemple utiles lorsque l'on souhaite allouer dynamiquement des tableaux multidimensionnels. Pour allouer un tableau d'entiers de taille $2 \times 3$, on écrira donc par exemple [66] :

```
int (*arr)[3] = malloc(2*3*sizeof(int));
```

64. La formulation du programme laisse à penser que les tableaux à une seule dimension seront privilégiés, quitte à représenter un tableau à plusieurs dimensions en un tableau à une seule dimension en manipulant les indices comme illustré dans le chapitre précédent. Vraisemblablement, la bonne connaissance de cette partie du chapitre n'est pas indispensable, mais il nous a semblé intéressant d'en dire un mot pour ceux qui en ressentiraient le besoin au-delà du cadre de la MP2I.

65. On peut aussi l'initialiser avec « = arr ».

66. La conversion explicite du type renvoyé par malloc, requise par le programme, n'a rien de triviale! Le type « int(*)[3] » nécessite bien des parenthèses pour bien avoir, comme précédemment, un pointeur vers un tableau contenant des int[3] et non un tableau de trois pointeurs vers des entiers (int*).

## 4.2 Tableaux multidimensionnels et fonctions

Cette notation permet aussi d'écrire des fonctions prenant en argument des tableaux à plusieurs dimensions. Par exemple, une fonction prenant un tableau de taille $n \times 3$ pourra être déclarée comme :

```
void foo(int n, int (*arr)[3]) {
    ...
}
```

On peut également, comme précédemment, faire disparaître le symbole « * » au profit de « [] » pour des raisons de lisibilité, et utiliser la syntaxe suivante, que l'on peut possiblement trouver plus naturelle :

```
void foo(int n, int arr[][3]) {
    ...
}
```

On remarquera cependant que si la taille du tableau vis-à-vis du premier index est spécifiée à part (comme pour un tableau à une seule dimension), la taille dans la seconde direction apparaît nécessairement comme partie intégrante du type du tableau, et doit être connue dès la compilation : on a besoin de savoir qu'il s'agit d'un tableau de `int[3]` !

Dans un tableau avec plus de deux dimensions, de la même façon, toutes les tailles, à l'exception de la toute première, devront donc être précisées.

Peut-on écrire une fonction qui accepterait un tableau à deux dimensions, dont les deux tailles peuvent varier d'un appel sur l'autre? *A priori* non, mais dans les versions récentes du langage, le support (optionnel actuellement) des *variable length arrays*[67] (VLA) donne accès à des pointeurs vers de tels objets, ce qui permet d'écrire :

```
void foo(int n, int p, int arr[][p]) {
    ...
}
```

Dans la déclaration précédente, `arr` est utilisé pour désigner un tableau à deux dimensions, donc la taille selon la seconde direction est `p`, et est précisée sous la forme d'un paramètre (qui doit figurer *avant* `arr` dans les paramètres de lafonction). Elle peut donc être utilisée pour écrire une fonction attendant un tableau de taille $n \times p$.

67. Que nous avons brièvement évoqués en notant leur caractère hors-programme. Si les tableaux à longueur variable (VLA) ont des inconvénients et des limitations, les pointeurs vers ces tableaux ne posent aucun réel problème pratique.

## 4.3 Tableaux de pointeurs et tableaux « Iliffes »

Un tableau peut contenir des objets de n'importe quel type (tant que tous les éléments du tableau sont de même type). Par conséquent, on peut parfaitement construire un tableau de pointeurs. Ainsi, « `int* arr[3]` » déclare un tableau de pointeurs vers des entiers[68].

On peut donc écrire par exemple[69] :

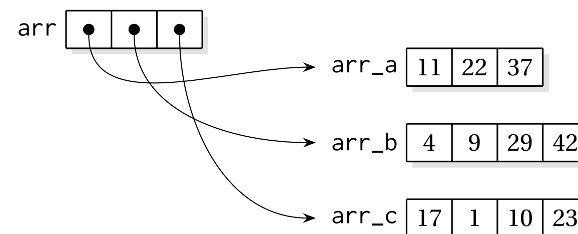
```
int a=37, b=42, c=54;
int* arr[3] = { &a, &b, &c };
```

« `arr[1]` » désigne donc le second élément du tableau `tab`, soit un pointeur vers un entier, contenant présentement l'adresse de la variable `b`. Ce pointeur peut être normalement déréférencé par l'opérateur « `*` » en écrivant « `*arr[1]` » pour obtenir le contenu de la variable `b` (il n'est pas besoin de parenthèses car « `[]` » a priorité sur « `*` »).

Mais, on l'a vu, un pointeur vers un entier peut également servir à désigner un *tableau* d'entiers. On peut donc utiliser la déclaration de `tab` pour mémoriser un ensemble de tableaux :

```
int arr_a[] = { 11, 22, 37 };
int arr_b[] = { 4, 9, 29, 42 };
int arr_c[] = { 17, 1, 10, 23 };

int* arr[3] = { arr_a, arr_b, arr_c };
```

Une telle structure, parfois appelée « *tableau Iliffe* », est assez fréquemment utilisée, mais n'est vraisemblablement pas considérée par le programme. Elle correspond donc au schéma suivant, où plutôt que de faire figurer les adresses des tableaux `arr_a`, `arr_b` et `arr_c` dans le tableau `arr`, nous avons utilisé des flèches indiquant quels objets ces adresses pointent :




68. Nous avons choisi de coller le symbole « `*` » à `int` pour souligner le fait que `arr` est un tableau de pointeurs (`int*`), mais la déclaration « `int *arr[3]` » désigne le *même* type, comme cela a été évoqué dans la section précédente.
69. Dans le cas d'une initialisation partielle du tableau, la valeur `0` est utilisée pour les cases pour lesquelles on n'a pas spécifié de vecteur, c'est donc `NULL` qui fait office de défaut.

Ainsi, « arr[1] » est un pointeur contenant l'adresse du tableau arr_b. Puisque arr[1] contient une adresse (de type int*), on peut utiliser à nouveau « [] » et écrire par exemple « arr[1][3] » pour obtenir la valeur 42 ! Le comportement ne semble donc pas très différent, dans son usage, de celui d'un tableau à deux dimensions d'entiers, mais les différences sont néanmoins importantes [70] :

- les données ne sont pas nécessairement placées contiguement en mémoire ;
- les « lignes » du tableau Iliffe (les tableaux arr[0], arr[1], etc.) ne sont pas forcément de même longueur ;
- il est possible que deux cases du tableau Iliffe contiennent la même adresse, autrement dit deux « lignes » peuvent faire référence au même tableau d'entiers, et il est donc possible que la modification d'un élément sur une « ligne » ait une influence sur d'autres lignes du tableau Iliffe ;
- si « int arr[][3] » peut servir pour désigner un tableau à deux dimensions lorsque l'on déclare une fonction, il faudra utiliser « int* arr[] » ou « int** arr » pour un tableau Iliffe.

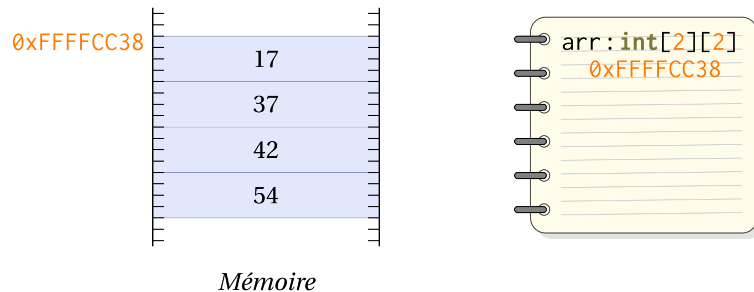
La confusion de cette struture avec les tableaux à deux dimensions est une source courante de « bugs » en langage C (et n'aident en rien la mauvaise réputation qu'ont les pointeurs dans le langage !) En particulier lorsqu'ils sont utilisés comme paramètres de fonction, car les deux types ne sont pas compatibles, même s'ils sont utilisés largement de la même façon !

Pour mieux illustrer la différence, examinons un peu comment ces deux structures sont rangées dans la mémoire...

Si le tableau à deux dimensions défini par exemple de la sorte

```
int arr[2][2] = { { 17, 37 },
                  { 42, 54 } };
```

correspond à la situation mémoire suivante :



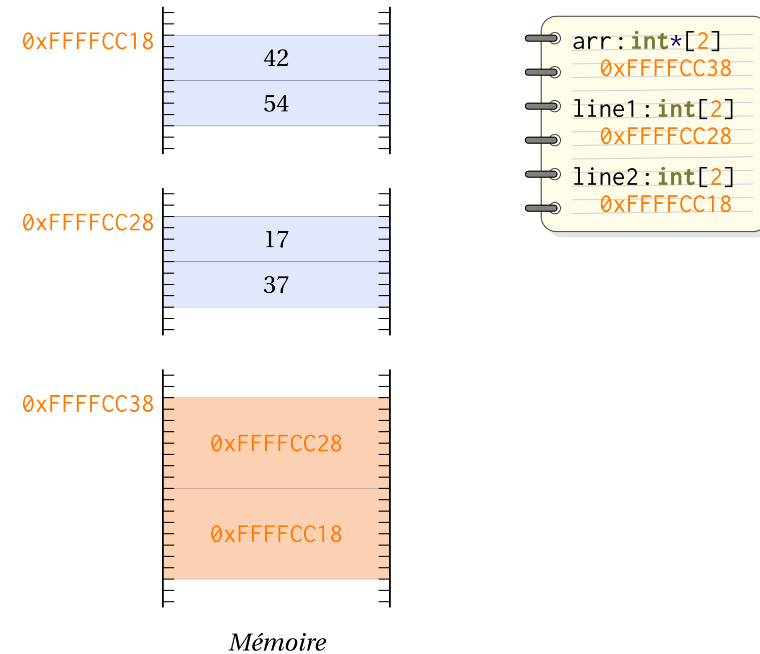

*Mémoire*

70. Les tableaux Iliffe ne sont pas si différents des listes de listes en Python (à ceci près qu'en C, les éléments doivent tous être de même type), tandis que les tableaux bidimensionnels ont davantage de parenté avec les tableaux numpy, même si cette correspondance n'est pas parfaite.

Le tableau Iliffe défini par

```
int line1[] = { 17, 37 };
int line2[] = { 42, 54 };
int* arr[] = { line1, line2 };
```

correspond, quant à lui, à la situation mémoire très différente illustrée ci-dessous [71] :




*Mémoire*

Et ce même si, à l'usage, arr[0] désigne dans les deux cas un tableau de deux entiers (plus précisément 17 et 37) et arr[0][1] donne dans les deux cas accès à la valeur 37.

# 5 Pointeurs de fonctions

## 5.1 Principe et déclaration

Les fonctions sont en C des objets très particuliers. Il n'est pas possible de déclarer une variable comme pouvant contenir une fonction. Toutefois, une fonction étant un morceau de code stocké en mémoire, on peut naturellement lui associer une adresse (le début

71. Le fait que les données ne soient pas *nécessairement* placées consécutivement en mémoire a été mis en évidence, même si dans le cas présent, les déclarations consécutives des trois variables auraient probablement conduit à des réservations de zones connexes en mémoire.

de la zone mémoire où résident les instructions machine correspondant à la fonction proprement dite).

Il est donc possible de déclarer des variables de type pointeurs contenant des adresses de fonctions. La syntaxe des pointeurs de fonctions n'est pas triviale, et dépasse largement le cadre de ce cours. Nous ne présenterons ici qu'un court exemple afin d'illustrer brièvement le pricipe.

Ainsi, la variable pointeur `ptr_func` définie ci-dessous pourra accueillir l'adresse de fonctions (telles que `round`, `sqrt`, `sin`...) prenant en argument un paramètre de type **`double`** et renvoyant un élément de type **`double`** :

```
double (*ptr_func)(double);
```

On peut mémoriser l'adresse d'une fonction, telle que la fonction mathématique `sin`, dans un tel pointeur en écrivant :

```
ptr_func = &sin;
```

Il est ensuite possible de déréférencer le pointeur `ptr_func` pour obtenir la fonction elle-même, puis éventuellement de l'utiliser, comme ci-dessous [72] :

```
double res = (*ptr_func)(3.14);
```

Comme les fonctions sont des objets très particuliers, il existe des raccourcis d'écriture qui facilitent la vie au programmeur (mais rendent malheuresement la syntaxe moins facile à comprendre). Tout d'abord, comme pour un tableau, si l'on mentionne une fonction avec juste son nom, sans parenthèses et argument, c'est l'adresse qui est considérée, donc on peut écrire :

```
ptr_func = sin;
```

Dans ce dernier cas, c'est bien l'*adresse* de la fonction `sin` qui est mémorisée dans la variable pointeur `ptr_func`. Et de la même façon, là encore, que pour les tableaux, la présence de parenthèses et d'arguments provoque en fait un déréférencement, de sorte que l'on peut également omettre « `*` » lors de l'utilisation, et écrire :

```
double res = ptr_func(3.14);
```

72. Les parenthèses autour de « `*ptr_func` » sont indispensables car les parenthèses utilisées pour le passage des paramètres ont une précédence plus élevée que « `*` ». Si l'on écrit « `*ptr_func(3.14)` », on sous-entend que `ptr_func` est une fonction prenant un **`double`** et renvoyant une adresse de type **`double`**`*`, et l'opérateur « `*` » déréférencerait cette adresse renvoyée par la fonction.

## 5.2 Fonctions comme paramètres

Comme les tableaux, les fonctions ne peuvent pas être utilisées directement comme paramètres d'une autre fonction. En revanche, il est possible de passer un *pointeur* vers une fonction comme paramètre.

On peut ainsi écrire des fonctions prenant en argument un pointeur vers une fonction, comme la fonction ci-dessous qui attends un tableau `tab` et sa taille `n`, ainsi qu'une fonction `foo` prenant un entier et renvoyant un entier, et qui applique `foo` à tous les éments du tableau :

```
void apply(int arr[], int n, int (*foo)(int)) {
    for (int i=0; i<n; ++i) {
        arr[i] = foo(arr[i]);
    }
}
```

On peut alors utiliser la fonction `apply` précédente afin d'appliquer la fonction `abs` (qui prend bien un argument de type **`int`** et renvoie un résultat de type **`int`**) à tous les éléments d'un tableau `arr` de taille `n` en écrivant simplement :

```
apply(arr, 15, abs);
```

Pour prendre un exemple un peu plus élaboré, il serait par exemple possible de réécrire la fonction de partition de Hoare sur les tableaux d'entiers, si l'on voulait utiliser une relation d'ordre $\preceq$ différente de l'ordre usuel sur les entiers, de façon à ce qu'elle prenne en argument une fonction prenant deux entiers `a` et `b` et renvoyant un booléen indiquant si `a`$\preceq$`b`, en écrivant :

```
int partition_Hoare(int arr[], int n, bool (*compare)(int, int)) {
    int pivot = arr[n/2];
    int l = 0, r = n-1;
    while (true) {
        while (compare(arr[g], pivot) && arr[g]!=pivot) { l++; }
        while (compare(pivot, arr[d]) && pivot!=arr[d]) { r--; }
        if (l <= r) {
            return r;
        }
        array_swap(arr, l, r);
        l++; r--;
    }
}
```

L'algorithme de tri rapide sera lui modifié de la façon suivante, là encore en ajoutant un

troisième argument permettant de passer la fonction servant à comparer deux éléments du tableau :

```c
void quicksort_Hoare(int arr[], int n, bool (*compare)(int, int)) {
    if (n >= 2) {
        int idx = partition_Hoare(arr, n, compare);
        quicksort_Hoare(arr, idx+1, compare);
        quicksort_Hoare(&arr[idx+1], n-idx-1, compare);
    }
}
```

Par exemple, pour trier un tableau `arr` de taille `n` en utilisant une fonction `compare` comparant deux entiers en fonction de leur valeur absolue :

```c
bool compare_abs(int a, int b) {
    return abs(a) <= abs(b);
}
```

il suffira d'appeler

```c
    quicksort(arr, n, compare_abs);
```

## Exercices

### Ex. 4.1 – Décryptage

On suppose `sizeof(int) = 4`, et on considère le programme suivant [73] :

```c
    int arr1[] = { 17, 42, 54, 37 }; // Adresse 0xCC30
    int arr2[] = { 11, 29, 22 };     // Adresse 0xCC24
    int a;                           // Adresse 0xCC20
    int b = 9;                       // Adresse 0xCC1C
    int *p1;                         // Adresse 0xCC10
    int *p2 = &a;                    // Adresse 0xCC08

    a = (arr1[3]-arr2[1]);       // a <-
    b /= *p2/3;                  // b <-
    p1 = &arr1[1];               // p1 <-
    p1[1] = *p1/2;               //
    *p1 -= p1[-1] + p2[0];       //
    p1 = arr2;                   //
    *p1 = *p2;                   //
    p2 = &p1[*p2/b];             //
```

Déterminer l'effet de chaque instruction du programme sur la mémoire.

### Ex. 4.2 – Copies de tableau

**1.** Proposer une fonction `copy(int src[], int dst[], int n)` prenant en argument deux pointeurs vers des tableaux, et copiant les `n` premiers éléments de l'un vers l'autre.

**2.** Préciser ce qu'il se passe si on écrit :

```c
    int arr[] = { 1, 2, 3, 4, 5 };
    copy(arr, &arr[1], 4);
```

**3.** Comment modifier la fonction `copy` pour obtenir dans `arr` la séquence d'entiers $1,1,2,3,4$ après la copie, si ce n'est pas le cas ? Est-ce une meilleure solution [74] ?

73. Les adresses indiquées, en hexadécimal, pour les différentes variables, ont été raccourcies pour simplifier leur manipulation et ne correspondent pas aux adresses usuellement observées sur un ordinateur courant.

74. Il n'est généralement pas possible, en C, de savoir si deux pointeurs désignent des zones d'adresses qui se recouvrent, car comparer des adresses obtenues à partir d'objets distincts est un cas de comportement indéterminé. La seule solution pour garantir que la zone désignée par `dst` après l'appel contienne la même chose que la zone désignée par `src` avant l'appel passe par l'utilisation de mémoire temporaire.

## Ex. 4.3 – Somme prescrite

Proposer une fonction `sum_possible(int arr[], int n, int target)` prenant en argument un tableau d'entiers `arr` dont les éléments sont rangés par ordre croissant, sa taille `n` et un entier `target`, et renvoyant, en temps linéaire, un booléen indiquant s'il existe deux éléments du tableau dont la somme est égale à `target`.

## Ex. 4.4 – Tri récursif

On suppose disposer d'une fonction prenant en argument un pointeur `arr` vers un tableau (ou un morceau de tableau) et sa taille `n`, et faisant remonter en premier place (à la position `arr[0]` donc) le plus petit élément d'un tableau sur le principe d'une « bulle », implémentée de la façon suivante :

```
void bubble(int arr[], int n) {
    int elem = arr[n-1];
    for(--n; n>0; --n) {
        if (arr[n-1] < elem) {
            arr[n] = elem;
            elem = arr[n-1];
        } else {
            arr[n] = arr[n-1];
        }
    }
    arr[0] = elem;
}
```

**1.** En utilisant la fonction précédente, proposer une version itérative d'une fonction `bubblesort(int arr[], int n)`.

**2.** Proposer une version récursive de la même fonction (la fonction ne comportera pas de boucle, mais pourra s'appeler elle-même).

## Ex. 4.5 – Concaténation de tableaux

Proposer une fonction `int* concat(int* arrays[], int sizes[], int n)` prenant en argument un tableau de pointeurs `arrays` et un tableau d'entier `sizes`, tous deux de taille `n`, de sorte que `arrays[i]` soit un pointeur vers un tableau d'entier de taille `sizes[i]`, et allouant puis renvoyant un tableau correspondant à la concaténation des `n` tableaux `arrays[i]`.

## Ex. 4.6 – Maximums locaux

Proposer une fonction `int* local_max(int arr[], int n, int k)` prenant en argument un tableau d'entiers `arr` et sa taille `n`, ainsi qu'un entier `k` $\in [\![1\,..\,n]\!]$, et qui alloue un tableau de taille $n+1-k$, le remplit de façon à ce que dans la case d'index `i` on obtienne la valeur

$$\max_{i \leqslant j < i+k} \texttt{arr}[j]$$

et renvoie ce tableau.

On recherche une complexité temporelle linéaire en `n` indépendante de `k` (en particulier, si `k` est de l'ordre de `n/2`, on souhaite que la complexité reste linéaire).

## Ex. 4.7 – Algorithme du drapeau

On considère un tableau `arr` de taille `n` contenant des entiers positifs. On cherche à réorganiser les éléments dans le tableau de sorte qu'apparaissent en premier les entiers congrus à 0 modulo 3, puis ceux congrus à 1 modulo 3, et enfin ceux congrus à 2 modulo 3. On souhaite par ailleurs conserver une complexité linéaire, et ne tester la congruence qu'une seule et unique fois pour chaque entier.

En s'inspirant des algorithmes de partition en place proposés dans le cours, écrire un algorithme [75] répondant à ces critères. On précisera clairement l'invariant de boucle choisi, en illustrant l'état du tableau lorsque la réorganisation des éléments est en cours.

## Ex. 4.8 – Rotation de tableau

Proposer une fonction `void rotate(int arr[], int n, int k)` prenant en argument un tableau `arr`, sa taille `n` et un entier `k` $\in [\![1\,..\,n-1]\!]$ et effectuant une « rotation » en place des éléments du tableau. On s'efforcera de proposer une solution de complexité temporelle linéaire en `n` (et indépendante de `k`) et de complexité spatiale constante.

## Ex. 4.9 – Tableau quasi-trié

On considère un tableau d'entiers `arr` de taille `n` vérifiant `arr[0]`$\geqslant$`arr[n-1]` et quasi-trié par ordre croissant, c'est-à-dire qu'il existe au plus un *unique* $i \in [\![0\,..\,n-2]\!]$ tel que

$$\texttt{arr}[i] > \texttt{arr}[i+1]$$

75. Cet algorithme a été proposé par Edsker Wybe Dijkstra, et porte ce nom car, dans sa version originale, il propose de réordonner des objets rouges, blancs et bleus de sorte qu'ils se retrouvent dans cet ordre (les couleurs du drapeau néerlandais, patrie de Edsker Wybe Dijkstra) à la fin du réarrangement.

**1.** Proposer une fonction `int min(int arr[], int n)` qui renvoie, en temps logarithmique en `n`, le plus petit élément du tableau.

**2.** Proposer une fonction `int index(int arr[], int n, int elem)` qui renvoie un index `i`$\in [\![0 \mathinner{..} n-1]\!]$, s'il existe, tel que `arr[i] == elem`, et $-1$ sinon.

## Ex. 4.10 – Orbites de permutations

On s'intéresse à une permutation $\sigma$ de $E_n = [\![0 \mathinner{..} n-1]\!]$. On rappelle qu'une permutation de $E_n$ est une bijection de $E_n$ dans lui-même. On représente $\sigma$ par un tableau `sigma` de taille `n` tel que `sigma[i]` contienne l'image $\sigma(i)$ de $i$ pour la permutation $\sigma$.

**1.** Proposer une fonction `int* invert(int sigma[], int n)` prenant en argument un tableau `sigma` représentant une permutation $\sigma$ de $E_n$ et sa taille `n`, et allouant et renvoyant un tableau contenant son inverse $\sigma^{-1}$.

La période d'un indice $i$ pour la permutation $\sigma$ est définie comme le plus petit entier $k$ non nul tel que $\sigma^k(i) = i$.

**2.** Proposer une fonction `int period(int sigma[], int i)` prenant en argument un tableau `sigma` représentant une permutation de $E_n$ et un entier $i \in E_n$ et renvoyant sa période.

L'orbite de $i$ pour la permutation $\sigma$ est l'ensemble des indices $j$ tels qu'il existe $k$ avec $\sigma^k(i) = j$.

**3.** Écrire une fonction `bool same_orbit(int sigma[], int i, int j)` prenant en argument un tableau `sigma` représentant une permutation de $E_n$ et deux entiers $i$ et $j$ de $E_n$ et renvoyant un booléen indiquant si $j$ est dans l'orbite de $i$ [76]. Quelle est sa complexité ?

**4.** Proposer une fonction `int* all_periods(int sigma[], int n)` prenant en argument un tableau `sigma` représentant une permutation de $E_n$ ainsi que l'entier `n`, et allouant puis renvoyant un tableau `periods` de taille $n$ tel que `periods[i]` contienne la période de l'élément `i`. On souhaite obtenir une complexité temporelle **linéaire** en $n$ ($O(n)$).

**5.** Justifier qu'il existe une infinité d'entiers $k \geqslant 1$ tel que $\sigma^k$ soit la fonction identité, et proposer un algorithme permettant de calculer le plus petit de tels $k$.

## Ex. 4.11 – Paire de tableaux

On suppose disposer de deux tableaux d'entiers `arr1` et `arr2`, de tailles respectives `n1` et `n2`, tous deux triés par ordre croissant.

76. Et réciproquement, il s'agit en fait d'une relation d'équivalence.

**1.** Proposer une fonction `nb_min(int arr1[], int n1, int arr2[], int n2)` déterminant la quantité suivante :

$$\left|\left\{(i,j) \in [\![0 \mathinner{..} n_1-1]\!] \times [\![0 \mathinner{..} n_2-1]\!] \;\middle|\; \texttt{arr1[i]} \leqslant \texttt{arr2[j]}\right\}\right|$$

**2.** En supposant, dans un premier temps, que tous les éléments de `arr1` sont distincts, et que tous les éléments de `arr2` sont distincts, proposer une fonction `int dupes(int arr1[], int n1, int arr2[], int n2)` renvoyant le nombre d'entiers présents à la fois dans `arr1` et `arr2` en temps linéaire en `n1+n2`.

**3.** Même question si on ne considère plus les éléments distincts dans chaque tableau (si le premier tableau contient deux fois 42 et le second trois fois, on considérera qu'il y a six paires).

**4.** Proposer une fonction `int median(int arr1[], int n1, int arr2[], int n2)` renvoyant une médiane de l'ensemble des éléments présents dans les deux tableaux de la façon la plus efficace possible.

## Ex. 4.12 – Sélection rapide (quickselect)

On souhaite déterminer, pour un tableau `arr` de taille `n`, le k^e^ plus petit élément du tableau sans avoir à explicitement trier le tableau. Pour ce faire, on propose d'utiliser l'algorithme de *sélection rapide*, qui s'inspire largement de l'algorithme du tri rapide. On supposera disposer d'une fonction `int partition(int arr[], int n)` fonctionnant comme dans le cours.

**1.** On applique la fonction `partition` sur le tableau. Où faut-il rechercher l'élément souhaité en fonction de l'index renvoyé par la fonction `partition` ?

**2.** En déduire un algorithme permettant d'obtenir le k^e^ plus petit élément du tableau, et proposer une fonction `int selection(int arr[], int n, int k)` déterminant cet élément (on supposera que la fonction `selection` peut déplacer les éléments du tableau passé en argument comme elle le souhaite).

**3.** Quelle est la complexité dans le pire et dans le meilleur des cas ?

**4.** Comment obtenir une médiane d'un tableau avec la fonction `selection` ? On pourra se poser la question de l'obtention de la médiane inférieure et de la médiane supérieure du tableau.

## Ex. 4.13 – Médiane des médianes

Une médiane d'un tableau est évidemment un pivot idéal pour un algorithme de tri rapide ou de sélection rapide. Malheureusement, dans le pire des cas, la complexité de

l'algorithme de sélection rapide, tel que proposé précédemment, est trop grande pour que l'on puisse l'utiliser dans le choix du pivot.

Cependant, tout n'est pas perdu! Nous allons montrer que l'on peut écrire une fonction `int select_mom(int arr[], int n, int k)` renvoyant le k$^e$ plus petit élément du tableau `tab` en temps linéaire. Pour ce faire, applique les idées suivantes :

- si le tableau contient moins de cinq éléments, on calcule explicitement le k$^e$ élément;
- sinon :
  - — on décompose le tableau en ensembles de cinq éléments (si le tableau ne comporte pas un nombre d'éléments divisible par cinq, on complète le cinquième groupe avec des éléments dupliqués, en utilisant des éléments de la fin de l'avant-dernier tableau pour compléter le dernier tableau);
  - — on calcule les médianes de chacun de ces ensembles de cinq éléments, et on appele récursivement la fonction `select_mom` sur l'ensemble des médianes calculées pour déterminer al médiane de cet ensemble;
  - — on se sert de cette valeur comme pivot pour partitionner le tableau et en déduire le k$^e$ plus petit élément en appliquant la même méthode que dans le cas de l'algorithme de sélection rapide.

**1.** Justifier que la valeur renvoyée, lors de l'appel récursif, est plus grande qu'au moins 30% des éléments du tableau, et plus petite qu'au moins 30% des éléments du tableau (on supposera que la taille du tableau est un multiple de 5).

**2.** En réfléchissant à la séquence des appels effectués par la fonction et la taille des tableaux lors de chaque appel, montrer que le coût $C(n)$ pour un tableau de taille $n > 5$ vérifie $C(n) = C(\lceil n/5 \rceil) + C(\lceil 7n/10 \rceil) + O(n)$.

**3.** En déduire que la complexité en temps, dans le pire des cas, de `select_mom` est bien linéaire.

Dans la pratique, cependant, les cas défavorables pour les algorithmes de tri et de sélection rapides sont trop rares pour que l'utilisation de l'algorithme de la médiane des médianes soit réellement utile, et il pénalise les autres cas en augmentant le préfacteur.

## Ex. 4.14 – Médiane d'un tableau de taille 3, 4 ou 5

**1.** Proposer un algorithme permettant de déterminer la médiane d'un tableau de trois entiers, effectuant le minimum possible de comparaisons dans le pire des cas, et proposer une fonction `int median_of_3(int arr[])` implémentant cet algorithme.

**2.** Proposer de même un algorithme donnant une médiane d'un tableau de quatre éléments, et la médiane d'un tableau de cinq élements [77]).

77. Ce dernier cas est relativement complexe... Le minimum de comparaisons est 6. Pour une liste d'algorithmes identifiant efficacement le $k^e$ plus petit élément parmi $n$ pour de petites valeurs de $n$, on pourra consulter par exemple `www.cs.hut.fi/~cessu/selection`.

## Ex. 4.15 – Tri de Shell

On suppose ici vouloir trier un tableau `arr` de `n` entiers relatifs. Le tri de Shell (proposé par Donald Shell en 1959) est une amélioration du tri par insertion visant à limiter le gaspillage en temps d'exécution dû aux « tortues », c'est-à-dire aux éléments qui seront déplacés un nombre de fois de l'ordre du nombre $n$ d'éléments dans le tableau avant de trouver leur position finale.

Pour un incrément entier donné `step`, on considère l'opération suivante : pour tous les décalages entiers `offset` vérifiant $0 \leqslant$ `offset` $<$ `step`, on effectue un tri par insertion sur les éléments du tableau dont l'index est de la forme `offset`$+k\times$`step`. Cela revient donc à effectuer `step` tris par insertion sur autant de sous-tableaux du tableau à trier.

Par exemple, pour le tableau $5,4,7,8,2,3,0,9,1,6$, et pour `step=3`,

| 5 | 4 | 7 | 8 | 2 | 3 | 0 | 9 | 1 | 6 |
|---|---|---|---|---|---|---|---|---|---|

On considère les trois « sous-tableaux » :

| 5 | 8 | 0 | 6 |
|---|---|---|---|

| 4 | 2 | 9 |
|---|---|---|

| 7 | 3 | 1 |
|---|---|---|

On trie ces derniers, en utilisant un tri par insertion

| 0 | 5 | 6 | 8 |
|---|---|---|---|

| 2 | 4 | 9 |
|---|---|---|

| 1 | 3 | 7 |
|---|---|---|

Ce qui conduit, après avoir réuni les sous-tableaux, au résultat suivant

| 0 | 2 | 1 | 5 | 4 | 3 | 6 | 9 | 7 | 8 |
|---|---|---|---|---|---|---|---|---|---|

Bien que l'on ait, sur cet exemple, « extrait » explicitement les sous-tableaux afin de faciliter la compréhension de l'algorithme, il n'est pas indispensable de le faire, les opérations peuvent être directement effectuées sur le tableau original.

**1.** Proposer une fonction `void partial_sort(int arr[], int n, int step)` qui effectue un tri par insertion sur chacun des `step` sous-tableaux tels que définies précédemment. On s'efforcera de n'utiliser que deux boucles imbriquées (il n'est pas nécessaire de faire une boucle supplémentaire pour chaque sous-tableau), et on précisera un invariant de boucle.

**2.** Quelle est la complexité en temps, dans le pire des cas, de la fonction `partial_sort`, en fonction de la taille $n$ du tableau `arr` et de `step`, en supposant que les comparaisons entre les éléments se font en temps constant $O(1)$ ?

Le tri de Shell effectue plusieurs fois l'opération précédente, pour différentes valeurs de `step`, décroissantes, en terminant avec `step=1`.

Dans la version originale de l'algorithme proposé par D. Shell, les valeurs successivement utilisées pour `step` étaient :

$$\left\lfloor \frac{n}{2} \right\rfloor, \quad \left\lfloor \frac{n}{2^2} \right\rfloor, \quad \left\lfloor \frac{n}{2^3} \right\rfloor, \dots \quad , \quad \left\lfloor \frac{n}{2^p} \right\rfloor = 1$$

Par exemple, pour le tableau `{5, 4, 7, 8, 2, 3, 0, 9, 1, 6}` contenant $n = 10$ éléments, on utilisera successivement les incréments `step=5`, `step=2` et `step=1`.

**3.** Déterminer la relation entre la taille $n$ du tableau `arr` et le nombre d'étapes $p$.

Les états successifs du tableau avant et durant la succession des trois appels à `tri_partiels` seront les suivants :

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| 5 | 4 | 7 | 8 | 2 | 3 | 0 | 9 | 1 | 6 |
| 3 | 0 | 7 | 1 | 2 | 5 | 4 | 9 | 8 | 6 |
| 2 | 0 | 3 | 1 | 4 | 5 | 7 | 6 | 8 | 9 |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |

**4.** Proposer une fonction `void shell_sort(int arr[], int n)` qui trie par ordre croissant le tableau `arr` fourni en argument, en utilisant la suite des `step` proposée originalement par `D. Shell`.

**5.** Justifier qu'après l'appel à la fonction `shell_sort`, le tableau est correctement trié.

**6.** Proposer une estimation de la complexité du tri de Shell dans le pire des cas ?

Le choix des incréments successifs a une influence très importante sur l'efficacité (et la complexité) du tri de Shell. À l'heure actuelle, la suite 701, 301, 132, 57, 23, 10, 4, 1, déterminée empiriquement, semble la plus efficace. La complexité qu'il est possible d'atteindre avec une suite d'incréments bien choisie n'est pas connue avec certitude, et reste l'objet de recherches.

Même si ce tri n'est pas aussi efficace que d'autres méthodes, et présente des inconvénients (tri instable, efficacité moindre vis-à-vis du cache, etc.), la simplicité de sa mise en œuvre en font un tri très utile dans des systèmes aux ressources limitées (en mémoire ou en terme de récursion), par exemple dans certains systèmes embarqués.

# 5 Introduction au langage OCaml

> « Humans are allergic to change. They love to say, “We’ve always done it this way.” I try to fight that. That’s why I have a clock on my wall that runs counterclockwise. »
>
> — *Grace Hopper*

## 1 Présentation de la famille Caml

### 1.1 Philosophie du langage

Le langage Caml est un langage créé par INRIA en 1985. La version actuelle du langage, OCaml, activement développée et largement utilisée, est celle figurant au programme de la filière MPI. La documentation officielle du langage peut être obtenue sur le site d’INRIA (à l’adresse `caml.inria.fr`).

Au cours des dernières années, plusieurs façons d’envisager la programmation ont émergé. Il est difficile de classifier les différentes approches mises en œuvre par les différents langages, tant les frontières sont poreuses, mais on peut quand même identifier quelques styles distincts.

Le style utilisé par les tous premiers langages est qualifié de *programmation impérative*. Il est basé sur la notion de machine abstraite équipée d’une mémoire, le programme étant une suite d’instructions modifiant l’état de la mémoire. La gestion de la mémoire est, souvent, en grande partie à la charge du programmeur. C’est le style de programmation que nous avons mis en œuvre depuis le début de ce cours avec le langage C.

Parallèlement s’est développée la notion de *programmation fonctionnelle*, qui repose, quant à elle, principalement sur la définition et l’évaluation de fonctions. Ce style évite tant que faire ce peu les variables et mécanismes d’affectation. Il n’est donc pas besoin de se soucier de la façon dont la mémoire est gérée. La programmation fonctionnelle trouve ses racines dans le *lambda calcul*, un système formel de calcul proposé par Alonzo Church dans les années 1930. On considère généralement que le premier langage informatique exploitant ces idées est le langage LISP, apparu à la fin des années 1950.

D’autres approches ont également émergé, comme la *programmation objet*, centrée autour des données que l’on manipule, auxquelles sont directement associées des méthodes agissant sur ces données, la *programmation déclarative*, ou bien encore la *programmation événementielle*. Dans la pratique, beaucoup de langages sont qualifiés de *multi-paradigmes*, c’est-à-dire qu’ils mélangent les possibilités offertes par plusieurs des styles de programmation que nous venons d’évoquer.

OCaml est un langage avec un fort héritage fonctionnel, et dans la mesure où cette façon de travailler diffère avec ce que l’on a vu précédemment, c’est cette philosophie que l’on va mettre en avant dans un premier temps. Nous verrons cependant un peu plus tard que le langage permet également de faire de la programmation impérative, et la lettre « O » dans le nom « OCaml » est l’initiale de « objective », indication claire que la programmation objet est également un aspect important du langage.

### 1.2 Compilateurs et interpréteurs OCaml

En principe, OCaml est un langage compilé. Dans la distribution INRIA officielle, sont fournis deux compilateurs distincts. Le premier, baptisé « `ocamlopt` », produit à partir d’une source OCaml du code machine qui pourra être exécuté directement par le processeur. Il s’invoque de façon similaire aux compilateurs C que nous avons déjà utilisé. Par exemple, pour construire un exécutable `my_prog` à partir d’une source `my_prog.ml`, on écrira

```
ocamlopt my_prog.ml -o my_prog
```

Le second compilateur, nommé « `ocamlc` », s’utilise de façon similaire mais produit du code destiné à une machine virtuelle [1] plutôt qu’au processeur, code que ladite machine virtuelle pourra ensuite interprêter [2], sur un modèle quelque peu similaire aux langages de la famille `Java`. L’interpréteur inclus avec la distribution officielle s’appelle « `ocamlrun` ».

Les deux solutions ont leurs avantages et inconvénients. La compilation en bytecode est plus rapide, et le bytecode produit peut être exécuté sur des machines au hardware différent sans besoin de recompilation tant que l’on dispose d’une implémentation de `ocamlrun` compatible. En revanche, l’exécution est en général un peu moins véloce.

On dispose également d’un outil permettant d’interprêter directement le code source, nommé « `ocaml` » dans la distribution officielle. On peut ainsi exécuter sans compilation le contenu d’un fichier `my_prog.ml` en écrivant « `ocaml my_prog.ml` ». Mais surtout, cet interpréteur peut fonctionner de manière interactive, ce qui permet d’exécuter les commandes une à une et d’étudier au fur et à mesure les résultats qu’elles fournissent.

On peut accéder à ce mode interactif directement depuis la ligne de commande en utilisant la commande « `ocaml` », mais il existe des environnements qui permettent d’en faciliter l’utilisation, tels que WinCaml et MacCaml [3] (respectivement pour Windows et

1. On parle généralement, dans ce cas, de *bytecode*.
2. Les binaires produits par `ocamlc` sont généralement directement exécutables, l’invocation de la machine virtuelle étant inclus dans le binaire produit.
3. Que l’on trouvera à l’adresse `http://jean.mouric.pagesperso-orange.fr/`. Attention, OCaml et Caml Light sont tous deux disponibles, il faudra bien choisir le bon langage via le menu *CamlTop*.

OS-X), l'outil utop, le module « Ocaml Platform » de VS Code, ou bien encore le « Tuareg mode » de l'éditeur emacs.

Dans la suite du cours, nombre d'exemples courts seront présentés au travers de ce mode interactif, car il permet de voir directement les résultats obtenus par l'évaluation des expressions étudiées. Dans ce cadre, les commandes envoyées à l'interpréteur sont les lignes précédées du symbole « # », les réponses étant reportées juste en-dessous.

## 2 Premiers pas

### 2.1 Calculs et expressions

Les objets que l'on manipulera le plus souvent en OCaml sont les mêmes que ceux que l'on a déjà rencontré en C. Il est donc naturel de retrouver des types similaires à ceux déjà vus. OCaml dispose donc, entre autres choses, d'un type `int` correspondant aux entiers, d'un type `float` pour les flottants [4] et d'un type `bool` pour les booléens.

Les entiers (`int`) permettent de représenter des valeurs entières, positives ou négatives, comprises entre deux valeurs limites auxquelles il est possible de faire référence via les noms min_int et max_int [5]. On dispose des opérateurs usuels d'addition « + », de soustraction « - », de multiplication « * », de division entière « / » ainsi que d'un opérateur permettant d'obtenir le reste d'une division entière, « mod ».

Effectuer un calcul dans l'interface interactive d'OCaml est simple : il suffit d'entrer une expression que l'on termine par un double point-virgule. Ce double point-virgule ne fait pas à proprement parler du langage OCaml. Il indique simplement, lorsque l'on utilise l'interface interactive, que l'on a terminé d'entrer une expression et que l'on souhaite que l'interpréteur l'évalue. Par exemple [6] :

```
# 17 / 3;;
- : int = 5

# 42 mod 17;;
- : int = 8
```

Intéressons-nous aux réponses d'OCaml. Elles contiennent trois éléments. Au centre est précisé le type du résultat, ici des entiers (`int`). Le résultat proprement dit se trouve à droite du signe égal (respectivement 5 et 8). Nous reviendrons plus tard sur l'élément à gauche des deux points.

Naturellement, les priorités et règles concernant l'associativité des opérateurs sont respectées, les mêmes que celles du le langage C. Pour effectuer les opérations dans un ordre différent, il est possible d'utiliser des parenthèses :

```
# 2 + 3 * 5;;
- : int = 17

# (2 + 3) * 5;;
- : int = 25
```

Précisons que, contrairement au C, il n'est pas illégal de dépasser, dans un calcul, la valeur max_int [7], mais le résultat obtenu sera complètement différent de celui attendu, et ce de façon totalement silencieuse. On parle de *débordement*, et cela peut avoir des conséquences néfastes si l'on n'y prend pas garde. Par exemple,

```
# max_int;;
- : int = 4611686018427387903

# 4611686018427387903 + 1;;
- : int = -4611686018427387904

# 4611686018427387903 * 2;;
- : int = -2
```

Il reste illégal de diviser par zéro (que ce soit avec « / » ou avec « mod »), mais cela provoquera une erreur [8] et non une valeur arbitraire ou un comportement arbitraire du programme. il n'y a pas de comportement indéfini en OCaml.

Travailler avec des flottants peut apparaître au premier abord un brin plus difficile. En effet, la solution qui pourrait sembler « naturelle » provoque une erreur :

```
# 1.41 + 3.14;;

Characters 2-6:
  1.41 + 3.14;;
  ^^^^
Error: This expression has type float but
       an expression was expected of type int
```

4. Des flottants sur 64 bits, donc en double précision, qui correspondent bien au type `double` du langage C, et non au type `float` en C qui représente des flottants sur 32 bits.

5. Il s'agit généralement de l'intervalle $[\![-2^{-30} \,..\, 2^{30}-1]\!]$ ou de l'intervalle $[\![-2^{-62} \,..\, 2^{62}-1]\!]$, soit des entiers sur 31 bits ou 63 bits. Cette taille quelque peu étrange est liée à la représentation interne des objets en OCaml.

6. Rappelons que le symbole « # » ne fait pas partie de l'expression, il s'agit simplement de l'invite de commande affichée par l'interpréteur.

7. Ou de descendre « en-dessous » de min_int.

8. En fait une « exception », que le programme pourra gérer au besoin.

La raison de cette erreur est que OCaml utilise des opérateurs différents pour *chaque* type qu'il peut manipuler. L'opérateur « `+` », utilisé pour sommer des entiers, ne peut donc pas sommer des flottants. Les quatres opérateurs courants pour les flottants [9] sont représentés par les symboles usuels, mais suivis d'un point (soit « `+.` », « `-.` », « `*.` » et « `/.` »). On dispose aussi d'un opérateur d'exponentiation, « `**` », cette fois sans point [10].

```
# 1.41 +. 3.14 ** 2.0;;
- : float = 11.2696
```

On dispose également de nombreuses fonctions mathématiques courantes (telles que `sqrt`, `exp`, `log`, `sin`, `cos`, `tan`, `asin`, `acos`, `atan`...), fonctions qui n'ont pas besoin d'être importées. Précisions que, comme en C, le logarithme fourni à travers la fonction `log` est le logarithme *népérien* et non décimal [11].

```
# log 2.0e4;;
- : float = 9.9034875525361272
```

Remarquons dès maintenant un point sur lequel nous reviendrons : les paramètres des fonctions suivent celles-ci sans qu'il soit nécessaire d'utiliser de parenthèses, excepté lorsqu'elles sont requises [12] pour que l'ordre des calculs soit celui désiré (les fonctions ont une précédence plus élevée que les opérateurs usuels, et l'associativité se fait de la gauche vers la droite).

```
# cos 3.1415926535897932 /. 3.0;;
- : float = -0.33333333333333331

# cos (3.1415926535897932 /. 3.0);;
- : float = 0.50000000000000011

# sqrt (exp 1.0);;
- : float = 1.6487212707
```

Pour les expressions booléennes, les notions de « vrai » et « faux » sont représentées en OCaml comme en C par « `true` » et « `false` » (sans majuscule). Comme en C, on note [13] `&&` l'opérateur logique « et », et `||` l'opérateur logique « ou ». L'opérateur unaire `not` permet d'obtenir la négation d'une expression booléenne. L'opérateur `not` a la plus grande priorité, suivi de `&&` et enfin de `||` :

```
# true || true && false;;
- : bool = true

# not false || true;;
- : bool = true
```

Mais on peut naturellement utiliser des parenthèses pour changer cet ordre d'évaluation :

```
# not (false || true);;
- : bool = false
```

Pour « construire » des booléens, le langage OCaml dispose de différents opérateurs pour comparer deux éléments : on écrira `=` pour tester l'égalité, `<>` pour tester la « non-égalité », et enfin `<` `<=` `>` `>=` pour ce qui est des comparaisons. Ces opérateurs ont priorité sur les opérateurs logiques. Ces opérateurs, dits *polymorphes*, acceptent n'importe quels types :

```
# true = false;;
- : bool = false

# 37 <> 42;;
- : bool = true

# 3.14 <= 1.41;;
- : bool = false
```

Pour qu'une comparaison soit valide, cependant, il faut que les deux éléments comparés soient impérativement de même type :

```
# 3 = 3.0;;

Characters 6-9:
  3 = 3.0;;
      ^^^
Error: This expression has type float but
       an expression was expected of type int
```

**On notera que l'égalité s'écrit avec un unique signe égal, et sa négation avec `<>`.** On aura tôt fait de remarquer que `==` et `!=` existent, et semblent à première vue fonctionner de la même façon, mais nous verrons qu'il s'agit d'opérateurs d'identité et non d'égalité (ils correspondent aux opérateurs `is` et `is not` en Python), et il convient de veiller à ne pas les mélanger.

Comme en C, l'évaluation des expressions booléennes est paresseuse, c'est-à-dire qu'elle

9. Les priorités et précédences usuelles, déjà vues en C, sont respectées, y compris pour l'exponentiation.

10. Il n'y a pas de raison d'en mettre un puisqu'il n'existe pas, essentiellement pour des raisons historiques, d'opérateur d'exponentiation travaillant sur les entiers.

11. Il existe une fonction `log10` fournissant le logarithme décimal.

12. Pour garder ce point à l'esprit, nous glisserons toujours une espace entre le nom d'une fonction et la parenthèse ouvrante, même si celle-ci n'est pas requise par le langage, afin de souligner que les parenthèses n'ont qu'un rôle de contrôle de l'ordre d'évaluation non lié à l'appel de fonction.

13. Signalons que OCaml tolère **or** à la place de `||` et `&` à la place de `&&`, mais ni **and**, ni `|` car utilisés pour d'autres usages. Nous éviterons ces notations dans la suite pour les risques de confusions qu'elles comportent.

cesse dès que l'on a pu déterminer avec certitude le résultat, sans évaluer la totalité des expressions, comme le montrent ces exemples (les expressions contenant une division par zéro dans les deux premières expressions ne sont en particulier jamais évaluées) :

```
# 2 < 3 || (1/0) == 42;;
- : bool = true

# 2 > 3 && (1/0) == 42;;
- : bool = false

# 2 > 3 || (1/0) == 42;;
Exception: Division_by_zero.
```

## 2.2 Typage fort

Si l'utilisation d'opérateurs différents en fonction du type peut paraître contraignante, ce choix a été fait pour permettre à OCaml de déterminer automatiquement, aussi souvent que possible, les types des opérandes. Par exemple, lorsque l'on écrira « `x + y` », OCaml pourra en déduire que `x` et `y` sont à des entiers.

Conjointement à ces limitations sur les opérateurs, OCaml utilise ce que l'on appelle un *typage fort*, c'est-à-dire qu'il n'essaiera jamais, de lui même, de changer le type d'un objet pour pouvoir réaliser une opération. En particulier, le typage fort d'OCaml fait qu'il ne sera pas possible d'additionner deux valeurs de types différents :

```
# 3.0 +. 2;;

Characters 9-10:
  3.0 +. 2;;
         ^
Error: This expression has type int but
       an expression was expected of type float
```

De même, les fonctions attendent un objet d'un type donné, et le langage n'opérera pas de lui-même de conversion :

```
# cos 0;;

Characters 7-8:
  cos 0;;
      ^
Error: This expression has type int but
       an expression was expected of type float
```

Le langage Python a également un typage fort (on peut s'en convaincre en constatant que « `range(2.0)` » est refusé par l'interpréteur, car `range` attend un paramètre entier, et Python ne peut convertir de sa propre initiative un flottant en entier), mais comme c'est un langage *polymorphe* (lors d'une affectation, le type de l'objet associé à un nom peut changer, et les types des arguments des fonctions ne sont pas imposés) et que les fonctions et opérateurs usuels s'accomodent de types différents [14], ce n'est pas toujours une évidence.

Le langage C a, quant à lui, un typage plus faible : si l'on passe un entier à une fonction attendant un flottant par exemple, il effectuera spontanément, sans avertissement, une conversion de type. Pour les opérateurs entre valeurs numériques de type différents, il existe des règles explicites de conversion prévues par le langage. Quelques conversions d'un type de pointeur à un autre peuvent également être permises, parfois avec un avertissement à la compilation. Mais les conversions implicites restent relativement limitées en comparaison d'autres langages.

Pour résoudre le « problème », il est heureusement possible de convertir un type en un autre, à condition de le faire explicitement. Ainsi, la fonction `float_of_int` permet de convertir un entier en flottant :

```
# float_of_int 2;;
- : float = 2.0

# 3.0 +. float_of_int 2;;
- : float = 5.0

# cos (float_of_int 0);;
- : float = 1.
```

De même, `int_of_float` permet d'effectuer la conversion inverse (en tronquant la valeur réelle si nécessaire) :

```
# 2 + int_of_float 3.0;;
- : int = 5

# 2 + int_of_float 3.5;;
- : int = 5
```

Si le typage fort, et le fait d'avoir des opérateurs distincts pour chaque type, peuvent sembler des contraintes importantes (elles le sont indubitablement par moment), ce sera un allié de poids lorsqu'il s'agira d'écrire des programmes corrects.

14. En particulier, selon le type des opérandes, l'opérateur `+` fera par exemple spontanément appel à différentes fonctions travaillant spécifiquement avec un type ou un autre, telles que `int.__add__`, `float.__add__`, `str.__add__`, `list.__add__`...

# 3 Définitions

## 3.1 Définitions globales

On ne déclare pas, en OCaml, des variables comme on le fait en C. En fait, le programmeur n'aura pas à se soucier de la gestion de la mémoire, et il n'aura en particulier jamais à effectuer de réservations de mémoire.

Toutefois, il reste indispensable de pouvoir mémoriser une valeur ou un résultat pour un usage ultérieur. Il est donc possible d'associer un *nom* (comme en C, constitué d'un ensemble de lettres, de chiffres et du symbole _, commençant impérativement par une lettre *minuscule*) à une valeur grâce à l'instruction **let**. En langage OCaml, on appelle ceci une *définition*.

Par exemple, pour associer le nom cherry à la valeur 7, on écrira :

```
# let cherry = 7;;
val cherry : int = 7
```

Dans l'interpréteur interactif, la troisième information que nous renvoie OCaml, la plus à gauche, correspond donc au *nom* auquel est associé le résultat. Si, comme précédemment, aucun nom n'est défini, on trouvera simplement un tiret - à gauche.

Si la valeur ne résidait pas déjà quelque part en mémoire, le langage réserve de lui-même la place pour la mémoriser, et y associe le nom choisi. Une fois le nom défini, il peut être utilisé dans des calculs.

```
# cherry * 5 + 7;;
- : int = 42
```

Dans la définition, on peut parfaitement utiliser une expression. Celle-ci est évaluée immédiatement, et c'est le résultat obtenu qui est associé au nom.

```
# let mango = cherry + 30;;
val mango : int = 37
```

Une définition dure jusqu'à la fin du programme, elle n'a pas de limite de portée. Il n'est pas non plus possible de *modifier* les valeurs désignées par les noms cherry, mango et pear. Il s'agit donc, techniquement, de noms désignant des constantes !

Il est en revanche permis, grâce à un nouveau **let**, d'écrire une *nouvelle* définition pour un nom déjà associé à une valeur. Cette nouvelle définition va avoir pour effet de « masquer » la définition précédente. Si la différence peut sembler anodine, nous verrons qu'elle est au cœur de la philosophie de la programmation fonctionnelle, et a des conséquences importantes.

Dans une définition, les noms apparaissant à droite du signe = sont remplacés par les objets qu'ils désignent (et l'expression est évaluée s'il y a lieu). Les redéfinitions d'un nom n'ont donc pas d'effet rétroactif :

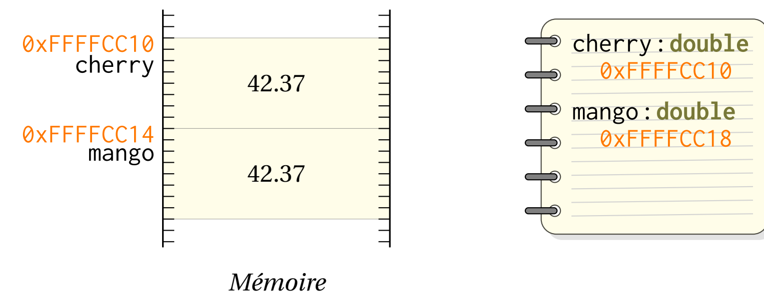
```
# let cherry = 0.7;;           (* On définit ici le nom cherry       *)
val cherry : float = 0.7

# let mango = cherry +. 3.0;; (* On définit à présent le nom mango *)
val mango : float = 3.7

# let cherry = 1.2;;           (* On redéfinit le nom cherry         *)
val cherry : float = 1.2

# mango;;                      (* Sans incidence sur le nom mango    *)
- : float = 3.7
```

## 3.2 Différences avec les variables en C

La gestion de la mémoire en OCaml est radicalement différente de celle du langage C. En particulier, si en C on écrit :

```
double cherry = 42.37;
double mango = cherry;
```

On réserve typiquement deux zones mémoires distinctes pour chacune des deux variables, et l'état de la mémoire sera quelque chose comme :



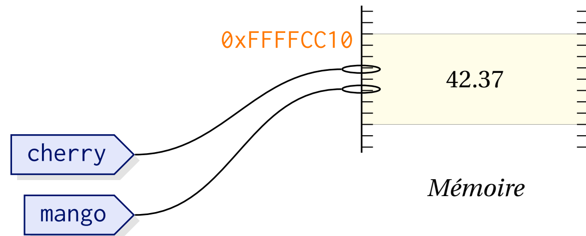

*Mémoire*

En revanche, la séquence, d'apparence similaire, de définitions OCaml

```
let cherry = 42.37;
let mango = cherry;
```

donne quelque chose de subtilement différent, la valeur 42.37 n'étant en particulier stockée qu'une seule fois en mémoire, les deux noms faisant référence à la même zone mémoire.

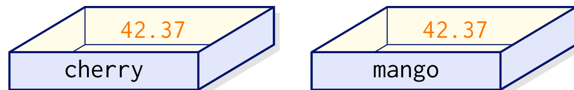
Le résultat en mémoire s'apparenterait à quelque chose de ce genre :



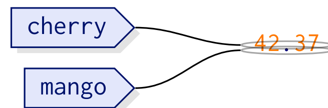

En particulier, la zone mémoire n'appartient pas aux noms `cherry` et `mango`. Les noms ne permettent que de retrouver la zone mémoire contenant la valeur qui leur est associée (et deux noms peuvent donc très bien faire référence à la même zone mémoire).

La représentation que l'on donne ici est, à dessein, moins précise que dans le cas du C : le programmeur n'a généralement pas besoin de savoir comment le langage s'occupera de gérer la mémoire. Il peut en revanche être utile d'en avoir une image mentale.

On peut ainsi visualiser les variables en C comme des « boîtes », créées lors de la déclaration, et dans laquelle on peut placer des valeurs (et éventuellement les changer). Un objet donné ne peut se trouver que dans une seule boîte.




Cette image ne convient pas aux langages comme OCaml [15], et une meilleure image serait de voir les noms comme des étiquettes accrochés aux objets (tels que des valeurs numériques ou booléennes) que l'on manipule. Plusieurs étiquettes peuvent donc désigner un même objet en mémoire. On parle d'*aliasing*.




Si l'on compare cette situation avec ce que l'on a vu avec le langage C, les noms en OCaml se comportent quelque peu comme des pointeurs [16], à ceci près qu'il n'est pas besoin de les déréférencer explicitement, et que toute la partie liée à la mémoire (réservation, obtention de l'adresse, etc.) est gérée « automatiquement » par le compilateur et cachée au programmeur [17].

Ce comportement n'est pas trivial pour le compilateur : lorsque l'on définit un nom, si l'objet associé n'est pas déjà présent en mémoire, il doit allouer de la mémoire pour l'y ranger. Mais lorsque l'on parvient à la limite de portée d'un nom local (ou lorsque l'on procède à une nouvelle définition globale d'un nom existant) se pose la question de la libération de la mémoire. Elle n'est pas systématique : d'autres noms (ou d'autres objets) peuvent encore faire référence à l'objet en mémoire !

Le langage dispose donc d'un outil particulier appelé *ramasse-miettes* (ou *garbage collector* en anglais), travaillant en arrière-plan, et capable d'identifier les objets en mémoire qui ne sont plus plus utiles, et le cas échéant de libérer la mémoire correspondante. Cela permet d'épargner à l'utilisateur la charge de la gestion explicite de la mémoire, mais c'est un mécanisme complexe, et qui peut avoir un impact sur les performances. Raisons pour lesquelles les premiers langages ne fonctionnaient pas de cette façon, mais un nombre croissant de langages [18] ont suivi cette route depuis, dont Python et OCaml (de même que la grande majorité des langages fonctionnels).

Précisons quand même que cette image d'« étiquettes » accrochées à des objets a des limites : s'il est possible de retrouver un objet à partir d'un nom, il n'existe en revanche aucun moyen de retrouver les noms associés à un objet donné, autrement dit aucune façon de « remonter le fil ».

## 3.3 Définitions multiples

Par commodité, il est possible d'effectuer plusieurs définitions d'un seul coup, grâce au mot-clé **and** [19] :

```
# let cherry = 37 and mango = 42;;
val cherry : int = 37
val mango : int = 42
```

Les définitions sont bien interprétées *simultanément* et non successivement, comme on peut le voir si l'on redéfinit `cherry` et `mango` en écrivant :

```
# let cherry = 37 and mango = 42;;
val cherry : int = 37
val mango : int = 42

# let cherry = 0 and mango = cherry;; (* Le valeur associée à cherry *)
val cherry : int = 0                 (* dans la seconde définition  *)
val mango : int = 37                 (* est celle précédant le let  *)
```

15. Qui, sur ce point, s'apparente largement à de nombreux langages comme Python, C#, etc.

16. C'est peu ou prou ce qui se passe réellement à l'insu du programmeur, même s'il y a des exceptions notables, à commencer par les entiers qui ne se comportent pas de la façon décrite ici. Toutefois, dans un programme écrit normalement, ces subtilités dans la gestion de la mémoire resteront invisibles du programmeur.

17. En particulier, une zone mémoire réservée sera automatiquement libérée par un mécanise appelé « *ramasse-miette* » (ou « *garbage collector* ») lorsque le programme ne peut plus y accéder (notamment lorsque plus aucun nom n'y fait référence). Idéalement, cela évite tout risque de fuite mémoire, mais la détermination des zones qui peuvent être libérées est une tâche potentiellement complexe et coûteuse.

18. Le premier langage ayant introduit ce concept est le langage LISP.

19. Raison pour laquelle **and** ne peut remplacer `&&`.

Dans cet exemple, la valeur associée à `cherry` lors de la définition de `mango` est le 37 de la définition précédente et non 0 !

Ce mécanisme de définitions multiples permet, entre autre choses, de redéfinir deux noms en échangeant les valeurs qui leur sont associées :

```
# let cherry = 37 and mango = 42;;
val cherry : int = 37
val mango : int = 42

# let cherry = mango and mango = cherry;;
val cherry : int = 42
val mango : int = 37
```

Encore une fois, soulignons que l'on n'a pas changé ce que désignent les noms `cherry` et `mango` créés par la première double définition, mais bien défini deux nouveaux noms `cherry` et `mango` qui, incidemment, masquent les définitions précédentes.

## 3.4 Définitions locales

On ne peut se satisfaire de ce seul mécanisme de définition, car avoir une portée des noms ainsi définis qui s'étend jusqu'à la fin du programme n'est pas toujours souhaitable. Un résultat de calcul intermédiaire n'a, par exemple, fréquemment pas vocation à être conservé plus longtemps que nécessaire.

Il est donc possible d'effectuer une définition dont la portée est limitée à l'expression qui la suit, grâce à last structure `let ... in ...` :

```
# let apple = 3 + 3
    in apple * 7;;
- : int = 42
```

Le retour à la ligne et l'indentation ne sont pas nécessaires (comme C, OCaml ignore l'indentation et les retours à la ligne), mais utilisés ici pour faciliter la lecture de l'expression.

On peut vérifier que la définition (et donc le nom `apple`) n'existent bien que le temps d'évaluer l'expression `apple * 7` :

```
# apple;;

Characters 2-7:
  apple;;
  ^^^^^
Error: Unbound value apple
```

On peut également faire des définitions locales multiples :

```
# let apple = 17 and banana = 25
    in apple + banana;;
- : int = 42
```

Il est possible d'utiliser une définition globale d'un nom qui a déjà été défini globalement. La définition globale n'est pas affectée :

```
# let cherry = 0;;
cherry : int = 0

# let cherry = 6 in cherry * 7;;
- : int = 42

# cherry;;
- : int = 0
```

Précisons que la construction « `let ... in ...` » est une expression, et peut donc s'utiliser n'importe où, comme dans les exemples suivants [20] :

```
# (let x = 2 in x*7) * (let y = 17 in y/5);;
- : int = 42

# cos (let pi = 3.1415926535897932 in pi /. 3.0);;
- : float = 0.500000000000000011
```

De même, il est possible d'imbriquer les définitions locales, par exemple :

```
# let cherry =
    let mango = 6
      in let pear = mango+1
           in mango * pear;;
cherry : int = 42
```

Dans ce dernier exemple, `mango` désigne 6 le temps du second `let ... in ...`, et `pear` désigne 7 juste le temps de la multiplication. Le résutat, 42, est associé au nom `cherry` via le premier `let`. On notera que *tous* les `let`, à l'exception possible du `let` initial, sont nécessairement associés à des `in`.

20. La seconde paire de parenthèses dans le premier exemple est inutile. Retirer la première paire de parenthèses n'aurait pas non plus d'effet ici sur le résultat, mais la sémantique serait légèrement différente : dans le cas présent, la portée du nom x s'arrête sur la première parenthèse fermante, elle irait jusqu'au double point-virgule en l'absence de la première paire de parenthèses.

# 4 Les fonctions

## 4.1 Fonctions avec un unique argument

Les fonctions sont des objets au cœur des langages fonctionnels tels que OCaml. Il n'est donc guère surprenant que le langage mette à notre disposition de nombreux moyens de les définir. Supposons tout d'abord que l'on souhaite créer une fonction $f$ définie par[21] :

$$f\begin{cases}\mathbb{R} \longmapsto \mathbb{R}\\ x \longmapsto 3x^2\end{cases}$$

Comme les constantes, les fonctions sont définies en associant un nom à leur expression au moyen d'un **let**. Une façon rapide de définir une telle fonction en OCaml est d'écrire

```
# let f x = 3. *. x ** 2.;;
val f : float -> float = <fun>
```

On a placé ici deux noms entre le **let** et le signe « = » : le premier est le nom de la fonction, le second est le nom qui désignera son argument dans l'expression, à droite du signe « = », constituant le corps de la fonction.

La signature obtenue indique que le nom « f » désigne à présent une fonction (`<fun>`) qui prend en argument un flottant et renvoie un flottant (« `float -> float` »). L'usage d'un opérateur spécifique pour les nombres flottants a permi à OCaml d'identifier correctement le type attendu pour l'argument de la fonction, ainsi que le type de la valeur renvoyée, sans qu'il eût été besoin de le faire explicitement, comme en C. La signature est donc un moyen de contrôler ce que l'on écrit.

Aucun nom x n'a ici été défini, comme on le voit dans la signature obtenue. Comme en langage C, ce nom, local à la fonction, n'existera que le temps de l'exécution de la fonction (plus qu'une variable locale, il s'agirait plutôt ici d'une « constante » locale, la fonction ne pouvant pas plus modifier la valeur associer à x qu'il n'est possible de modifier la valeur associée à un nom par une déclaration.

Une fois définie, la fonction s'utilise par exemple de la sorte :

```
# f 4.0;;
- : float = 48.0
```

ou bien encore en écrivant

```
# let z = 2.5 in f z;;
- : float = 18.75
```

21. La fonction ne sera évidemment pas réellement définie sur $\mathbb{R}$ mais simplement sur les flottants.

Il est également possible de définir *localement* des fonctions, par exemple :

```
# let g x = x * 3 in g 4;;
- : int = 12

# g;;

Characters 2-3:
  g;;
  ^
Error: Unbound value g
```

Comme nous l'avons signalé en introduisant les fonctions mathématiques usuelles, le langage OCaml ne requiert pas de parenthèses autour de l'argument lors d'un appel de fonction[22]. En mettre ne provoquerait pas une erreur, mais ce n'est pas l'usage car elles ne sont pas nécessaires.

Il y a une exception à cette règle : lorsque l'argument est négatif, il convient d'écrire

```
# f (-4.0);;
- : float = 48.0
```

En effet, ne pas mettre les parenthèses déclenche une erreur :

```
# f -4.0;;

Characters 2-3:
  f -4.0;;
  ^
Error: This expression has type float -> float
       but an expression was expected of type int
```

Dans ce dernier cas, OCaml pense que l'on a essayé de soustraire l'entier[23] `4.0` à l'entier `f`, et constaté que `f` était non pas un entier, mais une fonction prenant en argument un flottant et renvoyant un flottant, d'où le message d'erreur.

Compte tenu de l'ambiguité, il n'a pas pu reconnaître que le signe moins était l'opérateur unaire utilisé pour définir les nombres négatifs, et non l'opérateur binaire de soustraction. L'usage de parenthèses permet de résoudre cette difficulté.

22. Comme il n'en faut pas non plus autour des arguments dans la définition de la fonction, même si là aussi en ajouter inutilement ne conduirait généralement pas à une syntaxe incorrecte.

23. `4.0` n'est évidemment pas un entier, mais pour OCaml, les deux éléments à gauche et à droite de l'opérateur - devraient l'être. S'il n'y avait pas eu une erreur de type pour `f`, il y aurait eu une erreur de type sur `4.0`.

## 4.2 Le mot-clé « function »

Une autre manière de définir une fonction est d'utiliser le mot-clé function, qui utilise une syntaxe très proche des mathématiques :

```
# let f = function x -> 3.0 *. x ** 2.0;;
val f : float -> float = <fun>
```

À travers cette écriture, on associe le nom f à un objet OCaml défini comme « la fonction qui à $x$ associe $3x^2$ ».

La signature obtenue est exactement la même, et l'objet désigné par f rigoureusement identique, de sorte que son utilisation, entre autres choses, est la même :

```
# f 4.0;;
- : float = 48.0
```

En termes plus plus précis, function en OCaml[24] est un mot-clé qui permet de définir « *anonymement* » une fonction, indépendamment de tout nom. On pourrait donc écrire

```
# function x -> 3.0 *. x ** 2.0;;
- : float -> float = <fun>
```

mais aucun nom ne faisant référence à la fonction, on ne pourra l'utiliser, et elle sera perdue aussi vite qu'elle a été construire... On peut donc associer un identifiant à la fonction via une définition let. Mais on pourrait aussi très bien s'en servir directement :

```
# (function x -> 3 *. x ** 2.0) 4.0
- : float = 48.0
```

## 4.3 Arguments multiples

Le mot-clé function définit des fonctions qui, de façon similaire à ce qui se passe en mathématiques, prennent un *unique* argument. Il reste néanmoins possible de définir ce qui s'apparente à des fonctions attendant plusieurs variables. Par exemple, pour une fonction calculant la somme de deux entiers, on peut écrire

```
# let sum = function x -> function y -> x + y;;
val sum : int -> int -> int = <fun>
```

Il faut lire l'expression précédente « function x -> (function y -> x + y) » : la fonction sum est une fonction qui prend en argument un élément de $\mathbb{Z}$ et renvoie *une fonction* de $\mathbb{Z}$ à valeur dans $\mathbb{Z}$.

24. Comme lambda en Python.

Une telle construction correspond donc, mathématiquement, à :

$$\text{sum}\begin{cases}\mathbb{Z} \longmapsto (\mathbb{Z} \longmapsto \mathbb{Z}) \\ x \longmapsto \begin{cases}\mathbb{Z} \longmapsto \mathbb{Z} \\ y \longmapsto x+y\end{cases}\end{cases}$$

C'est l'interprétation qu'il faut donner à la signature fournie par OCaml. Elle est équivalente à int -> (int -> int), même si l'interpréteur n'indiquera pas, dans ce cas, les parenthèses, car la signature est lue de gauche à droite.

Si l'on fournit un int à la fonction sum, on obtient donc une fonction de signature int -> int. Ainsi,

```
# sum 2;;
- : int -> int = <fun>
```

On peut donner un nom à cette fonction, puis s'en servir :

```
# let add_two = sum 2;;
val add_two : int -> int = <fun>

# add_two 3;;
- : int = 5
```

Ici, add_two correspond donc à une fonction de $\mathbb{Z}$ dans $\mathbb{Z}$ qui, à tout $y \in \mathbb{Z}$ associe $2 + y$, autrement dit une fonction ajoutant 2 à son argument.

Heureusement, il n'est pas nécessaire d'aller si loin pour utiliser la fonction sum. Par exemple, on pourrait envisager de déterminer sum 2, puis d'appliquer le résultat à 3, en imposant cet ordre d'évaluation grâce à des parenthèses :

```
# (sum 2) 3;;
- : int = 5
```

Plus simplement encore, OCaml évaluant les expression de la gauche vers la droite (on parle *d'association à gauche*), l'expression sum 2 3 est équivalente à (sum 2) 3. Pour sommer deux entiers avec somme, il suffit donc, très simplement, d'écrire :

```
# sum 2 3;;
- : int = 5
```

Les deux arguments de sum succèdent donc au nom désignant la fonction, sans parenthèse ni virgule. Les fonctions OCaml de la bibliothèque standard nécessitant plusieurs arguments fonctionnent sur le même principe. Par exemple, la fonction mathématique

atan2 prenant en argument la partie imaginaire et la partie réelle d'un complexe [25] (ou l'ordonnée et l'abscisse d'un point du plan) et renvoyant son argument [26] permet par exemple de trouver l'argument de $3.14+1.41i$ en écrivant :

```
# atan2 1.41 3.14;;
- : float = 0.4220591190410658
```

Utiliser le mot-clé function (lequel ne permet de définir que des fonctions avec un unique argument) de la sorte étant un peu lourd, on dispose d'un autre mot-clé, **fun** qui fonctionne comme un raccourci :

```
# let sum = fun x y -> x + y;;
val sum : int -> int -> int = <fun>

# sum 2 3;;
- : int = 5
```

On remarque que la signature est exactement la même, et la fonction sum obtenue se comporte exactement de la même façon. On peut également définir sum d'une troisième et dernière façon, encore plus brève, mais toujours équivalente :

```
# let sum x y = x + y;;
val sum : int -> int -> int = <fun>

# sum 2 3;;
- : int = 5
```

Dans ces deux derniers cas, on peut fournir un seul argument et obtenir ainsi une fonction attendant un entier. Par exemple :

```
# let add_two = sum 2;;
val add_two : int -> int = <fun>

# add_two 40;;
- : int = 42
```

## 4.4 Signature de la fonction

Comme on a pu le voir sur les exemples précédents, OCaml détermine automatiquement le type des arguments de la fonction, ainsi que le type du résultat. Cela est rendu possible par le fait que les opérateurs nous renseignent sur la nature des opérandes. Il n'y a par exemple aucune ambiguité dans les définitions suivantes :

```
# let foo x y = int_of_float x + y;;
val foo : float -> int -> int = <fun>

# let bar f = f 1 +. 2.;;
val bar : (int -> float) -> float = <fun>
```

Dans ce second exemple, notamment, f est nécessairement une fonction prenant en argument un entier puisque le nom f est suivi dans l'expression définissant la fonction bar par un entier. Suit ensuite l'opérateur « +. », donc « f 1 » est de type float, ce qui donne le type du seul argument f de bar : « int -> float ».

Cela permet à OCaml de détecter très tôt d'éventuelles erreurs :

```
# let foo x y = int_of_float x +. y;;

Characters 17-31:
  let foo x y = int_of_float x +. y;;
                ^^^^^^^^^^^^^^
Error: This expression has type int but
       an expression was expected of type float
```

Il arrive parfois cependant qu'il ne soit pas possible de déterminer le type d'un argument, comme dans les exemples ci-dessous :

```
# let first x y = x;;
val first : 'a -> 'b -> 'a = <fun>

# let second x y = y;;
val second : 'a -> 'b -> 'b = <fun>
```

Ce sont des fonctions dites *polymorphes*. Le type 'a (ou 'b) indique que n'importe quel type est accepté. Cependant, dans la première fonction par exemple, le type du résultat sera, tout naturellement, le type du premier argument !

On peut ainsi utiliser les fonctions précédents avec des types différents :

```
# first 37 42;;
- : int = 37

# first 3.14 false;;
- : float = 3.14
```

25. Attention à l'ordre des arguments pour cette fonction !
26. Attention, on parle ici de l'argument d'un nombre complexe et non d'arguments de fonctions !

Ces fonctions peuvent être réutilisées dans d'autres fonctions, et le mécanisme de détermination des types tâchera toujours de déterminer le type d'un maximum d'arguments et de résultats :

```
# let sum x y = first x y + second x y;;
val sum : int -> int -> int = <fun>

# let sum x y = first x y + second y x;;
val sum : int -> 'a -> int = <fun>
```

Dans les deux définitions précédentes, la présence de l'opérateur + impose que les résultats des appels à `first` et `second` sont tous deux des entiers. Dans le premier cas, cela impose le type de `x` et `y`, mais dans le second cas, cela n'impose que le type de `x`, d'où les signatures différentes.

OCaml fournit quelques fonctions polymorphes, en particulier liées à la notion de comparaison. Les fonctions `max` et `min` notamment acceptent deux arguments de même type, et renvoient respectivement le plus grand et le plus petit des deux[27].

```
# max;;
- : 'a -> 'a -> 'a = <fun>

# max 42 37;;
- : int = 42
```

La fonction `compare`, elle, attend deux arguments `x` et `y` de même type et renvoie 0 si `x = y`, un entier positif si `x > y` et un entier négatif sinon[28].

```
# compare;;
- : 'a -> 'a -> int = <fun>

# compare 37 42;;
- : int = -1
```

## 4.5 Le type `unit`

Même si c'est moins courant dans un style fonctionnel, une fonction peut avoir un ou plusieurs « effets de bord » mais n'ait pas de résultat à renvoyer. Pour des raisons de cohérence, OCaml impose que toute fonction renvoie quelque chose, aussi dispose-t-on d'un type particulier désignant en fait, en quelque sorte, la notion de « rien », rôle que joue `void` dans le langage C[29]. C'est le type `unit`, réduit au seul élément « `()` », élément qui est notamment renvoyé par les fonctions effectuant des affichages. Il existe ainsi une fonction d'affichage pour la grande majorité des types courants[30] :

```
# print_int 42;;
42- : unit = ()

# print_float 3.14;;
3.14- : unit = ()

# print_string "Blop";;
Blop- : unit = ()
```

Dans l'interpréteur interactif, la lecture de la réponse n'est pas aisée, l'affichage demandé se confondant avec la valeur de renvoyée. On voit sur la même ligne l'affichage effectué par la fonction (par exemple « 42 ») suivi du résultat, de type `unit` (« `- : unit = ()` »). Pour un programme OCaml compilé, les types et valeurs renvoyés par les fonctions n'apparaissent pas lorsqu'on exécute normalement le programme. Seul les affichages produit par les fonctions `print_` seraient visibles.

OCaml n'effectue aucun retour à la ligne, afin de permettre plusieurs affichages sur la même ligne. La fonction `print_newline` permet d'obtenir ce retour à la ligne. En principe, elle ne devrait pas nécessiter d'argument, mais si l'on met uniquement le nom de la fonction, on obtient la fonction elle-même, on ne l'appelle pas !

```
# print_newline;;
- : unit -> unit = <fun>
```

On voit ici que la fonction `print_newline` prend un argument de type `unit`, aussi pour faire appel à celle-ci, on lui fournit en argument `()` :

```
# print_newline ();;

- : unit = ()
```

En apparence, la fonction semble être appelée de façon très similaire à d'autres langages, tels que C, en plaçant après le nom de la fonction des parenthèses n'encadrant rien[31] mais il s'agit bien ici d'une fonction prenant un argument[32], cet argument étant `()`.

27. En cas d'égalité, `min` renvoie le premier argument et `max` le second, ce détail pouvant avoir de l'importance.
28. La valeur exacte du résultat, en dehors de son signe, n'est pas spécifiée. La fonction `compare` gère également de façon particulière les flottants dits « `NaN` », mais cela ne nous occupe pas présentement.
29. Ou le type `NoneType` en Python, dont le seul représentant est `None`, et qui existe pour des raisons similaires.
30. La dernière fonction, `print_string`, permet d'afficher des chaînes de caractères, que l'on écrit encadrées par des guillemets. Nous aborderons les problématiques liées aux chaînes de caractères dans chacun des deux langages au programme un peu plus tard. Il n'existe pas de fonctions permettant d'afficher un booléen.
31. D'autant plus que l'espace entre le nom de la fonction et les parenthèses peut être omis... nous ne le garderons que pour rappeler la signification quelque peu différente de `()` en OCaml.
32. Il ne peut y en avoir zéro en OCaml.

## 4.6 Filtrage par motif

Le mot-clé `fonction` permet d'aller plus loin dans la définition de fonctions. Supposons par exemple que l'on souhaite créer une fonction sinc, définie de $\mathbb{R}$ dans $\mathbb{R}$ comme le prolongement par continuité en 0 de $x \mapsto \sin(x)/x$, c'est-à-dire :

$$\text{sinc} : \begin{cases} \mathbb{R} \longrightarrow \mathbb{R} \\ \begin{cases} 0 \longmapsto 1 \\ x \longmapsto \dfrac{\sin(x)}{x} \quad \text{si} \quad x \neq 0 \end{cases} \end{cases}$$

En l'absence du prolongement, nous pourrions définir la fonction de cette façon :

```
# let sinc = function x -> sin x /. x;;

val sinc : float -> float = <fun>
```

Mais cette définition laisse de côté le cas $x = 0$. OCaml nous propose une solution élégante de définir la fonction sinc, très similaire aux mathématiques :

```
# let sinc = function
    | 0.0 -> 1.0
    | x -> sin x /. x;;

val sinc : float -> float = <fun>
```

Ce type de structure est appelé *filtrage par motif*. Elle n'est possible qu'avec le mot-clé **`function`** en OCaml (**`fun`** ne le permet pas[33]). On associe ainsi un ensemble de *motifs* (à gauche des flèches) avec des expressions. Dans cette situation, **`function`** attendra un argument qu'il essaiera d'identifier avec, successivement, chacun des motifs dans leur ordre d'apparition, puis utilisera l'expression associée au *premier* motif qui convient.

Si le motif est une valeur « *immédiate* » (une constante), l'argument de **`function`** doit être égal à cette valeur. S'il s'agit d'un nom, il peut être identifié à n'importe quel valeur, ce nom désignant alors la valeur en question dans l'expression associée. Ainsi, `sinc 0.0` utilisera la première des expressions, tandis que `sinc 1.0` utilisera la seconde, l'identification de `1.0` avec le premier motif ayant échoué. Le nom `x` pouvant représenter n'importe quelle valeur, l'identification avec le second motif est un succès.

Dans le cas où l'argument correspond à plusieurs cas possibles, OCaml s'arrête sur le premier qui convient. Il est donc important de faire figurer le motif « `0.0 -> ...` » avant celui « `x -> ...` » qui accepterait également la valeur `0.0` !

33. C'était toutefois possible dans une ancienne version du langage destinée à l'enseignement, appelée Caml Light, et utilisée en classe préparatoire jusqu'en 2018, aussi prenez garde à ce qui pourrait apparaître dans des sujets ou des ouvrages un peu anciens.

Un point très important à saisir est qu'un nom apparaissant dans un motif, même s'il a été défini à l'extérieur, n'est **jamais** remplacé par sa valeur, comme on le voit ci-dessous :

```
# let zero = 0.0;;
val zero : float = 0.

# let sinc = function
    | zero -> 1.0;
    | x -> sin x /. x;;

Characters 50-51:
    | x -> sin x /. x;;
      ^
Warning 11: this match case is unused.
val sinc : 'a -> float = <fun>
```

Sur ce dernier exemple, la fonction `sinc` renverra *toujours* `1.0`, quel que soit l'argument[34], car tout argument peut être identifié au nom `zero` !

Si l'on n'a pas besoin de l'argument dans l'expression, on utilise généralement le nom « `_` » comme motif. Comme tout nom, il peut être identifié avec n'importe quoi, la valeur étant « perdue » lorsque l'on évalue l'expression :

```
# let is_zero = function
    | 0 -> true
    | _ -> false;;

val is_zero : int -> bool = <fun>
```

Contrairement à ce que l'on pourrait penser, on ne peut pas se servir des motifs pour construire simplement une fonction polymorphe non triviale :

```
# let is_zero = function
    | 0   -> true
    | 0.0 -> true
    | _   -> false;;

Characters 49-52:
      | 0.0 -> true
        ^^^
Error: This pattern matches values of type float
       but a pattern was expected which matches values of type int
```

34. Et quel que soit son type, comme en témoigne la signature.

Dès qu'un motif permet de déterminer le type de l'argument (que ce soit explicitement, à gauche de la flèche, ou en étudiant l'utilisation du motif à droite de cette même flèche), celui-ci ne peut plus être changé.

Ainsi, dans l'exemple précédent, la première ligne du filtrage a permi de déterminer que l'argument était un entier. La seconde ligne du filtrage, qui fait référence à un argument flottant, ne saurait donc convenir.

Signalons enfin la possibilité d'utiliser la même expression pour plusieurs motifs [35], en écrivant par exemple :

```
# let is_digit = function
    | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 -> true;
    | x -> false;;

val is_digit : int -> bool = <fun>
```

Profitons de l'occasion pour rappeler une fois encore que l'indentation n'a aucune signification particulière pour OCaml, ce sont les symboles « `|` » qui permettent de séparer les différents motifs [36]. Il n'est absolument pas besoin d'aligner les `|` des motifs (ou les `->`), hormis pour des raisons de lisibilité du code.

## 4.7 Motifs gardés

Ne pouvoir utiliser comme motif que des noms (qui s'identifient à tout objet) et des constantes serait vite limitant [37]. Il est également possible de définir des conditions supplémentaire dans un motif, sous la forme d'une expression booléenne (on ne pourra identifier le motif à l'argument de `function` que si celle-ci est évaluée à `true`) grâce au mot-clé `when`. On parle alors de *motif gardé* :

```
# let is_nonnegative = function
    | x when x >= 0 -> true
    | _             -> false;;

is_nonnegative : int -> bool = <fun>
```

Le filtrage par motif utilise le même genre de stratégie « paresseuse » que les expressions booléennes : les motifs étant considérés un par un, dans l'ordre, les conditions des motifs gardés ne sont évaluées que si l'on n'a pas réussi à identifier un motif qui convient parmi ceux qui précède, et si le motif associé à la garde peut correspondre à l'argument de `function` (après un motif « `x when x<=0` », on peut sans aucun risque mettre un motif « `x when sqrt x < 2.0` »).

Contrairement à ce qu'il se passait pour les motifs, on peut utiliser des noms pour ce qu'ils désignent dans la garde :

```
# let zero = 0;;
val zero : int = 0

# let is_nonnegative = function
    | x when x >= zero -> true
    | _                -> false;;

is_nonnegative : int -> bool = <fun>
```

Dans une fonction, on cherchera en principe à donner une expression pour toutes les valeurs possibles de l'argument [38]. Toute valeur passée à la fonction doit donc pouvoir être associée à au moins un des motifs. On dira alors que le filtrage est *exhaustif*.

Parfois, OCaml sera capable de détecter qu'un motif n'est pas exhaustif. Supposons par exemple que l'on cherche à écrire une fonction renvoyant un entier représentant le signe de son argument (entier), soit 1 si l'argument est strictement positif, 0 s'il est nul, $-1$ s'il est strictement négatif, et que l'on oublie de préciser un des cas :

```
# let sign = function
    | 0          -> 0
    | x when x<0 -> -1;;

Characters 14-58:
  ............function
      | 0 -> 0
      | x when x<0 -> -1..

Warning 8: this pattern-matching is not exhaustive.
Here is an example of a value that is not matched:
1

(However, some guarded clause may match this value.)

val sign : int -> int = <fun>
```

35. Bien entendu, si un nom apparaît dans un motif, il doit apparaître dans *tous* les motifs associés à la même expression.
36. Et, incidemment, la raison pour laquelle « `|` » ne pouvait servir de « ou » logique sur les booléens.
37. Même si nous verrons dans le chapitre suivant qu'il est possible d'écrire des motifs plus élaborés.

38. Pour les fonctions dont le domaine de définition ne s'étend pas à tous les éléments du type utilisé pour l'argument de la fonction, on préférera généralement proposer un motif pour les éléments qui ne figurent pas dans le domaine de définition, mais déclenchant une erreur spécifique. Nous verrons que l'on peut écrire par exemple « `-> failwith "Argument incorrect"` ».

Il a ici raison, le cas de 1 n'est pas considéré. Il ne s'agit que d'un avertissement, la fonction est quand même définie et peut être utilisée :

```
# sign (-42);;
- : int = -1
```

Mais elle déclenchera une erreur (ou plutôt lèvera une *exception*) si l'on utilise comme argument une valeur qui ne figure pas parmi les motifs possibles :

```
# sign 1;;
Exception: Match_failure ("//toplevel//", 225, 12).
```

OCaml n'est cependant pas toujours capable de se rendre compte qu'un ensemble de motifs gardés constitue un filtrage exhaustif :

```
# let sign = function
    | 0            -> 0
    | x when x > 0 -> 1
    | x when x < 0 -> -1;;

Characters 14-92:
  ...........function
      | 0            -> 0
      | x when x > 0 -> 1
      | x when x < 0 -> -1..

Warning 8: this pattern-matching is not exhaustive.
Here is an example of a value that is not matched:
1

(However, some guarded clause may match this value.)

val sign : int -> int = <fun>
```

Pour cette raison, on préférera l'écriture suivante, équivalente (car le dernier motif ne sera considéré que si les précédents ne conviennent pas) :

```
# let sign = function
    | 0            -> 0
    | x when x > 0 -> 1
    | _            -> -1;;

sign : int -> int = <fun>
```

## 4.8 Fonctions récursives

Comme en C, il est possible en OCaml d'écrire des fonctions récursives. Mais pour ce faire, il faut utiliser un opérateur spécifique lors de la définiion, car si on utilise `let` pour définir une fonction `foo`, dans le corps de cette fonction `foo`, le nom `foo` n'existe en principe pas encore (ou, pire, il fait référence à un *autre* objet, suivant une déclaration antérieure de `foo`). OCaml fournit un outil spécifique pour les définitions récursives au travers du mot-clé « `let rec` ».

Par exemple, il est très simple de définir récursivement la fonction `fact` représentant la factorielle en mathématiques :

```
# let rec fact = function
    | 0 -> 1
    | n -> n * fact (n-1);;

val fact : int -> int = <fun>
```

Nous verrons un peu plus tard que cette définition, très proche de la définition mathématique, n'est pas la plus efficace, mais nous nous en contenterons bien pour l'instant. Cette fonction n'a de sens que pour des entiers positifs ou nuls, et « pas trop grands[39].

Pour définir une fonction prenant en argument un entier et renvoyant un booléen indiquant sa parité, on écrira généralement quelque chose tel que :

```
# let is_even =
    function n -> n mod 2 = 0;;

val is_even : int -> bool = <fun>
```

Supposons un instant que, dans un but d'illustration du mécanisme de la récursion, on abandonne un temps toute raison. On *pourrait* écrire une fonction récursive déterminant, de manière souvent *très inefficace*, la parité d'un entier[40] de la sorte :

```
# let rec is_even = function
    | 0 -> true
    | 1 -> false
    | n -> is_even (n-2);;

val is_even : int -> bool = <fun>
```

39. On observe un débordement au-delà de « `fact 12` » pour des entiers sur 31 bits et de « `fact 20` » pour des entiers sur 63 bits.

40. En principe positif, mais dans le cas d'un nombre négatif, après un débordement et plusieurs miliards de milliards d'appels récursifs, on pourrait en théorie obtenir le résultat correct.

On peut même réduire le nombre de motifs en utilisant la négation booléenne [41] :

```
# let rec is_even = function
    | 0 -> true
    | n -> not (is_even (n-1));;

val is_even : int -> bool = <fun>
```

Pour mieux tenir compte des nombres négatifs, on pourrait ajouter, dans le filtrage, un motif supplémentaire entre les deux motifs précédents [42] :

```
# let rec is_even = function
    | 0          -> true
    | n when n<0 -> is_even (-n)
    | n          -> not (is_even (n-1));;

val est_pair : int -> bool = <fun>
```

Une autre façon un peu exotique [43] de déterminer la parité d'un entier serait d'utiliser deux fonctions *mutuellement récursives*, qu'ils nous faut définir simultanément grâce au mot-clé `and` :

```
# let rec is_even = function
    | 0          -> true
    | n when n<0 -> is_even (-n)
    | n          -> is_odd (n-1)
  and is_odd = function
    | 0          -> false
    | n when n<0 -> is_odd (-n)
    | n          -> is_even (n-1);;

val is_even : int -> bool = <fun>
val is_odd : int -> bool = <fun>
```

On le voit, le langage nous offre de nombreuses possibilités pour définir des fonctions récursives, que nous aurons l'occasion de mettre en œuvre dans des cas plus réalistes que celui ci-dessus.

41. Nous verrons, lorsque nous étudierons plus en détail le fonctionnement de la récursion en informatique, que cette fonction, tout aussi inefficace que la précédente, a en plus l'inconvénient de ne *pas* fonctionner pour de grandes valeurs de `n` en raison d'une consommation mémoire trop importante.

42. Il doit nécessairement se trouver après le premier motif sinon on risque une séquence infinie d'appels pour l'argument 0, et avant le second motif qui accepte tout argument.

43. Uniquement pour illustrer simplement les possibilités du langage !

# 5 Expressions avancées

## 5.1 Expressions conditionnelles

Revenons un temps sur l'exemple de la suite de Syracuse introduit au début du second chapitre, et supposons que l'on souhaite une fonction `next` prenant en argument un entier positif $u_n$ et renvoyant l'entier $u_{n+1}$ le suivant immédiatement dans cette suite. Pour déterminer $u_{n+1}$, il faut examiner la parité de $u_n$. Cela peut se faire avec un filtrage :

```
# let next = function
    | u when u mod 2 = 0 -> u/2
    | u                  -> 3*u+1;;

val next : int -> int = <fun>
```

Si le filtrage par motif est un outil puissant, on peut parfois avoir besoin de quelque chose de plus simple, par exemple lorsque, comme ici, on souhaite simplement utiliser une expression ou une autre en fonction d'un booléen. Pour ce faire, on dispose en OCaml de la structure « `if ... then ... else ...` ».

Naturellement, une expression dont l'évaluation conduit à un booléen est attendue entre le `if` et le `then`, et deux expressions, correspondant respectivement aux cas où ce résultat booléen est `true` ou `false`, après les mots-clés `then` et `else`.

On peut ainsi écrire notre fonction `next` ainsi :

```
# let next u =
    if (u mod 2 = 0) then u/2 else 3*u+1;;

val next : int -> int = <fun>
```

Malgré la ressemblance avec la structure de même nom en C, il ne s'agit pas ici à proprement parler d'une structure de contrôle, mais bien d'une expression, certes un peu particulière. En particulier, elle peut apparaître à tout endroit où une expression aurait sa place :

```
# let mango = 17;;
val mango : int = 17

# let cherry = 3 + (if mango<42 then mango else 42) * 2;
val cherry : int = 37
```

L'expression précédente, puisque `mango` désigne un entier plus petit que 42, est donc comprise comme « `3 + 17 * 2` ».

Comme la structure « `if ... then ... else ...` » constitue une *expression* en OCaml, elle a nécessairement un type bien défini, de sorte que l'expression qui suit le `then` et celle suivant le `else` doivent **nécessairement** être de même type.

Il serait donc illégal, en OCaml, d'écrire quelque chose comme :

```
# if mango < 42 then 5 else 3.7;;

Characters 26-29:
  if mango < 42 then 5 else 3.7;;
                            ^^^
Error: This expression has type float
       but an expression was expected of type int
```

De même, **le `else` ne peut jamais être absent** !

```
# if mango < 42 then 5;;

Characters 19-20:
  if mango < 42 then 5;;
                     ^
Error: This expression has type int
       but an expression was expected of type unit
```

Il y a une seule exception à cette règle : lorsque l'expression suivant le `then` est évaluée en un `unit`... par commodité, dans cette situation, un « `else ()` » est implicitement ajouté. Ce qui explique le message d'erreur précédent, faisant mention de `unit`.

Compte tenu du caractère éminemment pratique de ce genre d'écriture, un équivalent existe dans la plupart des langages, dont le C. On parle souvent d'*expression ternaire*. En langage C, cela s'écrit [44] « `cond ? expr1 : expr2` », où `cond` est une expression booléenne, et `expr1` et `expr2` les deux expressions à considérer selon le résultat de l'évaluation de l'expression booléenne. La fonction `next` pourrait donc être écrite en C de la façon suivante :

```c
int next(int u) {
    return u%2 == 0 ? u/2 : 3*u+1;
}
```

Il est aisé de voir que ce que l'on gagne en concision, on le perd rapidement en lisibilité. Il est sans doute plus raisonnable de limiter l'utilisation de l'opérateur ternaire aux seuls rares cas où il améliore la lisibilité de la fonction !

44. Les espaces, comme souvent en C, sont facultatives.

## 5.2 Filtrage

La construction « `if ... then ... else ...` » présente l'avantage d'être utilisable partout où on attend une expression. Mais il ne s'agit que d'un choix entre deux alternatives, cela n'a pas l'expressivité de `function`.

Le langage met donc à notre disposition la construction « `match ... with ...` » pour plus de souplesse. Entre les mots-clés `match` et `with` doit se trouver une expression (de type quelconque), laquelle sera évaluée, puis filtrée par les motifs suivant le `with`.

Pour la fonction `next`, on pourrait par exemple écrire :

```
# let next u =
    match u mod 2 with
      | 0 -> u/2
      | _ -> 3*u+1;;

val next : int -> int = <fun>
```

Sur cet exemple, la fonction `next` prend un argument nommé `u`, évalue le reste de la division entière de `u` par 2, puis selon que le reste vaille 0 ou 1 [45], le résultat de l'ensemble de l'expression « `match ... with ...` » vaudra `u/2` ou `3*u+1`.

La structure `match ... with ...` est donc une généralisation de ce que permet `function`. En particulier, une fonction définie par

```
# let foo = function
    | ...
```

pourrait très bien être définie, de manière tout à fait équivalente, avec une structure `match ... with` de la sorte :

```
# let foo x = match x with
    | ...
```

On remarquera la présence explicite d'un argument dans la seconde écriture, qui n'apparaît pas dans la première (le mot-clé `function` indiquant qu'un argument est attendu, et sera ensuite filtré).

Une telle structure est particulièrement puissante et versatile [46], mais nécessite du temps pour être bien maîtrisée. Elle est présentée ici à titre introductif, nous nous y habituerons progressivement au travers d'exemples concrets.

45. Noté ici _ car OCaml n'est pas capable de voir que l'évaluation de l'expression ne peut conduire qu'à 0 ou 1 et s'interrogerait quant à l'exhaustivité du filtrage si l'on avait mis explicitement 1.

46. Et est suffisamment intéressante pour qu'une variante apparaisse progressivement dans la plupart des langages, par exemple dans les versions *très* récentes des langages Python et C++.

# 6 Couples

## 6.1 Principe

Il est parfois utile de pouvoir « regrouper » plusieurs objets que l'on manipule. Ceci est possible à travers des « n-uplets » où les différents objets sont séparés par des virgules, l'ensemble étant fréquemment entouré de parenthèses [47]. Il est possible de grouper un nombre quelconque d'éléments, qu'ils soient ou non de même type. L'exemple suivant regroupe par exemple un flottant, un booléen, et une fonction :

```
# (3.14, false, function x -> x**2.);;
- : float * bool * (float -> float) = (3.14, false, <fun>)
```

Le type de l'objet ainsi construit correspond au produit cartésien des types des différents éléments. Dans la signature, le produit cartésien est noté `*`. On peut utiliser une définition pour associer un nom à un tel couple, telle que

```
# let grp = (3.14, false, function x -> x**2.);;
val grp : float * bool * (float -> float) = (3.14, false, <fun>)
```

De tels groupes peuvent être utilisés comme arguments d'une fonction. On peut par exemple écrire une fonction effectuant la somme de deux entiers ainsi :

```
# let sum = function (x, y) -> x + y;;
val sum : int * int -> int = <fun>
```

Les fonctions prenant en argument un n-uplet n'ont pas nécessairement besoin d'un filtrage explicite, on peut simplement écrire, de façon tout à fait équivalente [48] :

```
# let sum (x, y) = x + y;;
val sum : int * int -> int = <fun>
```

Si l'on regarde attentivement les signatures des fonctions précédentes, on a ici défini des fonctions de $\mathbb{Z} \times \mathbb{Z}$ à valeurs dans $\mathbb{Z}$. Elles ne prennent qu'un seul et unique argument, et le filtrent avec `function`. Pour faire appel à de telles fonctions, on écrira :

```
# sum (2, 3);;
- : int = 5
```

47. Toutefois, comme c'est également le cas en Python, les parenthèses ne sont en fait pas requises s'il n'y a pas d'ambiguïté.

48. On remarquera la grande similarité de cette écriture avec la façon dont les fonctions sont définies dans d'autres langages.

La présence de parenthèses, et de virgules séparant chaque élément, rend la syntaxe très proche de ce qui est utilisé dans d'autres langages. Toutefois, on a affaire à un objet un peu différent des fonctions présentées dans le premier chapitre, que l'on avait définies avec l'une des variantes (toutes trois équivalentes) suivantes :

```
# let sum_cur x y = x + y;;
val sum_cur : int -> int -> int = <fun>

# let sum_cur = fun x y -> x + y;;
val sum_cur : int -> int -> int = <fun>

# let sum_cur = function x -> function y -> x + y;;
val sum_cur : int -> int -> int = <fun>
```

Les trois formes ci-dessus sont dite « *currifiées* [49] ». On remarquera en particulier la différence de signature entre les deux approches, que l'on peut résumer ainsi (formes currifiées à gauche) :

$$\texttt{somme_cur}\begin{cases}\mathbb{Z} \longmapsto (\mathbb{Z} \longmapsto \mathbb{Z}) \\ x \longmapsto \begin{cases}\mathbb{Z} \longmapsto \mathbb{Z} \\ y \longmapsto x+y\end{cases}\end{cases} \qquad \texttt{somme}\begin{cases}\mathbb{Z} \times \mathbb{Z} \longmapsto \mathbb{Z} \\ (x, y) \longmapsto x+y\end{cases}$$

De fait, la *currification* est l'opération qui consiste à transformer une fonction à plusieurs variables (par exemple ici définie sur $\mathbb{Z} \times \mathbb{Z}$ à valeur dans $\mathbb{Z}$) en une fonction à une unique variable renvoyant une fonction sur le reste des arguments (par exemple ici une fonction définie sur $\mathbb{Z}$ à valeur dans l'espace des fonctions $\mathbb{Z} \mapsto \mathbb{Z}$).

Les formes currifiées, plus souples, sont beaucoup plus souvent utilisées en OCaml (et de façon générale dans les langages fonctionnels). Elles permettent notamment plus aisément de créer des *applications partielles*, c'est-à-dire des fonctions annexes où le premier des paramètres est fixé, à partir de la forme currifiée d'une fonction :

```
# let add_two = sum_cur 2;;
val add_two : int -> int = <fun>
```

Il reste néanmoins possible de créer une application partielle avec une fonction non currifiée, mais c'est un peu plus « lourd », et possiblement légèrement moins efficace :

```
# let add_two = function x -> sum (2, x);;
val add_two : int -> int = <fun>
```

49. Du nom du mathématicien et logicien Haskell Brook Curry, qui a popularisé cette notation, même s'il semblerait qu'elle ait été initialement proposée par Moses Shönfinkel. Trois languages de programmation ont été nommés en son honneur, Haskell, Brook et Curry, le premier des trois restant une référence dans le domaine de la programmation fonctionnelle.

Pour des « paires » d'éléments (2-uplets), il existe deux fonctions fournies par le langage, `fst` et `snd`, qui permettent d'obtenir chacun des deux éléments du 2-uplet :

```
# fst;;
- : 'a * 'b -> 'a = <fun>

# snd;;
- : 'a * 'b -> 'b = <fun>

# let pair = (42, 37);;
pair : int * int = 42, 37

# snd pair;;
- : int = 37
```

Il n'existe en revanche pas de manière simple d'accéder aux éléments d'un n-uplet dès lors qu'il contient trois éléments ou plus. Pour extraire les différents éléments d'un n-uplet à trois éléments, on peut cependant créer des fonctions spécifiques :

```
# let first_among_three (a, b, c) = a;;
val first_among_three : 'a * 'b * 'c -> 'a = <fun>

# let second_among_three (a, b, c) = b;;
val second_among_three : 'a * 'b * 'c -> 'b = <fun>

# let third_among_three (a, b, c) = c;;
val third_among_three : 'a * 'b * 'c -> 'c = <fun>
```

On peut alors les utiliser pour obtenir les différents éléments :

```
# first_among_three grp;;
- : float = 3.14

# third_among_three grp (first_among_three grp);;
- : float = 9.8596
```

## 6.2 Couples et filtrages

Si l'on préfèrera en général les formes curryfiées, il est parfois pratique d'utiliser un couple (ou un tuple) comme argument si on souhaite effectuer un filtrage. En effet, seul le mot-clé `function` permet d'effectuer un filtrage par motif, `fun` ne le permet pas, et `function` ne peut prendre qu'un seul et unique argument.

Par exemple, pour calculer le PGCD de deux nombres par l'algorithme d'Euclide, on ne peut pas écrire :

```
# let rec gcd = fun
    | a 0 -> a
    | a b -> gcd b (a mod b);;

Characters 23-24:
    | a 0 -> a
    ^
Error: Syntax error
```

On peut en revanche écrire [50] :

```
# let rec gcd = function
    | (a, 0) -> a
    | (a, b) -> gcd (b, a mod b);;

val gcd : int * int -> int = <fun>
```

Pour obtenir une fonction currifiée, on peut utiliser une fonction auxilliaire :

```
# let gcd a b =
    let rec gcd_aux = function
        | (a, 0) -> a
        | (a, b) -> gcd_aux (b, a mod b)
    in gcd_aux (a, b);;

val gcd : int -> int -> int = <fun>
```

Remarquons qu'ici, on peut en fait se passer de cette pirouette [51], car il suffit de filtrer le second paramètre de la fonction :

```
# let rec gcd a = function
    | 0 -> a
    | b -> gcd b (a mod b);;

val gcd : int -> int -> int = <fun>
```

50. Comme en Python, les parenthèses autour des n-uplets sont facultatives s'il n'y a pas d'ambiguïté. On peut donc s'en dispenser dans les motifs de filtrage ici, mais *pas* dans l'appel récursif, car l'appel de fonction étant prioritaire sur « , », « `gcd b, a mod b` » correspondrait à « `(gcd b), (a mod b)` ».

51. Il est en fait bien évidemment toujours possible de filtrer argument par argument, mais l'écriture peut devenir assez lourde.

# Exercices

## Ex. 5.1 – Définitions

Déterminer les réponses de Caml aux définitions suivantes :

```
let a = 1;;

let f n = 3 * n - 1;;

let a = 2 in f a;;

let a = f a in f a;;

let a = f a and b = f a in f b;;

let x =
  let x = 5 and y = 2 in
    let x = x+y and y = x-y in
      2 * x / y;;
```

## Ex. 5.2 – Composition

On définit deux fonctions f et g de la façon suivante :

```
let f n = n + 2 and g n = 3 * n;;
```

On souhaite définir une fonction h comme la composition de ces deux fonctions, soit $h = g \circ f$, et calculer $h(5)$. Parmi les définitions suivantes, lesquelles sont correctes ?

```
let h   = g f       in h 5;;

let h n = g f n     in h 5;;

let h n = (g f) n   in h 5;;

let h n = g (f n)   in h 5;;

let h n = g(f) (n)  in h 5;;

let h n = g (f (n)) in h 5;;
```

## Ex. 5.3 – Typage

Proposer des expressions Caml qui ont pu donner les signatures suivantes :

```
val f1 : int -> int -> int = <fun>

val f2 : (int -> int) -> int = <fun>

val f3 : int -> (int -> int) -> int = <fun>
```

Déterminer le type des expressions suivantes :

```
let f4 = fun f x y -> f x y;;

let f5 = fun f g x -> g (f x);;

let f6 = fun f g x -> (f x) + (g x);;
```

Même chose avec ces expressions, où les noms sont moins révélateurs :

```
let f7 = fun x y z -> x y z;;

let f8 = fun x y -> y x x x;;

let f9 = fun x y z -> x (y z x);;

let f10 = fun x y z -> (x y) (z x);;
```

## Ex. 5.4 – Médiane

Proposer une fonction `med_of_3` prenant trois arguments $a$, $b$, $c$ de même type (et pour lesquels `<=` est une relation d'ordre) et renvoyant leur médiane.

## Ex. 5.5 – Exponentiation et logarithmes entiers

**1.** Proposer une fonction `pow` de signature `int -> int -> int` telle que « `pow p n` » renvoie $p^n$. On supposera $n \geqslant 0$.

**2.** Proposer de même une fonction `ilog` de signature `int -> int -> int` telle que « `ilog p n` » renvoie $\lfloor \log_p n \rfloor$, soit le plus grand entier $k$ tel que $p^k \leqslant n$. On supposera $p \geqslant 2$.

## Ex. 5.6 – Fonction récursive

Que fait la fonction suivante ? On supposera $a$ et $b$ positifs.

```
let rec foo a b =
  if a < b then foo a (b-a) else
  if a > b then foo b (a-b) else
  a
```

## Ex. 5.7 – Ordre et théorème de Charkovski

L'ordre de Charkovski est un ordre total sur $\mathbb{N}^*$ proposé par le mathématicien ukrainien Oleksandr Charkovski [52], défini par

$$3 \preceq 5 \preceq 7 \preceq 9 \preceq \ldots \preceq 2\cdot 3 \preceq 2\cdot 5 \preceq 2\cdot 7 \preceq \ldots$$

$$\ldots \preceq 2^n\cdot 3 \preceq 2^n\cdot 5 \preceq 2^n\cdot 7 \preceq \ldots \preceq 2^{n+1}\cdot 3 \preceq 2^{n+1}\cdot 5 \preceq 2^{n+1}\cdot 7 \preceq \ldots$$

$$\ldots \preceq 2^n \preceq 2^{n-1} \preceq \ldots \preceq 8 \preceq 4 \preceq 2 \preceq 1$$

Proposer une fonction `greater` de signature `int -> int -> bool` prenant en argument deux entiers strictement positifs `a` et `b` et renvoyant un booléen indiquant si `a`$\preceq$`b`.

Cet ordre est au cœur du théorème de Charkovski (1964) qui affirme que, si une fonction $f$ continue de I à valeurs dans I (où I est un intervalle non-trivial de $\mathbb{R}$) admet un point périodique de période $n$ (il existe $x \in I$ tel que $f^n(x) = x$), alors pour tout $m \succeq n$, $f$ admet un point périodique de période $m$. En particulier, s'il y a un point périodique de période 3, on peut trouver un point périodique pour une période quelconque !

## Ex. 5.8 – Fonction récursive

On propose la fonction suivante, définie pour $n \in \mathbb{N}^*$ :

```
let rec foo = function
  | 1 -> 1
  | n -> let rec bar = function
           | 0 -> 0
           | k -> foo k + bar (k-1)
         in bar (n-1)
```

**1.** Déterminer la signature de la fonction `foo` et décrire son fonctionnement (on pourra en particulier exprimer `foo n` en fonction d'un ou plusieurs `foo n'` avec `n'<n`).

**2.** En déduire la valeur de `foo k`.

**3.** Quelle est la complexité de la fonction `foo` en fonction de `n` ?

## Ex. 5.9 – Fonction tak

La fonction $\tau$, proposée par John McCarthy et baptisée « tak » (en hommage à Ikuo Takeuchi qui a proposé en 1978 une fonction « tarai » similaire), est définie par :

$$\tau(x,y,z) = \begin{cases} z \text{ si } x \leqslant y \\ \tau(\tau(x-1,y,z),\tau(y-1,z,x),\tau(z-1,x,y)) \text{ sinon} \end{cases}$$

Proposer une implémentation OCaml de cette fonction [53].

## Ex. 5.10 – Le problème du dépouillement

On considère une élection où deux votes sont possibles, A et B. On suppose qu'il y a eu $a$ votes pour A et $b$ votes pour B, avec $a \geqslant b$. Lors du dépouillement, les bulletins sont lus un par un, et le décompte des voix est effectué.

Proposer une fonction [54] `count` de signature `int -> int -> int` prenant en argument $a$ et $b$ et renvoyant le nombre d'ordres possibles pour l'ouverture des bulletins de sorte que B ne soit à aucun moment en tête des votes (strictement).

On pourra par exemple vérifier que si `a`= 5 et `b`= 4, il existe 42 tels ordres de dépouillement possibles.

## Ex. 5.11 – Diviseurs

**1.** Proposer une fonction `divisible` de signature `int -> int -> bool` prenant en argument un entier `d` strictement positif et un entier `n` et renvoyant un booléen indiquant si `n` est divisible par `d`

**2.** En *déduire* une fonction `is_even` de signature `int -> bool`.

**3.** Proposer une fonction récursive simple `is_prime`, de signature `int -> bool`, utilisant `divisible` et testant si son argument `n`> 1 a un diviseur propre entre 2 et `n`−1.

52. Il existe de nombreuses translittérations pour ce patronyme, originellement écrit dans la variante ukrainienne de l'alphabet cyrillique.

53. On pourra se contenter d'une implémentation naïve de cette fonction. Celle-ci a été en fait créée pour tester les capacités d'un langage en terme de récursion, et son évaluation peut être très coûteuse même pour de petites valeurs de `x`, `y` et `z`. Nous étudierons plus tard des techniques permettant d'obtenir le résultat plus efficacement.

54. Comme dans l'exercice précédent, on pourra se contenter d'une implémentation naïve.

### Ex. 5.12 – Différences finies

On suppose qu'une suite $(u_n)_{n\in\mathbb{N}}$ d'éléments de $\mathbb{R}$ est définie en Caml par une fonction u qui, à tout entier n positif, associe le terme $u_n \in \mathbb{R}$. Par exemple :

```
# let u = function n ->
            let fl_n = float_of_int n in
              3.2 *. fl_n *. (1.0 -. fl_n);;

val u : int -> float = <fun>
```

Écrire une fonction `delta` qui, à une suite $(u_n)_{n\in\mathbb{N}}$ associe la suite $(u_{n+1} - u_n)_{n\in\mathbb{N}}$.

Quelle est sa signature ?

### Ex. 5.13 – Taille de la représentation

Proposer une fonction `size_r` de signature `int -> int -> int` prenant en argument un entier b⩾ 2 et un entier n⩾ 1 et renvoyant le nombre de chiffres dans la représentation de n en base b.

### Ex. 5.14 – Récursion

**1.** Déterminer la signature et le but de la fonction `foo` suivante :

```
let rec foo f = function
  | 0 -> fun x -> x
  | n -> fun x -> foo f (n-1) (f x);;
```

On définit une suite récursive $(u_n)_{n\in\mathbb{N}}$ par $u_0$ et $u_n = f(u_{n-1})$ pour $n > 0$.

**2.** Proposer une fonction `build` de signature `'a -> ('a -> 'a) -> n -> 'a`, construite avec `foo`, qui prend en argument $u_0$ et $f$ et renvoie une fonction qui, lorsqu'on lui donne $k$, renvoie $u_k$.

### Ex. 5.15 – Nombres de Harshad

**1.** Proposer une fonction `sum_digits` prenant en argument un entier $n \geqslant 0$ et un entier $b \geqslant 2$, et renvoyant la somme des chiffres de l'écriture de $n$ en base $b$.

Un nombre de Harshad en base $b$ est un entier positif divisible par la nombre de ses chiffres.

**2.** Proposer une fonction `harshad` prenant en argument un entier $n \geqslant 0$ et un entier $b \geqslant 2$, et renvoyant un booléen indiquant si $n$ est un nombre de Harshad en base $b$.

### Ex. 5.16 – Factorielles avec des sommes

En remarquant que $(n+1)! = n! + n! + \ldots + n!$, proposer une implémentation de la fonction `factorielle` n'utilisant aucune multiplication.

### Ex. 5.17 – Réécriture

On propose la fonction C suivante :

```
int foo(int x, int y) {
    int res = 1;
    while (x<=y) {
        x++;
        y = y-x;
        res = 2*res;
    }
    return res;
}
```

Proposer une implémentation de cette même fonction en OCaml.

### Ex. 5.18 – Le retour de la composition

On définit h par :

```
let h f g x = g (f x);;
```

**1.** Donner la signature de h et préciser ce que fait cette fonction.

On définit `foo` par :

```
let foo x = h h h x;;
```

**2.** Quelle est la signature de `foo` (question difficile) ?

### Ex. 5.19 – Recherche dichotomique de racines

On considère une fonction $f$ continue sur $[a,b]$, telle que $f(a) \times f(b) \leqslant 0$. D'après le théorème des valeurs intermédiaires, il existe au moins un $x$, appelé *racine*, dans $[a,b]$ tel que $f(x) = 0$. On souhaite trouver une estimation de $x$ à $\epsilon > 0$ près.

Pour ce faire, on peut utiliser une approche dichotomique consistant à considérer le centre $m$ de l'intervalle $[a,b]$, poursuivre la recherche soit dans $[a,m]$, soit dans $[m,b]$,

selon les valeurs prises par $f$ en $a$, $m$ et $b$, et poursuivre de la sorte tant que la largeur de l'intervalle de recherche est supérieure à $2\epsilon$. Une fois que l'intervalle est suffisamment petit, on peut alors renvoyer son milieu.

Proposer une fonction `bisect` qui prend en argument $f$, $a$, $b$ et $\epsilon$, telle que « `bisect f a b eps` » retoune une approximation d'une racine de $f$ dans $[a,b]$ à $\epsilon$ près (sa signature serait donc `(float -> float) -> float -> float -> float -> float`). On pourra lever une erreur si les hypothèses ne sont pas vérifiées.

## Ex. 5.20 – Multiplication égyptienne

La multiplication égyptienne permet de calculer le produit de deux entiers $p$ et $q$ en n'utilisant que des multiplications et divisions par deux ainsi que des additions et soustractions. Elle est basée sur les propriétés suivantes :

$$p \times q = \begin{cases} \frac{p}{2} \times (2q) & \text{si } p \text{ est pair} \\ \frac{p-1}{2} \times (2q) + q & \text{si } p \text{ est impair} \end{cases}$$

En déduire une fonction récursive OCaml `produit` de signature `int -> int -> int` renvoyant le produit de deux entiers.

## Ex. 5.21 – Sommes

**1.** Proposer une fonction `sum` de signature `(int -> int) -> int -> int` prenant en argument une fonction $f$ et un entier naturel $n$ et renvoyant

$$\sum_{k=1}^{n} f(k)$$

**2.** En déduire une fonction `triangular` de signature `int -> int`, prenant en argument un entier naturel $n$ et renvoyant le $n^e$ nombre triangulaire (suite A000217 de l'OEIS), défini par

$$\sum_{k=1}^{n} k$$

**3.** De même, proposer une fonction `tetrahedral` de signature `int -> int`, prenant en argument un entier naturel $n$ et renvoyant (suite A000292 de l'OEIS)

$$p_n = \sum_{p=1}^{n} \sum_{k=1}^{p} k$$

## Ex. 5.22 – Fractale

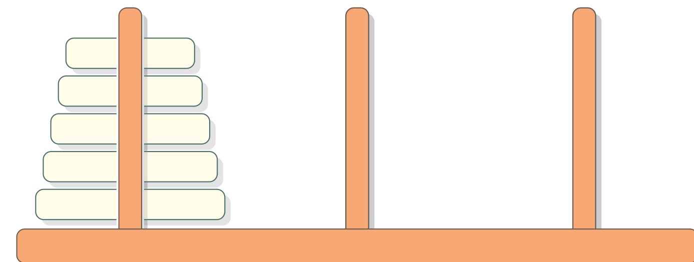
Les expressions « `print_char '#'` » et « `print_char ' '` » impriment respectivement le symbole « # » et une espace à l'écran à la position actuelle du curseur. « `print_newline ()` » provoque un retour à la ligne.

Proposer une fonction `fract` de signature `int -> unit` prenant en argument un entier `n` strictement positif dessinant une fractale dont l'ordre 4 serait :

```
        #
       ##
       #
     ####
      #
     ##
     #
########
   #
  ##
  #
####
 #
##
#
```

## Ex. 5.23 – Tours de Hanoï

On dispose de trois tiges, numérotées 1, 2 et 3. Sur la première tige se trouvent $n$ disques troués de taille distinctes, empilés de bas en haut par ordre de taille décroissante.



On dispose d'une fonction `move` de signature `int -> int -> unit` prenant en argument un entier $p$ et un entier $q$ tous deux dans $[\![1\,..\,3]\!]$ et déplaçant le disque le plus haut placé sur la tige $p$ et le déplaçant vers la tige $q$.

On souhaite déplacer tous les disques de la tige 1 vers la tige 3. Proposer une fonction `solve` de signature `int -> unit` prenant en argument $n$ et effectuant les appels à `move`

nécessaires. Par exemple, pour n= 1, la fonction fera un seul appel, « `move 1 3` ». Pour $n = 2$, il y aura trois appels, « `move 1 2` », « `move 1 3` » et enfin « `move 2 3` ».

## Ex. 5.24 – Recherche d'entiers

**1.** Proposer une fonction `first_over` de signature `(int -> int) -> int` prenant en argument une fonction $f$ définie sur $\mathbb{N}$ et telle qu'il existe $i \in \mathbb{N}$ vérifiant $f(i) > i$ et renvoyant le plus petit [55] de ces $i$.

**2.** Proposer une fonction `first_eq` de signature `(int -> 'a) -> (int -> 'a) -> int` prenant en argument deux fonctions $f$ et $g$ définies sur $\mathbb{N}$ et telles qu'il existe $i \in \mathbb{N}$ vérifiant $f(i) = g(i)$ et renvoyant le plus petit de ces $i$.

**3.** Proposer une fonction `first_inv` de signature `(int -> 'a) -> ('a -> int) -> int` prenant en argument deux fonctions $f$ et $g$ définies sur $\mathbb{N}$ et telles qu'il existe $i \in \mathbb{N}$ vérifiant $(g \circ f)(i) = i$ et renvoyant le plus petit de ces $i$.

## Ex. 5.25 – Quaternions

Les quaternions sont le plus simple exemple des *nombres hypercomplexes*, des algèbres qui vont au-delà de l'arithmétique des nombres complexes [56] Ils ont été introduits en 1843 par le mathématicien irlandais William Rowan Hamilton. De même que les complexes sont de la formes $\alpha + i\beta$ avec $i^2 = -1$, les quaternions s'écrivent $\alpha + i\beta + j\delta + k\gamma$, où les éléments $i$, $j$ et $k$ satisfont les *relations quaternioniques* :

$$i^2 = j^2 = k^2 = ijk = -1$$

Ces éléments forment un corps *non-commutatif*. Ils ont de nombreuses applications pratiques, par exemple la repsentation de rotations dans l'espace.

On suppose représenter en OCaml le quaternion $\alpha + i\beta + j\delta + k\gamma$ par le quadruplet $(\alpha, \beta, \gamma, \delta)$ (de type `float * float * float * float`). Après avoir établi à partir des relations quaternioniques les différents produits mêlant $i$, $j$ et $k$, proposer une fonction `qmul` prenant en argument deux quadruplets représentant deux quaternions et renvoyant un quadruplet représentant leur produit.

55. Dans tout l'exercice, on supposera que les entiers que l'on cherche sont suffisamment petits pour qu'il soit possible de les trouver avant d'être confrontés aux limitations de la machine (récursion, représentation, etc.)

56. D'après le théorème de Frobenius, démontré par Ferdinand Frobenius en 1877, il n'existe pas d'autres algèbres associatives à division de dimension finie sur l'ensemble des réels (à isomorphisme près). On a perdu la notion d'ordre en passant de l'ensemble des réels à l'ensemble des complexes, on perd la commutativité en passant aux quaternions, et c'est à l'associativité qu'il faudra renoncer avec les octonions.

## Ex. 5.26 – Géométrie dans l'espace

On considère les points de l'espace à coordonnées entières, et on les représente par des triplets d'entiers relatifs (`int * int * int`). Un vecteur $\overrightarrow{AB}$ entre deux tels points A et B sera également représenté sous cette forme.

**1.** Écrire une fonction `vect` prenant deux points A et B et renvoyant le vecteur $\overrightarrow{AB}$.

**2.** Écrire une fonction `same_len` prenant deux vecteurs et renvoyant un booléen indiquant s'ils ont même norme.

**3.** Écrire une fonction `colinear` prenant deux vecteurs et renvoyant un booléen indiquant s'ils sont colinéaires.

**4.** Écrire une fonction `tetrahedron` prenant quatre points et renvoyant un booléen indiquant s'ils forment un tétraèdre régulier.

**5.** Écrire une fonction `coplanar` prenant quatre points et renvoyant un booléen indiquant s'ils sont coplanaires.

# 6 Structures de données

« Bad programmers worry about the code. Good programmers worry about data structures and their relationships. »

— *Linus Torvalds*

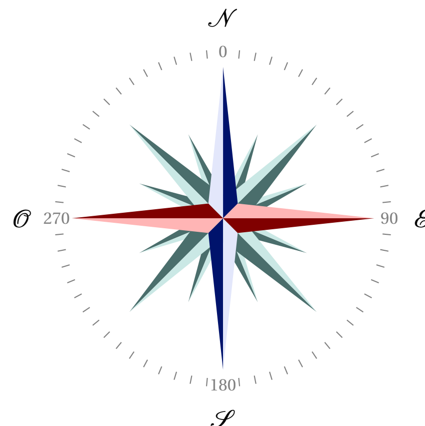
## 1 Les types construits en OCaml

### 1.1 Définir ses propres types

Les avantages à définir ses propres types en informatique ne manquent pas. Ils aident à la compréhension du code, en clarifiant les objets manipulés, et permettent d'encapsuler plusieurs informations en un seul objet. Cela permet notamment de réduire le nombre d'arguments à transmettre à une fonction, et de renvoyer des informations plus complexes. Cette possibilité est offerte dans la quasi-totalité des langages. Nous nous pencherons dans ce chapitre sur les possibilités offertes par OCaml dans un premier temps, puis nous ferons le lien avec ce que propose le langage C.

### 1.2 Types « somme » (énumérations, unions)

Supposons que l'on souhaite représenter une *direction* à partir d'un endroit sur Terre, telle que celles que l'on peut s'imaginer sur un compas ou une rose des vents. Puisqu'il ne s'agit guère que d'un angle, il serait naturellement possible de représenter un telle direction par un simple nombre flottant (traditionnellement, 0° représente le nord, et les angles sont mesurés par rapport au nord dans le sens indirect).




Mais pour pleinement profiter du mécanisme de typage de OCaml et clarifier les programmes manipulant de telles directions, disposer d'un type spécifique peut se révéler plus intéressant.

Pour définir un type en OCaml, on utilise le mot-clé « `type` », suivi du nom qui désignera ce type, du symbole « `=` » puis de sa définition, laquelle peut prendre plusieurs formes.

Une première possibilité consiste à définir un type comme une *énumération* entre plusieurs « *constantes* » que l'on précise. Ces constantes sont de simples identifiants commençant par une majuscule. Le symbole « `|` » joue le rôle de « ou » logique pour séparer chacune des constantes. Par exemple, on peut définir un type `direction` représentant les quatre points cardinaux comme une énumération de quatre constantes [1] :

```
# type direction = North | East | South | West;;
```

À présent, `North`, `East`, `South` et `West` sont les représentants du type `direction` :

```
# let dir = West;;
val dir : direction = West
```

De la même façon, le type « `bool` » du langage OCaml est simplement un type énuméré avec deux constantes, `true` et `false` (même si le langage a pris une petite liberté et écrit ces constantes avec des majuscules).

Pour un peu plus d'expressivité, on peut définir des types appelés « *union* » en OCaml qui sont une généralisation des énumérations. Outre des constantes, on peut également définir des *constructeurs*, associés à un type quelconque. Par exemple, pour pouvoir manipuler des directions plus précises que seuls quatre points cardinaux, on peut également considérer une direction via son angle sur la rose des vents. Pour ce faire, on peut ajouter aux points cardinaux précédents, un constructeur `Angle` paramétré par un `float`, de la sorte :

```
# type direction = North | East | South | West | Angle of float;;
```

Un constructeur se comporte essentiellement comme une fonction [2] permettant de construire des objets du type leur étant associé. Le constructeur `Angle` prend ainsi en argument un flottant et renvoie un object de type `direction` :

```
# let dir = Angle 45.0;;
val dir : direction = Angle 45.
```

Les déclarations de types peuvent être récursives, ce qui permet de définir des directions comme étant les médianes de deux directions, en écrivant :

```
# type direction = North | East | South | West | Angle of float
                 | Mediane of direction * direction;;
```

1. La réponse de l'interpréteur interactif OCaml pour une déclaration de type, lorsqu'elle est syntaxiquement correcte, consiste juste à afficher le type nouvellement défini. On l'omettra donc.
2. Il ne s'agit cependant pas, dans la version OCaml du langage, d'une fonction. Ainsi, le constructeur `Angle` ne peut être l'argument d'une fonction attendant un élément de type « `float -> direction` ».

Plus précisément, on considérera que la médiane de deux directions est la direction « médiane » de l'angle inférieur à 180 ° formé par les deux directions [3]. Ce qui permet de définir très simplement d'autres directions à partir de directions existantes [4] :

```
# let se = Mediane (South, East);;
val se : direction = Mediane (South, East)

# let sse = Mediane (Angle 180.0, se);;
val sse : direction = Mediane (Angle 180., Mediane (South, East))
```

Là encore, OCaml a défini, en même temps que le type, un constructeur `Mediane` qui prend un couple de deux objets de type `direction` et renvoie un objet de type `direction`.

Les opérateurs d'égalité « `=` » et « `<>` » permettent de comparer des objets de types « somme ». Pour que deux objets soient égaux, il faut qu'ils correspondent à la même constante (`North = North`) :

```
# North = North;;
- : bool = true
```

Ou au même constructeur et que les valeurs associées soient égales entre elles (si ces valeurs sont elles-mêmes des objets complexes, ce principe s'appliquant récursivement, comme dans le second exemple ci-dessous) :

```
# Angle 42. = Angle 42.;;
- : bool = true

# Mediane (Angle 42., East) = Mediane (Angle 42., East);;
- : bool = true
```

Attention toutefois, OCaml ne peut bien évidemment pas deviner nos intentions, et comprendre que deux objets différents correspondent à la même direction :

```
# Angle (-90.) = Angle 270.;;
- : bool = false

# South = Angle 180.;;
- : bool = false

# Mediane (South, East) = Mediane (East, South);;
- : bool = false
```

3. On supposera que ces deux directions ne sont pas opposées.
4. Les noms de ces nouvelles directions commencent par des minuscules, car il ne s'agit pas de constructeurs ou de valeurs déclarés à l'intérieur d'une définition de type.

Pour pouvoir comparer deux objets, il faut revenir à une grandeur commune, par exemple un angle avec la direction du nord, ramené dans l'intervalle [5] $[-180.0, 180.0[$. On écrit tout d'abord une fonction pour ramener un angle dans l'intervalle adéquat [6] :

```
# let rec modulo = function
    | x when x >=  180. -> modulo (x -. 360.)
    | x when x <  -180. -> modulo (x +. 360.)
    | x                 -> x;;

val modulo : float -> float = <fun>
```

Puis, on peut écrire une fonction qui transforme tout objet de type `direction` en un angle de cet intervalle. On manipule les objets de type « somme » grâce aux fonctions de filtrage du langage. Une solution peut être :

```
# let rec calc_angle = function
    | Angle a               -> modulo a
    | North                 -> 0.0
    | East                  -> 90.0
    | South                 -> -180.0
    | West                  -> -90.0
    | Mediane (dir1, dir2)  ->
        let angle1 = calc_angle dir1 and angle2 = calc_angle dir2
          in match modulo (angle2 -. angle1) with
              | 180.0 -> failwith "Opposing directions"
              | diff  -> modulo (angle1 +. diff /. 2.);;

val calc_angle : direction -> float = <fun>
```

On s'intéressera tout particulièrement, dans la fonction précédente, à la manière dont la médiane est calculée : on ne peut faire sans précaution la moyenne des angles car la médiane des directions sud-est et sud-ouest conduirait, de façon erronée, à la direction nord ! On préfère faire la moitié du chemin sur la rose des vents en allant de la première direction vers la seconde, dans le sens adéquat. Dans le cas de directions opposées, lorsque la médiane n'est pas définie, on déclenche une erreur avec « `failwith "Directions opposées"` ».

Une telle fonction permet ainsi de calculer l'angle avec le nord de la direction sse :

```
# calc_angle sse;;
- : float = 157.5
```

5. On aurait pu préférer $[0.0, 360.0[$, mais $[-180.0, 180.0[$ sera préférable pour gérer les médianes.
6. Cette fonction peut être coûteuse si l'angle x est loin de l'intervalle requis. Il existe des alternatives plus efficaces d'écrire une telle fonction, mais nous laisserons ce problème de côté pour l'instant.

Construire une fonction testant l'égalité de deux directions est alors simple :

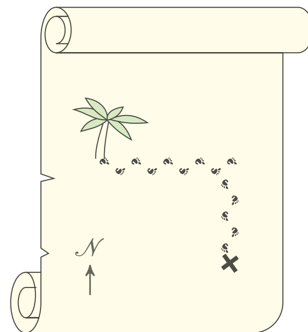
```
# let directions_equals dir1 dir2 =
    calc_angle dir1 = calc_angle dir2;;

val directions_equals : direction -> direction -> bool = <fun>
```

Elle donne bien les résultats attendus :

```
# directions_equals (Angle (-90.)) (Angle 270.);;
- : bool = true

# directions_equals South (Angle 180.);;
- : bool = true
```

## 1.3 Types « produit » (enregistrements)

Intéressons-nous à présent à une façon de représenter un déplacement à la surface de la Terre. Sur de petites distances, cela peut être décrit comme la combinaison d'un déplacement est-ouest et d'un déplacement nord-sud (cela peut permettrait également de définir une position si l'on dispose d'une origine).



On pourrait évidemment utiliser un simple couple de flottants (`float * float`) pour ce faire, mais on perd les avantages du système de typage, et il faut alors prendre garde à ne pas se tromper dans l'ordre des deux valeurs dans ce couple.

Par commodité, le langage OCaml propose un type « *enregistrement* » qui est une variation des n-uplets où chaque élément est identifié par un nom (fréquemment appelé *étiquette*).

Pour définir un type « enregistrement », on place, entre accolades, une liste des étiquettes et des types qui leurs sont associés, séparés par des points-virgules. Un type « déplacement », constitué de deux flottants, identifés par des étiquettes `e_w` et `n_s`, peut par exemple être déclaré de la façon suivante :

```
# type displacement = { e_w : float; n_s : float };;
```

On supposera dans la suite que le déplacement est-ouest est compté positivement vers l'est, le déplacement nord-sud vers le nord. On a donc l'équivalent du repère $(\vec{e}_x, \vec{e}_y)$ utilisé fréquemment en mathématiques.

Pour créer un objet de ce type, on renseigne les différents « *champs* », en utilisant le signe « = » cette fois pour séparer l'étiquette et la valeur qui lui est associée :

```
# let d = { e_w = 4.5;  n_s = 10.2 };;
val d : displacement = {e_w = 4.5; n_s = 10.2}
```

Notons que l'ordre d'apparition des étiquettes lors de la création n'a pas d'importance :

```
# let d = { n_s = 10.2; e_w = 4.5 };;
val d : displacement = {e_w = 4.5; n_s = 10.2}
```

Mais il faut impérativement renseigner tous les champs :

```
# let d = { e_w = 4.5; };;

Characters 12-25:
  let d = { e_w = 4.5; };;
          ^^^^^^^^^^^^^
Error: Some record field labels are undefined: n_s
```

La présence des étiquettes favorise également la récupération des informations. Il suffit en effet de faire suivre le nom désignant un objet de type « enregistrement » par un point suivi du nom d'une étiquette pour accéder à la valeur associée à cette étiquette :

```
# d.e_w;;
- : float = 4.5
```

Alternativement, on peut utiliser le mécanisme de filtrage du langage, comme la fonction ci-dessous qui extrait le déplacement est-ouest d'un objet de type `deplacement` :

```
# let displ_e_w =
    function { e_w = d; n_s = _ } -> d;;

val displ_e_w : displacement -> float = <fun>
```

On peut à présent écrire une fonction déterminant le déplacement qui correspond à une distance (représentée par un flottant) associée à une direction :

```
# let calc_position dist dir =
    let pi = 3.1415926535897932 in
    let angle = (calcAngle dir) *. pi /. 180.
      in { e_w = dist *. sin(angle); n_s = dist *. cos(angle) };;

val calc_position : float -> direction -> displacement = <fun>
```

Ainsi, un déplacement de 50 m vers le sud-sud-est

```
# calc_position 50.0 sse;;
- : displacement = {e_w = 19.134171618254495; n_s = -46.193976625564339}
```

correspond donc à un déplacement d'environ 19 m vers l'est et 46 m vers le sud !

# 2 Listes chaînées

## 2.1 Les listes en informatique

La nécessité de manipuler de grandes quantités de données en informatique conduit naturellement au besoin d'objets capable de rassembler de nombreux éléments. Nous avons déjà étudié la notion de *tableaux*, qui permettent de stocker un nombre potentiellement important de données dans un même objet.

Les tableaux, en informatique, peuvent être décrits comme des ensembles *ordonnés* d'éléments, très souvent de même type, identifiés par un index (généralement entre 0 à $n-1$), pour lequels il est possible d'accéder efficacement[7] à un quelconque élément à partir de son index (comme pour la mémoire, on parle d'accès *aléatoire*, puisque les index des éléments auxquels on accèdent ne respectent pas de règle particulière). En général, les tableaux ont une taille qui n'évolue pas dans le temps.

Bien souvent, il est pratique de disposer de structures dans lesquelles on peut aisément ajouter ou retirer des éléments. En informatique, on appelle *liste* une structure de données contenant un ensemble *ordonné* d'éléments (généralement tous de même type) permettant l'ajout ou le retrait d'éléments. Puisque les listes se distinguent des tableaux par ces opérations d'ajout et de retrait, ces opérations doivent être efficaces (en temps constant ou logarithmique, donc). On s'attend à pouvoir aisément itérer sur l'ensemble des éléments d'une liste, mais on va peut-être être amenés à renoncer à d'autres possibilités, par exemple à un accès aléatoire efficace au éléments.

Il existe des dizaines de manières de créer des structures informatiques permettant l'ajout et le retrait d'éléments, et chacune a ses spécificités propres : sur la disponibilité ou non des différentes opérations ou leur complexité temporelle. Les solutions les plus courantes pour implémenter des listes sont les *tableaux redimensionnables*[8] et les *listes chaînées*, mais d'autres solutions sont envisageables.

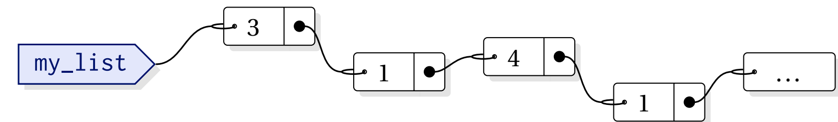
Les listes chaînées ont ceci d'intéressant que les complexités des différentes opérations sont très différentes de celles sur les tableaux « classiques ». Elles répondent donc à des besoins très différents. Il n'est pas rare que, dans un ouvrage, le terme « liste » fasse directement référence à des listes chaînées, ce qui occasionne quelques malentendus de langage. Par exemple, vous lirez peut-être « les listes Python ne sont pas des listes ». Il faut comprendre que, dans l'implémentation usuelle, les listes Python ne sont pas implémentées comme des listes chaînées[9]. Mais il s'agit bien de listes au sens générique du terme puisqu'il est bien possible d'ajouter et retirer des éléments (avec `List.append` et `List.pop` par exemple).

7. Généralement en temps constant $O(1)$, mais une structure permettant un accès en $O(\log n)$ à un élément à partir de son index serait considérée comme suffisamment efficace.

8. Qui correspondent à l'implémentation standard des listes dans le langage Python, et sur lesquels nous reviendrons un peu plus tard.

## 2.2 Principe des listes chaînées

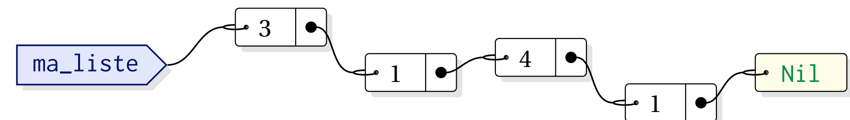
Supposons tout d'abord que l'on souhaite représenter une liste d'entiers. Une liste chaînée d'entiers est constituée d'un ensemble de « cellules » contenant une valeur entière, et un moyen d'accéder à la cellule suivante dans la liste. Schématiquement, cela correspond à la situation suivante :




Ce qui suit un élément, dans la liste d'entiers, est également une liste d'entiers. On peut donc envisager de décrire une liste d'entiers par un type « enregistrement » tel que :

```
# type int_clist = { value : int; next : int_clist };;
```

L'ennui avec la définition précédente est que les listes n'ont pas de fin ! Pour pouvoir décrire une liste avec un nombre fini d'éléments, on doit pouvoir avoir un « bouchon », une constante qui indiquera l'extrémité de la liste. Par exemple, pour une liste de quatre entiers :




Dans la description du type, l'étiquette `next` doit donc pouvoir désigner, soit une cellule constituée d'un entier et d'un suivant, soit une valeur particulière servant de « bouchon ». En informatique, il est d'usage d'utiliser le terme « `Nil` » pour faire référence à un tel bouchon. Cela vient du latin *nihil* signifiant « *néant, rien* ».

On peut donc définir notre type pour recueillir des listes de la façon suivante :

```
# type int_clist = Nil | Cell of int_cell
  and int_cell = { value : int ; next : int_clist };;
```

9. L'implémentation CPython actuelle utilise des tableaux redimensionnables.

Dans cette définition, « int_cell » construit un type enregistrement, utilisé ensuite dans la définition de « int_clist » de type union. Le constructeur Cell permet de construire un objet de type int_clist à partir d'un objet de type int_cell, et donc de différencier une cellule de l'étiquette Nil identifiant la fin de la liste. Il est indispensable de définir ces deux types dans une même déclaration de types car chacun de ces deux types fait référence à l'autre.

Il n'est toutefois pas indispensable de définir explicitement le type int_cell et on peut tout aussi bien se contenter de :

```
# type int_clist = Nil | Cell of { value : int ; next : int_clist };;
```

Ceci fait, on peut ensuite définir une liste :

```
# let my_list = Cell { value = 3 ; next =
                 Cell { value = 1 ; next =
                   Cell { value = 4 ; next =
                     Cell { value = 1 ; next =
                       Nil } } } };;

val my_list : int_clist =
  Cell
   {value = 3;
    next =
     Cell
      {value = 1;
       next =
        Cell
         {value = 4; next = Cell {value = 1; next = Nil}}}}
```

Ce n'est pas une façon très pratique de définir notre liste. On préférera généralement enfiler les éléments un par un comme des perles sur un fil. On peut aisément définir une fonction qui ajoute un élément à *gauche* de la liste :

```
# let addLeft elem lst =
      Cell { value = elem ; next = lst };;

val addLeft : int -> int_clist -> int_clist = <fun>
```

Pour construire la liste, on peut alors commencer par créer une liste vide :

```
# let my_list = Nil;;
val my_list : int_clist = Nil
```

Puis ajouter les éléments un par un, de la droite vers la gauche [10] :

```
# let my_list = addLeft 1 my_list;;
val my_list : int_clist = Cell {value = 1; next = Fin}

# let my_list = addLeft 4 my_list;;
val my_list : int_clist =
  Cell {value = 4; next = Cell {value = 1; next = Fin}}

# let my_list = addLeft 1 my_list;;
val my_list : int_clist = ...

# let ma_liste = addLeft 3 my_list;;
val my_list : int_clist = ...
```

On peut également tout faire en une seule définition :

```
# let my_list = addLeft 3 (addLeft 1
                     (addLeft 4 (addLeft 1 Nil)));;
val my_list : int_clist = ...
```

On peut ensuite vouloir écrire des fonctions qui agissent sur la liste chaînée. Comme il s'agit d'une union, on travaillera par filtrage. Un nom désignant une liste peut ainsi désigner deux choses, d'après le type int_clist :

- un objet de type int_cell, identifié par le constructeur Cell, contenant un champ **value** (de type **int**) et un champ next (de type int_clist) ;
- la constante Nil, indiquant la terminaison de la liste.

Les fonctions agissant sur des objets liste_int vont donc généralement utiliser un filtrage correspondant aux deux cas du dessus. Par exemple, le seul élément directement accessible est l'élément en tête de liste, et il est aisé de l'obtenir.

```
# let head = function
    | Cell { value = v ; next = _ } -> v
    | Nil                           -> failwith "Liste vide !";;

val head : int_clist -> int = <fun>
```

On remarquera la façon dont le filtrage nous permet d'accéder aux différents éléments du type enregistrement.

10. Rappelons qu'ici on ne modifie pas le « contenu » de my_list puisque c'est impossible en OCaml, on crée successivement des objets de même nom, mais comme la dernière définition masquera les autres, le résultat essentiellement le même. On a raccourci les réponses de l'interpréteur.

On pourrait également écrire [11] :

```
# let head = function
    | Cell c -> c.value
    | Nil    -> failwith "Liste vide !";;

val head : int_clist -> int = <fun>
```

Pour obtenir une nouvelle liste chaînée, correspondant à la liste en argument privée de son premier élément, c'est également assez simple :

```
# let tail = function
    | Cell { value = _ ; next = lst } -> lst
    | Nil                             -> failwith "Liste vide !";;

val tail : int_clist -> int_clist = <fun>
```

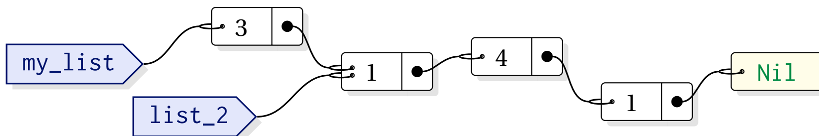
Ou bien, de façon équivalente :

```
# let tail = function
    | Cell c -> c.next
    | Nil    -> failwith "Liste vide !";;

val queueListe : int_clist -> int_clist = <fun>
```

Il faut bien voir ici que l'on obtient une *nouvelle* liste, qui partage ses éléments avec sa parente. Par exemple, si l'on écrit :

```
# let list_2 = tail my_list;;
val list_2 : int_clist = ...
```

on se retrouve avec deux listes, identifiées par les noms « `my_list` » et « `list_2` », qui correspondent en mémoire à quelque chose de ce genre :




La liste désignée par le nom « `list_2` » correspond bien à la séquence d'entiers « 1, 4, 1 ». Les « `int_cell` » de cette liste sont partagées avec la liste désignée par « `my_list` ». Ce détail est sans importance ici car les objets manipulés ne peuvent pas être altérés, mais cela aura de l'importance dans la suite.

11. C'est essentiellement un choix de style, vous pouvez choisir celui avec lequel vous êtes le plus à l'aise.

Obtenir le dernier élément de la liste est un peu plus difficile, mais ce n'est pas insurmontable en utilisant la récursion :

```
# let rec last = function
    | Cell { value = v ; next = Nil } -> v
    | Cell { value = _ ; next = lst } -> last lst
    | Nil                             -> failwith "Liste vide !";;

val last : int_clist -> int = <fun>
```

Ou bien encore :

```
# let rec last = function
    | Cell c when c.next = Fin -> c.value
    | Cell c                   -> last c.next
    | Nil                      -> failwith "Liste vide !";;

val last : int_clist -> int = <fun>
```

De la même façon, on peut obtenir la longueur de la liste :

```
# let rec length = function
    | Cell { value = _ ; next = lst } -> length lst + 1
    | Nil                             -> 0;;

val length : int_clist -> int = <fun>
```

Ou bien

```
# let rec length = function
    | Cell c -> length c.next + 1
    | Nil    -> 0;;

val length : int_clist -> int = <fun>
```

L'un des inconvénients de notre type liste est qu'il ne peut contenir que des entiers. On peut faire un peu mieux, et définir un type mémorisant des éléments certes tous de même type, mais qui pourra être différent d'une liste à l'autre. Pour ce faire, on va indiquer que notre type `liste` est « paramétré » par un autre type en faisant précéder sa définition de « `'a` » :

```
# type 'a clist = Nil | Cell of 'a cell
   and 'a cell  = { value : 'a ; next : 'a clist };;
```

Un type défini comme « `'a clist` » permet donc de définir des objets de type « `int clist` », « `float clist` », « `bool clist` »... ou même « `int clist clist` » (une liste de listes d'entiers) ! Lorsque l'on définit `my_list` (comme précédemment), le type indique explicitement qu'elle contient des entiers :

```
my_liste : int clist = Cell {value = 3; next = ...
```

Mais il est, à présent, possible de créer aussi une liste de flottants avec la même déclaration de type :

```
my_liste : float liste = Cell {value = 3.0; next = ...
```

Toutes les fonctions définies précédemment restent correctes, sous réserve de les définir *après* la définition du type `'a clist`, mais leur typage va bien évidemment changer ! Par exemple, pour la fonction construisant une nouvelle liste par ajout d'un élément à gauche d'une liste existante :

```
val addLeft : 'a clist -> 'a -> 'a clist = <fun>
```

# 3 Les listes OCaml

## 3.1 Création et manipulation

Nous n'irons pas plus loin dans cette direction car... le langage OCaml dispose de son propre type `'a list` pour décrire des listes chaînées d'éléments, et dans la suite, nous utiliserons celui-ci. L'implémentation est toutefois similaire à celle décrite précédemment, ce qui nous permettra de mieux comprendre ce qui se passe.

Les listes OCaml sont donc des conteneurs immutables pouvant recueillir un nombre quelconque d'éléments, dont le type peut être librement choisi. Elles sont représentées entre crochets, les éléments étant séparés par des points-virgules [12] :

```
# let my_liste = [ 1; 2; 3 ];;
val my_list : int list = [1; 2; 3]

# let my_list = [ [ 1.41; 3.14 ]; [ 1.0; 2.0; 3.0 ]; [] ];;
val my_list : float list list = [[1.41; 3.14]; [1.; 2.; 3.]; []]

# let my_list = [ sin; cos ];;
val my_list : (float -> float) list = [<fun>; <fun>]
```

12. Attention, la liste `[ 1, 2, 3 ]` est une liste à un seul élément, un tuple (de type `int * int * int`).

Il reste impératif que tous les éléments d'une liste soient du même type :

```
# let my_list = [ 1; 2.3; 5 ];;

Characters 21-24:
  let my_list = [ 1; 2.3; 5 ];;
                     ^^^
Error: This expression has type float
    but an expression was expected of type int
```

On dispose d'un nouvel opérateur, noté « `::` » et généralement appelé « *conse* », qui permet de construire une (nouvelle) liste en « accrochant » un élément accroché à gauche d'une liste existante d'éléments *de même type* :

```
# 1 :: [ 2; 3; 4; 5 ];;
- : int list = [1; 2; 3; 4; 5]
```

Comme nous l'avons vu en créant nos propres listes chaînées, il n'est pas possible d'ajouter simplement un élément à droite d'une liste existante. Il en est de même avec les listes OCaml, et il n'est donc pas permis d'écrire :

```
# [ 2; 3; 4; 5 ] :: 6;;

Characters 20-21:
  [ 2; 3; 4; 5 ] :: 6;;
                    ^
Error: This expression has type int
    but an expression was expected of type int list list
```

De même, on dispose d'un opérateur de concaténation de deux listes (contenant des éléments de même type), noté « `@` » :

```
# [ 1; 2; 3 ] @ [ 5; 6 ];;
- : int list = [1; 2; 3; 5; 6]
```

La liste vide est simplement désignée par `[]` et son type, faute d'élément, est `'a list`. Il est possible de l'associer à des éléments de tous types avec « `::` » :

```
# let my_list = [];;
val my_list : 'a list = []

# let my_list = 1 :: my_list;;
val my_list : int list = [1]
```

La fonction[13] `List.hd` permet d'obtenir le premier élément d'une liste :

```
# List.hd;;
- : 'a list -> 'a = <fun>

# List.hd [ 1; 2; 3; 4 ];;
- : int = 1
```

De même, la fonction `List.tl` renvoie une nouvelle liste, correspondant à la liste passée en argument privée de son premier élément[14] :

```
# List.tl;;
- : 'a list -> 'a list = <fun>

# List.tl [ 1; 2; 3; 4 ];;
- : int list = [2; 3; 4]
```

Notons que la « queue » d'une liste d'entiers sera toujours une liste d'entiers, quand bien même elle serait vide. On ne pourra pas insérer autre chose qu'un entier dans une telle liste. Cette restriction est indispensable pour que le mécanisme de typage puisse fonctionner correctement.

```
# let my_list = [];;
val my_list : 'a list = []

# let m_list = 1 :: lst;;
val my_list : int list = [1]

# let my_list = List.tl my_list;;
val my_list : int list = []

# 3.14 :: my_list;;

Characters 9-12:
  3.14 :: my_list;;
          ^^^
Error: This expression has type int list
    but an expression was expected of type float list
```

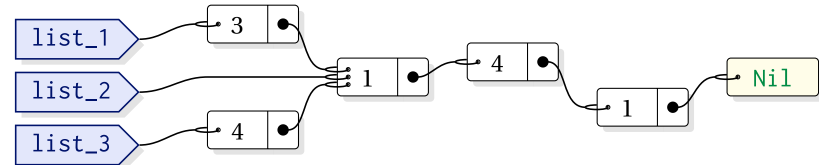
13. Les noms hd et `tl` sont des abréviations des termes anglais « head » et « tail », désignant respectivement la tête et la queue.
14. Aucun élément n'a été « enlevé » à la liste, immutable, en argument, il s'agit bien d'une *partie* de cette dernière.

## 3.2 Immutabilité des listes

Il convient de bien garder en tête que les listes étant des objets immuables, l'utilisation de l'opérateur *conse* « `::` » ou de la fonction `List.tl` produisent de nouveaux objets, mais que les listes (allogées ou raccourcies) qui en résultent sont créées sans que les éléments qui les constituent ne soient recopiés, plusieurs listes peuvent partager les mêmes éléments. Considérons par exemple la séquence d'instructions suivante :

```
# let list_1 = [ 3; 1; 4; 1 ];;
val list_1 : int list = [3; 1; 4; 1]

# let list_2 = List.tl list_1;;
val list_2 : int list = [1; 4; 1]

# let list_3 = 4 :: list_2;;
val list_3 : int list = [4; 1; 4; 1]
```

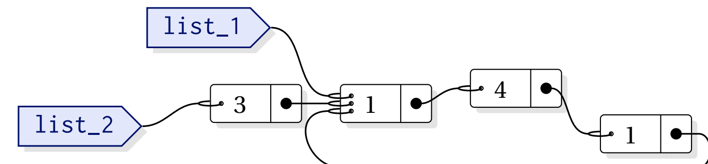
Le résultat, en mémoire, est quelque chose qui s'apparente à cette construction :




Conjugué avec le **`let rec`**, on peut même possiblement[15] construire des objets s'apparentant à des « listes sans fin » (que l'on ne considera pas comme listes). Par exemple[16] :

```
# let rec list_1 = 1::4::1::list_1;;
val list_1 : int list = [1; 4; 1; <cycle>]

# let list_2 = 3::liste;;
val list_2 : int list = [3; 1; 4; 1; <cycle>]
```

Ces définitions correspondent à la situation suivante :




15. La documentation actuelle ne garantit pas que cette possibilité sera supportée.
16. On remarquera que l'interpréteur OCaml détecte la boucle dans la liste et indique `<cycle>` plutôt qu'une infinité de termes, même s'il ne précise pas quels sont les termes qui sont répétés.

## 3.3 Autres fonctions sur les listes

Le langage OCaml fournit quelques autres fonctions élémentaires sur les listes. Tout d'abord, la fonction `List.length` permet d'obtenir le nombre d'éléments qu'elle contient :

```
# List.length;;
- : 'a list -> int = <fun>

# List.length [ 1; 2; 3; 4 ];;
- : int = 4
```

La fonction `List.nth` permet quant à elle d'obtenir l'élément d'index `n` de la liste :

```
# List.nth;;
- : 'a list -> int -> 'a = <fun>

# List.nth [ 1; 2; 3; 4 ] 3;;
- : int = 4
```

La fonction `List.mem` permet de tester l'appartenance d'un élément dans une liste. Ou plutôt, pour être plus précis, l'égalité entre un élément de la liste et l'élément fourni en paramètre. :

```
# List.mem;;
- : 'a -> 'a list -> bool = <fun>

# List.mem 42 [ 1; 2; 3; 4 ];;
- : bool = false
```

Enfin, la fonction `List.rev` permet d'obtenir une *nouvelle* liste contenant les mêmes éléments que la liste passée en argument, mais dans l'ordre inverse :

```
# List.rev;;
- : 'a list -> 'a list = <fun>

# List.rev [ 1; 2; 3; 4 ];;
- : int list = [4; 3; 2; 1]
```

Profitons de l'occasion pour signaler qu'il existe une fonction [17] `List.cons`, de signature `'a -> 'a list -> 'a list`, qui effectue la même opération que l'opérateur « `::` », et une fonction `List.append` de signature `'a list -> 'a list -> 'a list` qui est le pendant de l'opérateur « `@` » (et donc effectue une concaténation de deux listes).

17. Nous verrons que l'on peut également construire une fonction à partir d'un opérateur en le plaçant entre parenthèses : ainsi, « `(::)` » se comporte comme `List.cons` et « `(@)` comme `List.append`.

## 3.4 Écrire des fonctions sur les listes

Pour écrire des fonctions travaillant sur des listes OCaml on procédera de façon très similaire avec ce que nous avons écrit pour nos propres listes chaînées. On utilisera ainsi largement le filtrage par motif.

Comme il est fréquent de vouloir identifier des listes non vides, et d'en récupérer la tête et la queue, le langage permet d'utiliser « `::` » dans les motifs de filtrage. Ainsi, le motif « `h::t` » peut être mis en relation avec n'importe quelle liste non-vide, le nom `h` désignant ensuite l'élément en tête de liste, et le nom `t` le reste de la liste.

Par exemple, la fonction `List.hd`, renvoyant le premier élément d'une liste peut s'implémenter de la sorte :

```
# let head = function
    | []   -> failwith "Liste vide"
    | h::_ -> h;;

val head : 'a list -> 'a = <fun>
```

De même, la fonction `List.tl` renvoyant une liste correspondant à la liste placée en argument privée de son premier élément, on pourrait écrire :

```
# let tail = function
    | []   -> failwith "Liste vide"
    | h::t -> t;;

val tail : 'a list -> 'a list = <fun>
```

Il s'agit d'une particularité de « `::` ». Il n'est généralement pas possible d'utiliser d'autres opérateurs dans les motifs de filtrage. Ainsi « `lst1@lst2` », ou « `p+q` » (où `p` et `q` seraient deux entiers) ne sont pas des motifs valides. Mais ce n'est pas surprenant : la décomposition ne serait pas unique ! Tandis que l'écriture « `h::t` » ne conduit à aucune ambiguité sur `h` et `t`.

On peut par ailleurs utiliser plusieurs « `::` » dans un filtrage. Par exemple, la fonction suivante permet d'obtenir le deuxième élément d'une liste :

```
# let second = function
    | h1::h2::t -> h2
    | _         -> failwith "Liste trop courte";;

val second : 'a list -> 'a = <fun>
```

Une liste contenant un seul élément peut être reconnue de différentes façons. On peut utiliser le motif« `h::[]` », ou bien le motif « `[h]` ». Dans les deux cas, l'unique élément de la liste sera identifié par `h` dans l'expression qui suit.

On peut ainsi écrire une fonction renvoyant le dernier élément d'une liste en distinguant les éléments qui sont suivis d'autres éléments et celui qui se trouve en dernière position :

```
# let rec last = function
    | h::[] -> h
    | _::t  -> last t
    | []    -> failwith "Liste vide";;

val last : 'a list -> 'a = <fun>
```

De la même façon, pour identifier une liste à deux éléments, de nombreux motifs sont possibles : « `h1::h2::[]` », « `[h1;h2]` », ou même « `h1::[h2]` » !

Pour écrire une fonction renvoyant, comme `List.length`, la longueur d'une liste passée en argument, on peut écrire :

```
# let rec length = function
    | [] -> 0
    | _::t -> 1 + length t

val length : 'a list -> int = <fun>
```

Pour une fonction similaire à `List.nth` renvoyant le $n^e$ élément d'une liste, on écrira par exemple [18] :

```
# let rec nth lst i =
    match lst with
      | h::_ when i=0 -> h
      | _::t -> nth t (i-1)
      | [] -> failwith "Index incorrect"

val nth : 'a list -> int -> 'a = <fun>
```

On peut également définir une fonction `membre` indiquant, comme `List.mem`, si le premier élément est présent dans la liste correspondant au second argument :

```
# let rec member x = function
    | []              -> false
    | h::_ when h=x -> true
    | _::t            -> member x t;;

val member : 'a -> 'a list -> bool = <fun>
```

18. Comme la liste n'est pas en premier argument, nous utilisons `match` pour la filtrer, mais on pourrait envisager d'autres solutions.

C'est l'occasion de revenir sur un point important concernant le filtrage : un nom ne peut pas faire référence à une valeur dans un motif. Aussi ne peut-on écrire :

```
# let rec member x = function
    | []   -> false
    | x::t -> true               (* <- Attention, INCORRECT ! *)
    | _::t -> member x t;;
```

Ou plus précisément, on « peut » l'écrire, mais cela ne correspond pas à ce que l'on pourrait espérer, comme en témoignent l'avertissement et la signature de la fonction :

```
Characters 86-90:
    | _::t -> member x t;;
      ^^^^
Warning 11: this match case is unused.

val member : 'a -> 'b list -> bool = <fun>
```

En effet, le nom « x » qui apparaît dans le motif filtrage est distinct du nom x qui identifie le premier paramètre (un nom apparaîssant dans un motif de filtrage est un nom qui n'existe que le temps du motif et de sa conséquence, et qui masquera tout autre nom identique).

Enfin, pour une fonction qui concatène deux listes :

```
# let rec concat lst1 lst2 =
    match lst1 with
      | []   -> lst2
      | h::t -> h::(concat t lst2);;

val concat : 'a list -> 'a list -> 'a list = <fun>
```

## 3.5 Coût en temps des opérations sur les listes

Comme dans n'importe quel langage, il sera important de se poser la question de l'efficacité des fonctions que l'on écrit, notamment de leur complexité temporelle. Pour ce faire, il nous faut clarifier les complexités des opérations élémentaires que l'on vient de présenter.

Naturellement, la fonction `List.hd` renvoyant l'élément en tête d'une liste a une complexité constante ($\Theta(1)$). Ce qui peut être moins évident, c'est que la fonction `List.tl` a également une complexité temporelle constante ($\Theta(1)$). En effet, elle ne construit pas une nouvelle liste, mais se contente d'avancer d'un cran dans la liste et de renvoyer l'« endroit » auquel débute la liste lorsqu'on l'a privée de son premier élément.

Pour la même raison, l'opérateur « `::` » a également une complexité constante : il ne crée pas de nouvelle liste, il se contente de créer une « cellule » supplémentaire avec l'élément ajouté, et qui pointe vers la liste existante. Il en va de même pour « `::` » utilisé dans un motif de filtrage (puisqu'il s'agit ni plus ni moins que d'utiliser `List.hd` et `List.tl`).

En revanche, d'autres opérations sont plus coûteuses. `List.length` a un coût linéaire ($\Theta(n)$) en la longueur $n$ de la liste. En effet, la longeur de la liste n'est pas mémorisée par la structure de liste chaînée, et si l'on souhaite la connaître, il faut parcourir toutes les cellules de la liste en les comptant jusqu'à parvenir à l'étiquette indiquant la fin de la liste.

Pour la même raison, la fonction `List.nth` a un coût temporel proportionnel à l'index de l'élément que l'on souhaite obtenir : il faut parcourir les éléments de la liste jusqu'à parvenir à celui souhaité, il n'existe pas de façon d'y accéder directement comme c'est le cas dans un tableau [19].

De la même façon, `List.mem` a un coût qui correspond à la position de l'élément recherché (s'il se trouve dans la liste), et donc une complexité dans le pire des cas linéaire ($O(n)$) vis-à-vis de la taille $n$ de la liste (si l'élément recherché se trouve en début de liste, la recherche sera plus efficace).

La concaténation de deux listes est un cas particulièrement intéressant. Prenons par exemple le cas de la concaténation des listes `[ 1; 2; 3 ]` et `[ 4; 5 ]` avec notre fonction `concat`. Si l'on décompose les appels, les choses se passent de la façon suivante :

```
concat [ 1; 2; 3 ] [ 4; 5 ]
1 :: ( concat [ 2; 3 ] [ 4; 5 ] )
1 :: ( 2 :: ( concat [ 3 ] [ 4; 5 ] ) )
1 :: ( 2 :: ( 3 :: concat ( [ ] [ 4; 5 ] ) ) )
1 :: ( 2 :: ( 3 :: [ 4; 5 ] ) )
1 :: ( 2 :: [ 3; 4; 5 ] )
1 :: [ 2; 3; 4; 5 ]
[1; 2; 3; 4; 5]
```

On utilise trois fois le motif de filtrage pour extraire les trois éléments de la liste utilisée comme premier argument, avant d'utiliser le second motif de notre fonction `Concat`, puis on utilise trois fois également l'opérateur `::` pour recoller chacun des éléments à la liste utilisée comme second argument. On effectue également quatre appels à `concat`, et autant de filtrages.

La complexité temporelle de cette fonction est donc linéaire vis-à-vis de la longueur de la liste *de gauche*. Ce qui n'est pas étonnant : la liste résultat réutilise les éléments de la liste de droite, mais on a eu besoin de créer autant de cellules qu'il y a d'éléments dans la liste de gauche.

19. Ou comme c'est le cas dans l'implémentation standard des listes Python : comme il ne s'agit pas de listes chaînées, `L[i]` a une complexité temporelle constante indépendante de `i`. Obtenir la longueur d'une liste Python a également un coût constant.

## 3.6 Retour sur des algorithmes classiques

### Somme des termes d'une liste

Puisque le langage OCaml tend à faire la part belle à la récursion, on sera souvent amenés à réfléchir au lien entre le résultat attendu sur une liste et sur sa queue. Supposons par exemple que l'on cherche à sommer les termes (entiers) d'une liste. Il est assez facile de voir que

$$\mathrm{sum}(\mathtt{h} :: \mathtt{t}) = \mathtt{h} + \mathrm{sum}(\mathtt{t})$$

Une fois cette relation identifiée, l'écriture de la fonction est assez immédiate, tant que l'on peut trouver une terminaison acceptable :

```
# let rec sum = function
    | []   -> 0
    | h::t -> h + somme t

val sum : int list -> int = <fun>
```

En dehors de l'appel récursif, la fonction est de complexité constante ($\Theta(1)$). Les appels récursifs seront au nombre de $n$ (sur des listes dont on retire le premier terme à chaque appel) donc la complexité est fort heureusement linéaire ($\Theta(n)$).

### Plus grand terme d'une liste

La démarche est la même lorsque l'on cherche à déterminer le plus grand élément d'une liste. La relation qui nous mets sur la voie de la récursion serait ici :

$$\mathrm{maximum}(\mathtt{h} :: \mathtt{t}) = \max(\mathtt{h}, \mathrm{maximum}(\mathtt{t}))$$

La terminaison est un peu plus délicate à écrire (une liste vide doit conduire à une erreur, ce qui force à avoir un cas supplémentaire pour une liste à un seul élément). On écrira donc par exemple :

```
# let rec maximum = function
    | []    -> failwith "Liste vide"
    | h::[] -> h
    | h::t  -> max h (maximum t);;

val maximum : 'a list -> 'a = <fun>
```

Comme précédemment, la fonction est linéaire ($\Theta(n)$) puisqu'en dehors de l'appel récursif toutes les opérations sont de complexité temporelle constante ($\Theta(1)$).

### Index du plus grand terme

Les choses sont un peu plus délicates lorsque le résultat sur la queue de la liste ne suffit pas pour conclure, par exemple si l'on cherche l'*index* de ce même maximum. On aura donc besoin d'écrire une fonction auxiliaire qui renvoie à la fois l'index du maximum et le maximum lui-même sous la forme d'un couple, pour le garder finalement que le premier élément. Ce qui donne par exemple (on n'oubliera pas dans une telle fonction que l'index dans la liste est l'index dans la queue incrémenté d'une unité!) :

```
# let index_of_maximum lst =
    let rec index_and_maximum = function
      | []    -> failwith "Liste vide"
      | h::[] -> 0, h
      | h::t  -> let i, m = index_and_maximum t in
                   if m>h then (i+1, m) else (0, h)
    in fst (index_and_maximum lst);;

val index_of_maximum : 'a list -> int = <fun>
```

Si le corps de la fonction `index_and_maximum` est plus complexe que précédemment, toutes les opérations qui y sont effectuées, hors appel récursif, restent de complexité constante, donc la fonction `index_and_maximum` dans son ensemble est de complexité linéaire ($\Theta(n)$), et de même pour `index_of_maximum`.

### Plus longue section croissante

Pour poursuivre dans nos revisites d'algorithmes, penchons-nous sur l'écriture en OCaml d'une fonction qui renverrait la longueur de la plus longue série d'éléments consécutifs dans une liste qui soit croissante (au sens large). Cette fois encore, on ne peut pas écrire directement une relation entre la plus longue série croissante sur la liste entière et celle sur la queue de cette liste. Toutefois, on peut écrire que la plus longue série croissante est

- soit intégralement dans la queue de la liste;
- soit commence avec le premier élément de la liste.

On peut donc écrire une fonction auxiliaire renvoyant *à la fois* la longueur de la série croissante débutant avec le premier terme et la longueur de la plus longue série croissante de la liste, sous la forme d'un couple. Cela permet en effet de s'en sortir :

- si le premier terme de la liste est inférieur ou égal au second, la longueur de la série croissante débutant sur le premier terme de la liste est celle débutant sur le premier terme de la queue incrémenté d'une unité;
- sinon, la série croissante débutant sur le premier terme est réduite à un seul élément

et dans les deux cas, la plus longue série croissante est le maximum entre la plus longue série croissante dans la queue de la liste et la longueur de la série croissante débutant sur le premier terme.

On ne gardera, à l'issue de l'algorithme, que le second terme renvoyé par cette fonction auxiliaire. Cela donne donc :

```
# let longest_incr_subsequence lst =
    let rec aux = function
      | [] -> 0, 0
      | h1::h2::t when h1<=h2
        -> let actu, maxi = aux (h2::t)
             in 1+actu, max (1+actu) maxi
      | h::t -> 1, max 1 (snd (aux t))
    in snd (aux lst);;

val longest_incr_subsequence : 'a list -> int = <fun>
```

Le corps de la fonction `aux` étant une fois de plus de complexité constante si l'on excepte l'appel récursif, la fonction `aux` est de complexité linéaire ($\Theta(n)$), de même que `longest_incr_subsequence`.

### Sommes cumulées

Parfois, le problème se pose dans l'autre sens. Supposons par exemple que l'on souhaite construire la liste des valeurs cumulées : `[1; 4; 2; 3]` devant donner par exemple `[1; 4; 6; 10]`. Le souci, c'est qu'en OCaml, les listes sont construites à partir de la droite! Le premier élément qui sera placé dans la liste sur notre exemple sera le 10, et pour ce faire, il faudra connaître la somme des trois premiers éléments.

On peut résoudre ce problème en introduisant un argument supplémentaire entier à la fonction, et en créant une fonction renvoyant la liste des sommes cumulées en partant, non de 0, mais de cet argument supplémentaire :

```
# let cumsum lst =
    let rec cumsum_offset offset = function
      | [] -> []
      | h::t -> let nh = h+offset in nh::cumsum_offset nh t
  in cumsum_offset 0 lst;;

val cumsum : int list -> int list = <fun>
```

Notons qu'il s'agit d'un excellente occasion, pour qui embrasse la programmation fonctionnelle, d'utiliser une application partielle : on aurait pu commencer la fonction en ne précisant pas l'argument (« `let cumsum =` ») et terminer en renvoyant un application partielle (« `in cumsum_offset 0` »).

La fonction `cumsum`, comme les précédentes, a une complexité linéaire ($\Theta(n)$).

#### Extraction de maximum et tri bulle

Rappelons que le tri « bulle » est un tri par sélection qui, typiquement, détermine le plus petit élément en parcourant les éléments de droite à gauche et en emportant le plus petit des éléments rencontrés, et le dépose en tête, puis recommence sur l'ensemble des éléments à l'exception de celui venant d'être déposé. Cette démarche s'adapte très bien à des listes OCaml. On commence par écrire une fonction qui extrait le plus grand élément, en renvoyant ce dernier et la liste des *autres* éléments [20] :

```
# let rec min_and_rest = function
    | []    -> failwith "Liste vide"
    | h::[] -> h, []
    | h::t  -> let m, r = min_and_rest t in
               if m<h then m, h::r
                      else h, m::r;;

val min_et_reste : 'a list -> 'a * 'a list = <fun>
```

Puis on peut utiliser la fonction précédente pour écrire notre tri :

```
# let rec bubblesort = function
    | []  -> []
    | lst -> let m, r = min_and_rest lst in m::bubblesort r;;

val bubblesort : 'a list -> 'a list = <fun>
```

La fonction `min_and_rest` a une complexité temporelle linéaire ($\Theta(n)$), on retrouve donc une complexité quadratique ($\Theta(n^2)$) pour le tri bulle.

#### Insertion dans une liste triée et tri insertion

Supposons à présent que l'on souhaite insérer un élément dans une liste dont les éléments sont triés par ordre croissant. Cela, grâce au filtrage et à la récursion, n'a rien de bien complexe :

```
# let rec insertion elem = function
    | [] -> [elem]
    | h::t when elem>h -> h::insertion elem t
    | lst -> elem::lst

val insertion : 'a -> 'a list -> 'a list = <fun>
```

20. Il est utile de regarder précisément comment la fonction extrait le maximum, et de se convaincre que les mouvements des éléments dans la liste correspondent bien à la description du tri « bulle », entre tous les algorithmes de tri par sélection.

La complexité temporelle de cette opération est linéaire dans le pire des cas ($O(n)$), elle peut toutefois être constante dans les cas favorables où l'élément inséré trouve sa place dans les premières itérations.

On peut ensuite écrire un tri par insertion, toujours de manière récursive, en disant que cela consiste, pour une liste non-vide, à trier (récursivement) sa queue, puis à insérer sa tête dans le résultat [21] :

```
# let rec insertionsort = function
    | [] -> []
    | h::t -> insertion h (insertionsort t);;

val insertionsort : 'a list -> 'a list = <fun>
```

La complexité du tri par insertion sur les listes OCaml reste quadratique ($O(n^2)$), avec une complexité linéaire dans le meilleur des cas si la liste est déjà triée par ordre croissant.

#### Partition et tri rapide

Partitionner une liste est un peu plus délicat, mais on peut imaginer une fonction récursive répartissant les éléments d'une liste entre deux listes selon qu'ils sont plus petits ou plus grands que le pivot. On pourra par exemple écrire [22] :

```
# let rec partition pivot = function
    | [] -> [], []
    | h::t -> let lst1, lst2 = partition pivot t
              in if h<pivot then h::lst1, lst2
                            else lst1, h::lst2;;

val partition : 'a -> 'a list -> 'a list * 'a list = <fun>
```

Le tri rapide s'écrit ensuite simplement :

```
# let rec quicksort = function
    | [] -> []
    | pivot::t -> let lst1, lst2 = partition pivot t
                  in quicksort lst1 @ pivot :: quicksort lst2;;

val quicksort : 'a list -> 'a list = <fun>
```

21. On notera que le déroulement du tri se fait dans le sens « opposé » à celui présenté sur les tableaux en langage C : c'est ici une partie à *droite*, de plus en plus longue, qui reste à tout instant triée, puisque l'on insère les éléments de la gauche vers la droite.

22. Attention, le pivot n'est pas ici pris dans la liste, mais fourni comme un argument à la fonction !

La fonction quicksort a les mêmes propriétés en terme de complexité que précédemment : dans le pire des cas, la complexité est quadratique ($O(n^2)$), mais ele peut devenir quasi-linéaire ($O(n \log n)$) si le pivot est de bonne qualité. On peut imaginer les mêmes améliorations pour essayer d'augmenter les chances, voire garantir que l'on soit dans le cas quasi-linéaire.

## 3.7 Fonctionnelles agissant sur les listes

Outre le filtrage, un certain nombre de mécanismes permettent d'écrire plus simplement des opérations sur des listes, en appliquant, de diverses façons, une fonction donnée à tous les éléments d'une liste.

Il convient de s'en servir *avec parcimonie*, car si ces écritures peuvent régulièrement simplifier les expressions, elles peuvent tout aussi bien les rendre illisibles, surtout sans un mot d'explication !

**La fonction `List.iter`**

Le mécanisme le plus simple est fourni par la fonction `List.iter`, qui prend en argument une fonction et une liste, la fonction devant accepter des éléments de même type que ceux présents dans la liste. Cette fonction est alors appliquée à tous les éléments de la liste, de gauche à droite :

```
# let my_list = [ 1; 2; 3; 4; 5 ];;
val my_list : int list = [1; 2; 3; 4; 5]

# List.iter print_int my_list;;
12345- : unit = ()
```

Puisqu'une fonction ne peut renvoyer qu'un seul élément, la fonction `List.iter` prend en paramètre des fonctions renvoyant un type `unit` (comme `print_int`) :

```
# List.iter;;
- : ('a -> unit) -> 'a list -> unit = <fun>
```

Bien évidemment, la fonction `List.iter` n'a d'intérêt que si la fonction passée en argument a un effet de bord sur l'environnement, comme par exemple un affichage !

Nous le verrons sur des exemples concrets, `List.iter` est le mécanisme naturel, en OCaml, pour itérer sur les éléments d'une liste, mais sa nature fonctionnelle fait qu'il peut être plus difficile de trouver une présentation et une indentation du code pour que cette itération apparaisse clairement comme telle, ce qui est indispensable si on veut être compris de son lecteur.

**La fonction `List.map`**

Si la fonction à appliquer sur les éléments de la liste renvoie un résultat, on peut vouloir récupérer les résultats de l'application de la fonction à tous les éléments de la liste. Pour ce faire, on dispose de la fonction `List.map` qui attend une fonction et une liste :

```
# List.map;;
- : ('a -> 'b) -> 'a list -> 'b list = <fun>
```

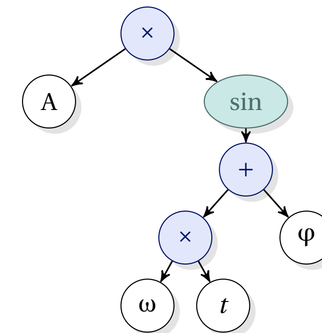
À la liste $[a_1, a_2, \ldots, a_n]$, on associe donc la liste $[f(a_1), f(a_2), \ldots, f(a_n)]$. On remarquera que le type des éléments de la liste renvoyée la la fonction `List.map` peut fort bien être différent du type des éléments de la liste fournie en paramètre.

Par exemple, on peut convertir en flottants et élever au carré une liste d'entiers :

```
# let my_list = [ 1; 2; 3; 4; 5 ];;
val my_list : int list = [1; 2; 3; 4; 5]

# let foo n = float_of_int n ** 2.0;;
val foo : int -> float = <fun>

# List.map foo my_list;;
- : float list = [1.; 4.; 9.; 16.; 25.]
```

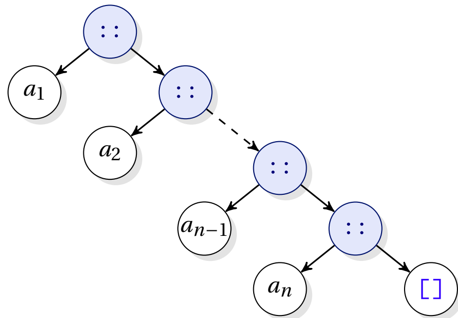
Afin de mieux comprendre le fonctionnement de `List.map` et des autres fonctionnelles que nous allons étudier, il peut être intéressant de visualiser les objets manipulés sous une forme arborescente.

Considérons par exemple l'expression $A\sin(\omega t + \varphi)$. On peut la représenter de la sorte :

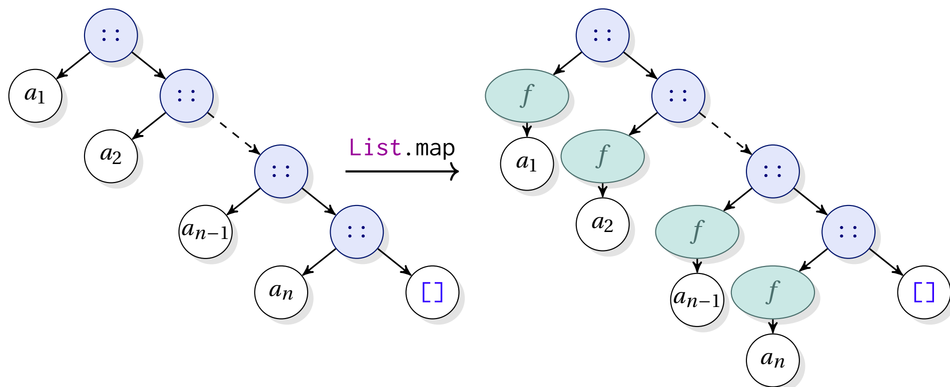



Cette écriture identifie clairement et sans ambiguïté les opérations permettant de construire le résultat de l'expression. Pour l'évaluation, on part des éléments les plus bas et on remonte. Ainsi, pour calculer $A\sin(\omega t + \varphi)$, on calcule tout d'abord le produit de $\omega$ avec $t$, puis on ajoute $\varphi$ ; on prend le sinus du résultat, et on multiplie enfin le tout par A.

De la même façon, une liste est simplement le résultat d'utilisations successives de `::` sur une liste vide `[]`, insérant un par un les éléments par la gauche. Ainsi, une liste $[a_1, a_2, ...a_n]$ peut être représentée par :




L'utilisation de `List.map` correspond donc à la transformation ci-dessous, dans laquelle on a inséré la fonction `f` entre chacun des éléments de la liste et les « conse », suivie de l'évaluation de la structure arborescente :




Tout se passe donc comme si la fonction `List.map` était définie de la sorte :

```
# let rec map f = function
    | []   -> []
    | h::t -> (f h)::(map f t);;

val map : ('a -> 'b) -> 'a list -> 'b list = <fun>
```

Profitons de l'occasion pour souligner un point important du langage OCaml : il ne spécifie pas dans quel ordre doivent être effectuées les opérations lors de l'évaluation d'une expression (exception faite des expressions booléennes qui, paresseuses, imposent l'évaluation du membre de gauche avant celui du membre de droite pour `&&` et `||`).

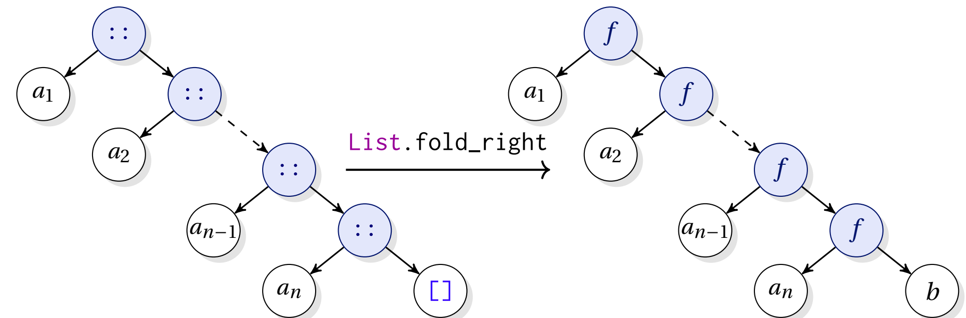
Ainsi, dans l'expression précédente, on ne peut pas savoir si « `f t` » sera évalué avant « `map f q` » ou inversement. L'ordre d'évaluation des « `f aux a_i` » est donc indéterminé, et il en est de même pour `List.map`, contrairement à ce qui se passe avec `List.iter`. Les programmes ne *devront donc pas* dépendre de cet ordre d'évaluation.

La complexité temporelle de `List.map` dépend directement de celle de la fonction agissant sur les arguments : celle-ci est appliquée autant de fois qu'il y a d'éléments dans la liste. Le reste des opérations (notamment la construction de la liste résultat) a un coût temporel, dans le pire des cas, du même ordre que celui des appels à la fonction, et peut donc être ignoré.

#### La fonction `List.fold_right`

La fonction `List.fold_right`, quant à elle, prend en argument une fonction $f$ attendant deux arguments de types différents, une liste $[a_1, a_2, ...a_n]$ d'éléments du premier type et un élément $b$ du second, et renvoie $f\left(a_1, f\left(a2, f\left(...\left(a_n - 1, f(a_n, b)\right)...\right)\right)\right)$.

Elle effectue donc la transformation suivante :




Sa signature est la suivante (on remarquera au passage que les $a_i$ et l'élément $b$ ne sont pas nécessairement de même type) :

```
# List.fold_right;;
- : ('a -> 'b -> 'b) -> 'a list -> 'b -> 'b = <fun>
```

Cette fonction pourrait être implémentée en Caml de la façon suivante :

```
# let rec fold_right f lst b =
    match lst with
      | []   -> b
      | h::t -> f h (fold_right f t b);;

val fold_right : ('a -> 'b -> 'b) -> 'a list -> 'b -> 'b = <fun>
```

Cette fois, il n'y a pas d'ambiguité sur l'ordre dans lequel les appels à `f` seront effectués : l'évaluation de « `fold_right f q b` » (et donc les appels à `f` qu'il contient)

précède nécessairement l'appel à f qui apparaît dans la fonction. Il en est de même pour la fonction List.fold_right : comme le laisse suggérer l'expression parenthésée $f(a_1, f(a2, f(\ldots(a_n - 1, f(a_n, b))\ldots)))$, les calculs se font en partant de $a_n$ et en remontant vers $a_1$.

Comme pour List.map, la complexité temporelle dépendra des $n$ appels à la fonction f, les autres opérations, de coût global linéaire ($\Theta(n)$), représentant un coût au plus équivalent à celui des appels à f.

La fonction List.fold_right, *utilisée à bon escient*, permet de simplifier l'écriture de nombreux algorithmes opérant sur des listes. On évite ainsi la nécessité d'écrire explicitement une récursion (ou, on le verra, une boucle).

Le tout est de parvenir à déterminer quelle fonction $f$ utiliser à la place des « :: » et quelle valeur choisir pour l'élément b. Par exemple, obtenir la somme des éléments d'une liste d'entiers peut se faire en plaçant des sommes à la place des « :: » et un 0 en bas à droite :

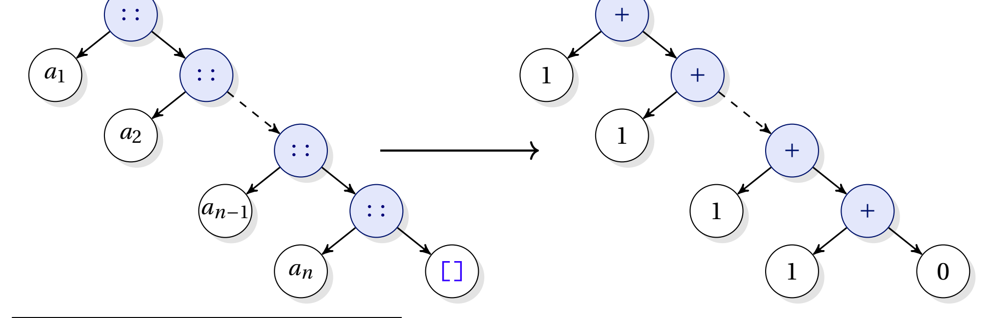
```
# let sum a b = a+b
    in List.fold_right sum [ 1; 2; 3; 4; 5 ] 0;;
- : int = 15
```

De même, pour déterminer la longueur d'une liste, on peut écrire :

```
# let count a b = b+1
    in List.fold_right count [ 1; 2; 3; 4; 5 ] 0;;
- : int = 5
```

Pourquoi cette fonction « count a b = b+1 »? Simplement parce que, pour chaque élément $a_i$ de la liste, sa longueur (initialisée à 0 pour []) augmente de 1, et cela quelle que soit la valeur de $a_i$.

En d'autres termes, le calcul effectué ici avec count pourrait être schématisé [23], par la transformation représentée ci-dessous :




23. Même si l'évaluation ne se fait pas rigoureusement comme le suggère l'arbre de droite.

On peut s'en servir pour définir des fonctions, par exemple sum_list :

```
# let sum_list lst =
    List.fold_right (fun a b -> a+b) lst 0;;

val sum_list : int list -> int = <fun>
```

Signalons que la fonction « fun a b -> a+b » peut être avantageusement remplacée par « (+) » :

```
# let sum_list lst =
    List.fold_right (+) lst 0;;

val sum_list : int list -> int = <fun>
```

Pour chaque opérateur binaire OCaml, on dispose d'une fonction dont le nom est l'opérateur entre parenthèses, prenant deux arguments et renvoyant l'application de l'opérateur à ces deux arguments.

On peut également écrire une fonction length prenant en argument une liste et renvoyant sa longueur :

```
# let length lst =
    List.fold_right (fun a b -> b+1) lst 0;;

val length : 'a list -> int = <fun>
```

Si l'on veut obtenir le plus grand élément d'une liste, le choix du troisième argument de List.fold_right est un peu plus délicat.

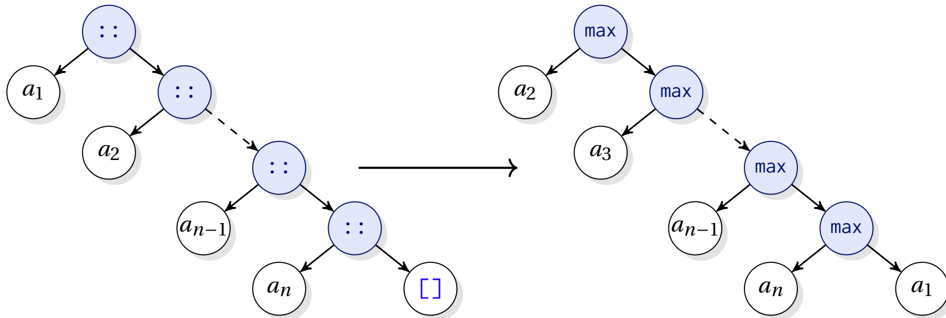
Habituellement, en programmation impérative, pour déterminer le plus grand élément d'une liste, il est d'usage de partir d'un élément quelconque de la liste. On peut de la même façon prendre ici la tête de la liste comme « élément b » (et n'appliquer List.fold_right qu'à la queue de la liste puisque le premier élément a déjà été pris en compte).

On définira donc max_of_list ainsi :

```
# let max_of_list lst =
    List.fold_right max (List.tl lst) (List.hd lst);;

val max_of_list : 'a list -> 'a = <fun>
```

Débuter le « repliement » par un élément de la liste permettra par ailleurs de conserver le caractère polymorphe de la fonction max, et il est possible d'obtenir le plus grand élément de listes contenant n'importe quel type d'éléments (entiers, flottants, caractères, chaînes de caractères, etc.).

La transformation que l'on a effectué est donc :




Signalons enfin qu'en utilisant pour type `'b` une liste, on peut parfaitement utiliser la fonction `List.fold_right` pour construire une liste. Il est ainsi par exemple possible de définir une fonction équivalente à `List.map` en écrivant :

```
# let rec map f lst =
    List.fold_right (fun a b -> (f a)::b) lst [];;

val map : ('a -> 'b) -> 'a list -> 'b list = <fun>
```

**La fonction `List.fold_left`**

La fonction `List.fold_right` a un pendant, `List.fold_left`, qui réalise une transformation très similaire, mais associe à la liste $[a_1, a_2, ... a_n]$ et un élément $b$ le résultat de $f\left(f\left(...\left(f(b, a_1),\right), a_2, ...\right), a_n\right)$. Elle est équivalente à la fonction suivante :

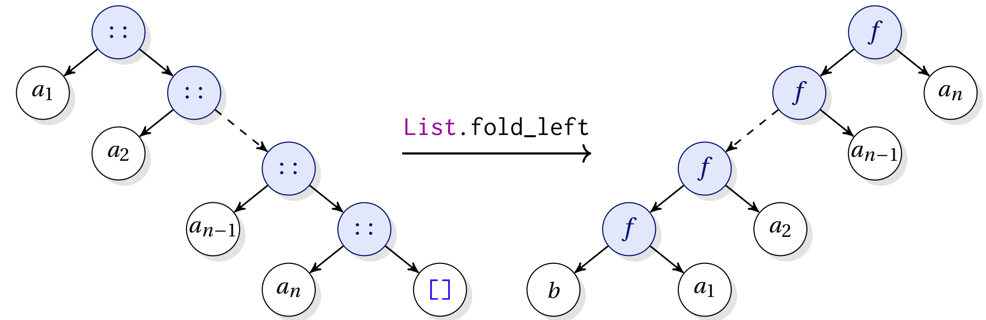
```
# let rec fold_left f b = function
    | []   -> b
    | h::t -> fold_left f (f b h) t;;

val plie_gauche : ('a -> 'b -> 'a) -> 'a -> 'b list -> 'a = <fun>
```

Attention, l'ordre des paramètres n'est pas le même que `List.fold_right`, la liste vient cette fois en troisième et dernier paramètre. De la même façon, la fonction passée en paramètre doit prendre en *second* argument les éléments $a_i$ de la liste (et en premier argument, des éléments du type de b).

On prendra également garde au fait que, du fait de cette inversion, dans la signature, les $a_i$ sont donc de type `'b` et $b$ est de type `'a` :

```
# List.fold_left;;
- : ('a -> 'b -> 'a) -> 'a -> 'b list -> 'a = <fun>
```

Cette fonction effectue donc la transformation suivante [24] :




Le fait que la liste se trouve en dernier paramètre permet de définir encore plus simplement une fonction sommant les éléments au travers d'une application partielle :

```
# let sum_list =
    List.fold_left (fun b a -> b+a) 0;;

val sum_list : int list -> int = <fun>
```

Ou bien, de façon équivalente :

```
# let sum_list =
    List.fold_left (+) 0;;

val sum_list : int list -> int = <fun>
```

Ou calculant la longueur de la liste [25] :

```
# let length =
    List.fold_left (fun b a -> b+1) 0;;

val length : '_weak1 list -> int = <fun>
```

Nous verrons un peu plus tard que la fonction `List.fold_left` est un peu plus performante que `List.fold_right`.

24. Le « renversement » de la structure arborescente fait que, théoriquement, `List.fold_left` n'est pas, *stricto-sensu*, un repliement.

25. On remarquera une bizarrerie dans la signature (parfois, on verra également `'_a` en lieu et place de `'_weak1`, ce qui est équivalent), sur laquelle nous reviendrons quelque peu ultérieurement : la fonction obtenue n'est pas totalement polymorphe, elle accepte des listes contenant un type quelconque, mais la première utilisation « fixera » ce type. Par exemple, après avoir calculé `longueur [ 1; 2 ]`, longueur sera de type `int list -> int`. Il n'est pas possible de définir une fonction polymorphe à partir d'une application partielle. Pour éviter ce problème, on fera explicitement apparaître le troisième argument dans la définition de la fonction. Les raisons de cette subtilité dépassent grandement le cadre de ce cours.

Les deux fonctions existent car elles ne répondent pas tout à fait aux mêmes besoins. Il est un peu délicat d'écrire une fonction `List.map` à partir de `List.fold_left` alors que la chose était facile avec `List.fold_right`. La raison en est que l'opérateur `::` ajoute les éléments à gauche, or `List.fold_left` tend à retourner la liste.

`List.fold_left` permet en revanche aisément de retourner une liste :

```
# let reverse =
    List.fold_left (fun lst e -> e::lst) [];;

val reverse : '_a list -> '_a list = <fun>
```

Ce retournement serait plus difficile à obtenir avec `List.fold_right`.

#### Un mot d'avertissement

Les fonctions `List.fold_left` et `List.fold_right` peuvent, dans certaines situations précises, être parfaitement naturelles : si par exemple on dispose d'une fonction « add » ajoutant un élément à un ensemble existant, et que l'on souhaite construire l'ensemble correspondant à une liste d'éléments, il est défendable d'utiliser `List.fold_left` pour, en partant d'un ensemble vide, lui appliquer la fonction `add` avec chacun des éléments de la liste.

Sur ce principe, il serait envisageable d'implémenter le tri par insertion (qui insère un à un les éléments d'une liste dans une liste initialement vide) en écrivant [26] :

```
# let insertion_sort lst =
    let rec insertion lst elem = match lst with
      | [] -> [elem]
      | h::t when elem>h -> h::insertion t elem
      | lst -> elem::lst

    in List.fold_left insertion [] lst;;

val insertion_sort : 'a list -> 'a list = <fun>
```

Mais il ne faut vraiment pas abuser de ces constructions pour des situations qui seraient moins naturelles. Parmi les exemples précédents, `sum_list` se défend probablement, `max_of_list` passe sans doute aussi, mais `length` commence à interroger : est-ce la solution la plus simple à comprendre ?

Quoi qu'il arrive, si ce n'est pas une transformation parfaitement évidente, il est indispensable d'expliquer en détail et avec précision le principe de la réduction.

26. Attention à l'inversion nécessaire dans l'ordre des arguments dans la fonction `insertion` !

#### À propos des langages fonctionnels

Les fonctions `List.map`, `List.fold_left` et `List.fold_right` sont présentes dans la quasi-totalité des langages fonctionnels. Parfois, seul un équivalent de `List.fold_left` est disponible (appelé `reduce` en Clojure, Ruby, D, `fold` en F#, etc.) et on se sert d'un renversement de liste pour obtenir l'alternative.

On les retrouve d'ailleurs aussi en Python, qui, bien que n'étant pas à l'origine un langage fonctionnel, dispose néanmoins d'une fonction `map` (quoi qu'elle fasse double emploi avec le mécanisme de compréhension de liste, plus puissant) et d'une fonction `functools.reduce` qui se comporte comme `List.fold_left`.

## 3.8 Listes et prédicats booléens

Terminons-en avec les listes en présentant quelques fonctions qui peuvent se révéler fort utiles pour écrire de nombreux algorithmes. Elles ne figurent pas au programme, il est vivement recommandé de savoir les réécrire au besoin. Ce sera l'occasion pour nous également de signaler une limite à l'utilisation des outils précédents.

Un *prédicat* est une propriété qui peut être vraie ou fausse pour un élément donné. En d'autres termes, il s'agit d'une fonction `'a -> bool` prenant en argument un élément et renvoyant un booléen. Par exemple, le prédicat « `is_even` » se traduit simplement par la fonction « `function n -> n mod 2 = 0` ».

Il peut très souvent s'avérer utile de savoir si l'ensemble des éléments d'une liste vérifient un prédicat. Pour ce faire, on dispose d'une fonction `List.for_all` de signature `('a -> bool) -> 'a list -> bool`, de complexité linéaire ($O(n)$), indiquant par le booléen qu'elle renvoie si c'est le cas. Elle est équivalente à la fonction suivante :

```
# let rec for_all pred = function
    | [] -> true
    | h::t -> pred h && for_all pred t;;

val for_all : ('a -> bool) -> 'a list -> bool = <fun>
```

Il s'agit en fait d'appliquer la fonction `pred` à tous les éléments de la liste, puis d'effectuer un « et logique » sur tous les résultats. Avec les outils que l'on vient de voir, on pourrait envisager d'écrire, dans un style très fonctionnel :

```
# let for_all pred lst =
    List.fold_left (&&) true (List.map pred lst);;

val for_all : ('a -> bool) -> 'a list -> bool = <fun>
```

Cela marche parfaitement, mais il faut néanmoins signaler une différence non négligeable : dans cette seconde version, `pred` est appliquée à *tous* les éléments, et le « et logique » ne sera pas paresseux. La fonction est alors de complexité temporelle linéaire dans *tous* les cas ($\Theta(n)$), tandis que la version récursive elle peut être de complexité constante dans un cas favorable, par exemple si le premier élément ne satisfait pas le prédicat.

En pendant à la fonction `List.for_all`, on dispose d'une fonction `List.exists`, de même signature et de complexité $O(n)$, renvoyant un booléen indiquant si *au moins un* des éléments de la liste satisfait le prédicat. Elle est équivalente à

```
# let rec exists pred = function
    | [] -> false
    | h::t -> pred h || exists pred t;;

val exists : ('a -> bool) -> 'a list -> bool = <fun>
```

On pourrait écrire, au prix d'une complexité devenant $\Theta(n)$, la fonction de la sorte :

```
# let exists pred lst =
    List.fold_left (||) false (List.map pred lst);;

val exists : ('a -> bool) -> 'a list -> bool = <fun>
```

Une fonction `List.find`, de signature `('a -> bool) -> 'a list -> 'a`, de complexité linéaire ($O(n)$), permet d'obtenir le premier élément de la liste en argument qui satisfasse le prédicat (et une erreur si elle n'en trouve pas), ce qui pourrait s'écrire :

```
# let rec find pred = function
    | [] -> failwith "Non trouvé"
    | h::_ when pred t -> t
    | _::t -> find pred q;;

val find : ('a -> bool) -> 'a list -> 'a = <fun>
```

Enfin, `List.filter` de signature `('a -> bool) -> 'a list -> 'a list`, de complexité linéaire ($\Theta(n)$), renvoie une liste contenant tous les éléments de la liste en argument qui vérifient le prédicat, en préservant leur ordre. Elle pourrait s'écrire :

```
# let rec filter pred = function
    | [] -> []
    | h::t when pred h -> h::filter pred t
    | _::t -> filter pred t;;

val filter : ('a -> bool) -> 'a list -> 'a list = <fun>
```

# 4 Créer des types complexes en C

## 4.1 Déclaration de types

Le langage C permet également de définir ses propres types, et de les nommer. Pour ce faire, on utilise le mot-clé « `typedef` » suivi d'une description du type et du nom qu'on souhaite lui donner.

L'usage le plus élémentaire de cette construction est la création d'« *alias* » de types, c'est-à-dire attribuer un nouveau nom à un type existant. Par exemple, en écrivant

```
typedef int age;
```

on définit un nouveau type appelé « age » mais qui, derrière ce nom, cache des entiers. Précisons qu'il n'y a pas de contrainte particulière concernant les noms de nouveaux types si ce n'est qu'ils doivent suivre les mêmes règles que les noms de variables.

La création d'alias de types permet d'être parfois un peu plus clair pour les arguments des fonctions (ou les valeurs renvoyées), comme sur cet exemple :

```
bool can_vote(age a) {
    return a >= 18;
}
```

Cela étant dit, même si le nom du type a changé, on continue ici à manipuler des objets de type « `int` ». Une variable de type « age » sera acceptée comme argument de toute fonction attendant un entier (et ne subira aucune conversion particulière). Il s'agit essentiellement de décoration facilitant la lecture du programme. Le typage faible du langage ne conduira pas à une sécurité accrue du programme.

Nous avons en fait déjà croisé de tels alias : nous avons par exemple évoqué le fait que le type booléen en C99 se nomme normalement[27] « `_Bool` ». Mais le fichier d'entête « `stdbool.h` » inclue, entre autres choses, la définition

```
typedef _Bool bool;
```

pour que l'on puisse utiliser le plus naturel « `bool` ».

## 4.2 Enregistrements

Pour aller plus loin, il nous faut des mécanismes pour créer des types plus élaborés comme en OCaml. Il est possible, en C, de créer des types « enregistrements » grâce au

27. Nous verrons un peu plus tard que ce type `_Bool` cache en fait lui-même un type entier !

mot-clé « **struct** ». Si l'on reprend l'exemple du début de ce chapitre, on peut définir un type décrivant un déplacement en écrivant :

```c
struct displacement {
    double n_s;
    double e_w;
};
```

Le type ainsi créé a pour nom « `struct displacement` ». Pour définir un objet de ce type, on écrira, et initialiser les différents champs, on pourra écrire par exemple :

```c
struct displacement d;
d.n_s = 10.2;
d.e_w = 4.5;
```

On remarque immédiatement la similarité avec le fonctionnement en OCaml : pour accéder au champs d'une variable de type enregistrement, on fait suivre le nom de la variable d'un point et du nom du champ. En revanche, il y a aussi une différence majeure avec ce que l'on a vu en OCaml : il est possible de modifier le contenu des champs, ce qui n'était pas possible avec les définitions que l'on a vu en OCaml (les valeurs des champs étant définies lors de la création de l'objet, et ne sont plus modifiables ensuite).

Il existe une écriture qui permet, de façon similaire aux tableaux, d'initialiser le contenu de la structure juste après sa déclaration (histoire d'éviter tout risque d'utiliser des variables qui n'ont pas été initialisées), en écrivant :

```c
struct displacement d = { .n_s = 10.2, .e_w = 4.5 };
```

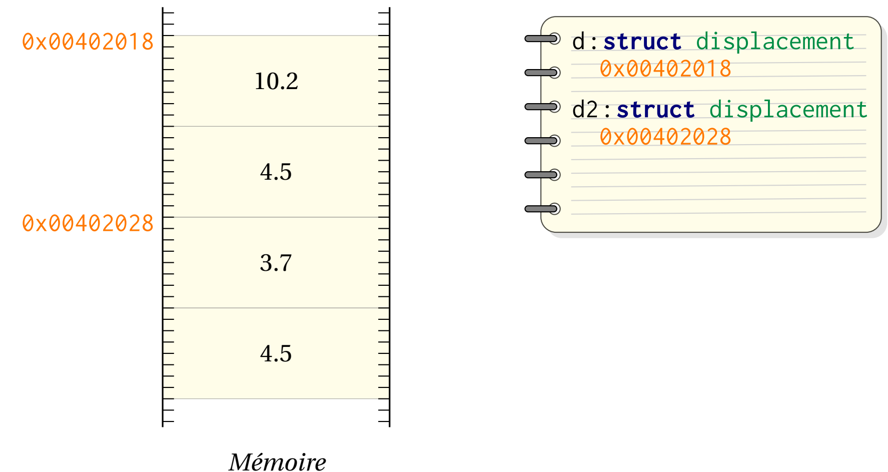
Comme pour les tableaux, cette initialisation utilisant des accolades n'est acceptable que lors d'une *initialisation* de la variable suivant immédiatement sa déclaration. En temps normal, pour changer les valeurs, il faudra procéder champ par champ comme précédemment.

Du point de vue de la mémoire, le compilateur réserve une taille suffisante pour mémoriser l'ensemble des données [28]. Les enregistrements se comportent similairement aux types élémentaires du langage (`int`, `double`, `bool`...). En particulier, lors d'une affectation, les données sont intégralement *copiées*. Par exemple, dans le morceau de code suivant,

```c
struct displacement d = { .n_s = 10.2, .e_w = 4.5 };

struct displacement d2 = d;   // <- les données sont copiées
d2.n_s = 3.7;                 // <- n'affecte pas dep !
```

28. Parfois un peu plus pour des raisons d'alignement des adresses.

la déclaration de la variable « d2 » réserve un espace mémoire suffisant pour ranger deux flottants, puis l'affectation « `d2 = d` » a pour effet de *recopier* le contenu de la structure contenue dans la variable dep dans celle contenue dans la variable d2. Par conséquent, l'affectation pour le champ n_s de d2 n'a aucun effet sur le champ de même nom de la variable d. En mémoire, les choses sont donc organisées de la façon suivante :



*Mémoire*

Lorsque l'on utilise une structure comme paramètre d'un appel de fonction, là encore, les données sont *copiées*. Il s'agit, comme pour les types élémentaires, d'un passage *par valeur*. La structure à laquelle on peut accéder depuis le corps de la fonction est une variable locale (contenant une copie des données de la structure fournie lors de l'appel), aussi modifier ses champs n'a aucun effet sur la structure utilisée en argument. Il est également parfaitement valable de renvoyer une variable locale contenant une structure comme résultat, puisque cette structure sera copiée lors du retour.

On peut même se servir de structures pour passer un tableau *par valeur* à une fonction : inclus dans une structure, un tableau statique sera bien copié ! Cette curiosité n'est pas souvent utilisée, et on évitera de stocker des tableaux de grande taille dans des structures, car les copies risquent d'être gourmandes en temps processeur.

S'il est possible de copier le contenu d'une variable contenant une structure dans une autre variable par une affectation, précisons qu'il est en revanche impossible en C de tester l'égalité de deux variables de type structure avec « `==` », comme on peut le faire en OCaml (comparaison avec, rappelons-le, `=`).

On peut tenter d'expliquer cela par le fait qu'il est nécessaire de procéder récursivement : on considérera généralement que les deux variables sont égales si et seulement si le contenu de leurs champs, deux à deux, sont égaux (c'est ce que fait OCaml). C'est cependant une tâche qui n'est pas triviale pour le compilateur, et une difficulté supplémentaire est qu'il n'est pas rare en C d'utiliser des structures où tous les champs ne sont

pas nécessairement toujours remplis [29]. Plutôt que de proposer une comparaison imparfaite, les développeurs du langage ont préféré laisser aux programmeurs la charge d'écrire eux-mêmes des fonctions de comparaison pour leurs propres structures.

Ainsi, dans notre cas, il faudra écrire (plutôt que `d1 == d2` qui ne fonctionnera pas) :

```
bool displ_equals(struct displacement d1, struct displacement d2) {
    return d1.n_s == d2.n_s && d1.e_w == d2.e_w;
}
```

La syntaxe « `struct displacement` » pour désigner le type peut sembler malcommode, aussi peut-on se servir de `typedef` pour faire disparaître le `struct` en créant un alias, en écrivant par exemple [30] :

```
typedef struct displacement displacement;
```

Après cette déclaration, « `displacement` » sera équivalent à « `struct displacement` ». Les deux noms (le nom suivant « `struct` » et l'alias) n'ont pas à être les mêmes, mais ce n'est pas inhabituel d'utiliser le même nom. Il n'y a en effet aucun risque de confusion, puisque la dénomination d'un des deux types commence sans ambiguïté par « `struct` ».

En fait, on gagne fréquemment du temps en déclarant l'alias et la définition de la structure en une seule opération [31] :

```
typedef struct displacement { double n_s; double e_w; } displacement;
```

Dans ce cas, il est fréquent de ne pas nommer la structure, et d'écrire simplement [32] :

```
typedef struct { double n_s; double e_w; } displacement;
```

En théorie, on peut même se passer de déclarer explicitement un type. Si on n'a besoin que d'une unique variable devant contenir quelques champs, on peut directement la déclarer de la sorte [33] :

```
struct { double n_s; double e_w; } d;
```

La variable `d` ainsi déclarée est donc une structure à deux champs, et on peut librement utiliser « `d.n_s` » et « `d.e_w` ».

29. Ou, éventuellement, peuvent avoir une fonction qui change au cours du temps, comme nous le verrons brièvement avec `union` un peu plus loin.
30. Précisons que pour pouvoir écrire ce `typedef`, le type `struct deplacement` n'a pas besoin d'avoir déjà été défini, il peut donc se situer librement avant ou après la définition du type `struct displacement`.
31. Cela n'est pas forcément évident, mais les deux types « `struct displacement` » et « `displacement` » sont tous deux disponibles après une telle déclaration.
32. Cela évite notamment d'avoir deux noms différents pour le type nouvellement déclaré.
33. On ne pourra en revanche pas transmettre cette variable à une quelconque fonction, puisqu'il ne pourra pas vérifier s'il y a compatibilité de types (sauf éventuellement en rusant avec des pointeurs).

## 4.3 Énumérations, unions

Comme OCaml, le langage C offre la possibilité de créer des types énumérés, et des équivalents aux types « unions ». Seule la connaissance des types « enregistrement » est exigible, donc nous n'évoquerons que brièvement les possibilités offertes par le langage.

Pour créer un type énuméré, on utilise le mot-clé « `enum` ». On peut par exemple définir un type décrivant une direction avec :

```
enum direction { north, south, east, west };
```

On définit ainsi un type appelé « `enum direction` » pouvant en principe prendre quatre valeurs possibles : `north`, `south`, `east` et `west`, et quatre noms correspondant à ces mêmes quatre constantes. On s'en sert ensuite comme on le ferait en OCaml :

```
enum direction dir;
dir = south;
```

Les noms correspondant aux valeurs possibles des variables du type énuméré se comportent comme des constantes, et il est notamment possible de tester l'égalité d'une variable avec ces noms, en écrivant par exemple « `if (dir == north) ...` ».

Comme précédemment, on peut utiliser un `typedef` pour obtenir un type cachant le « `enum` », en écrivant :

```
typedef enum direction direction;
```

et, possiblement, combiner l'alias et la définition en une même instruction :

```
typedef enum direction { north, south, east, west } direction;
```

ou bien encore, en déclarant une énumération anonyme :

```
typedef enum { north, south, east, west } direction;
```

Enfin, on peut même directement déclarer une variable sans explicitement créer un type :

```
enum { north, south, east, west } dir;
```

mais attention : si on a bien défini ici une variable de type énuméré « `dir` », pouvant prendre pour valeurs `north`, `south`, `east` ou `west`, on a en même temps défini ces quatres constantes, donc les noms ne peuvent pas réapparaître dans une autre déclaration du même bloc (en particulier, on ne peut pas définir une seconde variable de la même façon).

Pour définir un type « union », c'est un peu plus complexe. Supposons par exemple que l'on souhaite un équivalent C à la déclaration OCaml suivante :

```
type mytype = I of int | F of float
```

Il nous faut donc mémoriser à la fois si l'on est dans un cas ou dans l'autre, et des données supplémentaires. Comme il y a plusieurs choses à mémoriser, nous pouvons utiliser une structure et écrire :

```
typedef struct {
    enum { integer, floatingpoint } type;
    int i;
    double f;
} mytype;
```

Dans le programme, on utilisera donc le champ `.type`[34] pour mémoriser si l'on est dans un cas ou dans l'autre, et les données pourront être mémorisées dans le champ `.i` (s'il s'agit d'un entier) ou `.f` (s'il s'agit d'un flottant).

Le souci, avec cette approche, c'est que le langage réserve de la place en mémoire pour les *deux* champs, alors que l'on ne fera jamais en principe usage de ces deux champs simultanément. C'est du gaspillage. On dispose donc d'un moyen supplémentaire pour créer nos propres types en C, la construction appelée « **`union`** ». On écrira donc :

```
typedef struct {
    enum { integer, flotingpoint } type;
    union {
        int i;
        double f;
    } data;
} mytype;
```

Les différents champs dans une **`union`** sont tous accessibles comme on le ferait pour une structure, mais ils *partagent* la même zone mémoire (dont la taille est naturellement choisie pour pouvoir contenir, au minimum, le plus grand des éléments). Il n'y a donc plus de gaspillage !

Pour accéder au champ entier, on utilisera donc « `.data.i` », et pour le champ flottant « `.data.f` ». Compte tenu du fait que les deux utilisent la même zone mémoire, on ne devrait généralement lire que le dernier champ ayant été écrit, car affecter une valeur à l'un des champs a évidemment des effets sur l'autre !

34. Le champ n'a pas besoin de s'appeler « type », il s'agit d'un champ parfaitement normal.

# 5 Les listes en langage C

## 5.1 Création d'un type récursif

Comme pour nos listes chaînées OCaml, pour créer une liste chaînée d'entiers en C, nous allons définir un type récursif correspondant à une « cellule » de la liste chaînée, contenant une valeur entière et un moyen d'accéder à la suite.

En C, ce moyen d'accéder à la suite consiste en un pointeur : chaque cellule mémorisera l'*adresse* en mémoire de la cellule la suivant dans la liste chaînée. La définition de la structure devra donc être récursive.

Si la structure est nommée, cela n'est aucunement un problème. On peut écrire :

```
struct int_list {
    int value;
    struct int_list* next;
};
```

Cette définition est parfaitement suffisante : lorsque l'on veut un objet de ce type, on utilisera comme nom de type « **`struct`** `int_list` », comme dans la déclaration ci-dessous :

```
struct int_list mylist;
```

Mais pour alléger les choses et rendre le programme plus aisé à lire, on peut créer un alias en écrivant[35] :

```
typedef struct int_list int_list;
```

Reste maintenant à préciser quelques détails qui nous permettront de manipuler efficacement des listes chaînées en C.

## 5.2 Identifier l'extrémité droite de la liste

En OCaml, lorsque nous avons créé notre propre liste chaînée, nous avons utilisé, pour le `suivant` du dernier élément, un objet particulier que vous avions appelé « `Nil` ».

Cette fois, `next` se trouve être un pointeur. Plusieurs solutions sont possibles pour signifier que la structure correspond au dernier élément de la liste.

- Utiliser pour `next` la valeur `NULL`. C'est probablement la solution la plus couramment utilisée : il est en effet facile de tester `next` pour savoir si la structure est la dernière de la liste ou non.

35. Le **`typedef`** et la définition de la structure récursive peuvent éventuellement, comme précédemment, être regroupés en une seule instruction.

- Utiliser pour `next` l'adresse de la structure elle-même (comme si l'on avait terminé la liste par une boucle). Il reste relativement facile de tester si la structure est la dernière de la liste, et certains apprécient cette solution, notamment parce qu'elle limite les risques de déréférencer le pointeur `NULL` en allant un cran trop loin dans la liste[36] [37].
- Créer un objet spécifique, unique, (parfois qualifié de *sentinelle*) ayant le même type que les cellules constituant de la liste, dont le contenu ne sera pas utilisé, mais dont l'adresse sera utilisée de préférence à `NULL`. C'est une solution relativement proche de celle que nous avons utilisée en OCaml (où il n'est pas possible d'avoir un équivalent direct à `NULL`, puisqu'un nom désigne nécessairement un objet en mémoire), mais c'est une solution *a priori* plus rarement utilisée en C.

Dans la suite de ce cours, nous utiliserons la première solution, mais il convient d'être très clair dans ses choix lorsque l'on crée une structure de ce genre. Commentaires et documentation du code sont indispensables !

## 5.3 Mémoriser une liste

Nous disposons donc de structures, chaînées en mémoire, représentant une liste. Comme en OCaml, il suffit de disposer du début de la liste pour avoir accès à son intégralité. La question est à présent de savoir sous quelle forme on stocke le début de la liste. Là encore, plusieurs solutions sont envisageables.

- On peut directement manipuler le premier élément, donc traiter la liste comme un objet de type « `int_list` ». C'est une solution intéressante car simple. Le problème est que la plupart des cellules de la liste seront construites *dynamiquement*, la mémoire étant allouée au travers d'un `malloc`. En effet, comme on ne connaît généralement pas à l'avance la longueur des listes (c'est leur principal intérêt !), on ne peut pas créer les cellules via des déclarations de variables statiques. Cette solution introduit donc une dissymétrie entre la première cellule (variable statique) et la suite (variables allouées dynamiquement), ce qui, tout en n'étant pas insurmontable, peut rendre certaines fonctions plus lourdes. Par ailleurs, il faudra trouver un moyen de représenter une liste vide !
- Une autre solution consiste à utiliser un *pointeur* vers la première cellule de la liste, et donc mémoriser la liste comme un objet de type « `int_list*` ». C'est probablement la solution la plus simple, sous réserve de prendre garde à bien manipuler les pointeurs. C'est d'ailleurs à peu près la représentation utilisée en OCaml (lorsque l'on manipule un nom, c'est généralement l'adresse de l'objet qu'il désigne qui se cache derrière, ce qui permet d'ailleurs à deux noms distincts de faire référence au même objet). Dans cette situation, on dispose d'une manière simple de désigner une liste vide : le pointeur `NULL` !

36. Si l'on évite les comportements indéfinis, on peut en revanche, en cas d'erreur dans l'algorithme, obtenir aisément des boucles qui ne terminent jamais.
37. En revanche, cette solution peut ne pas être utilisable en fonction des choix que l'on fera pour mémoriser le *début* de la liste.

- Pour cacher quelque peu le pointeur, on peut envisager de créer une structure spécifique pour représenter la liste, structure qui contiendra le pointeur vers le premier élément, et éventuellement d'autres informations (la taille de la liste, par exemple). C'est une solution un peu plus élaborée qui présente des avantages au prix d'un peu plus de définitions.
- Parfois, on ruse un peu... Pour obtenir les mêmes avantages que la solution précédente en s'épargnant la création d'un type supplémentaire, on utilise la structure de la liste elle-même. Après tout, elle contient bien un pointeur de type `int_list*` ! Le champ `value` est simplement ignoré, et il n'y a plus de problème pour désigner une liste vide, il suffit simplement que le champ suivant contienne `NULL`. Cette solution est régulièrement utilisée, mais elle peut être très déroutante, donc nous l'éviterons pour le moment.

Pour le moment, nous opterons pour la seconde solution, autrement dit nous mémoriserons les listes sous forme de pointeurs de type « `int_list*` », car notre but pour le moment est de voir comment les listes peuvent être manipulées, sans se soucier d'un quelconque confort à l'usage. Nous explorerons les possibilités offertes par la troisième approche un peu plus tard, lorsque nous créerons des structures un peu plus souples d'utilisation.

## 5.4 Manipuler une liste

Récapitulons donc les choix que nous avons effectués. Nous allons utiliser le type suivant pour gérer nos listes chaînées :

```
struct cell {
    int value;
    struct cell* next;
};

typedef struct cell int_list;
```

Une liste chaînée sera un pointeur de type `int_list*`, vers la première cellule constituant la liste. Une liste vide sera représentée par le pointeur `NULL`. La dernière cellule de la chaîne sera identifiable par son champ `next` contenant `NULL`. Il est maintenant temps d'écrire des fonctions permettant de manipuler de telles listes.

**Ajout en tête**

Pour ajouter un élément en tête d'une liste, il nous faut comme en OCaml créer une nouvelle cellule, dont le champ `next` pointera vers la liste avant l'ajout. L'adresse de cette nouvelle cellule sera le nouveau point de départ de notre liste chaînée, et cette adresse sera renvoyée par la fonction.

Comme nous l'avons évoqué, la création de cette nouvelle cellule se fera au moyen d'une allocation dynamique. Cela donne [38] :

```c
int_list* add_left(int elem, int_list* p_list) {
    int_list* p_res = malloc(sizeof(int_list));
    (*p_res).value = elem;
    (*p_res).next = p_list;
    return p_res;
}
```

On remarquera en particulier la façon dont la variable « p_res » est manipulée. Elle contient une adresse vers un nouveau maillon (une cellule) de la chaîne. Pour pouvoir accéder aux champs de cette cellule, il faut [39] déréférencer l'adresse avec « * ».

#### Obtention de la tête

Obtenir la tête d'une liste est simple, il s'agit simplement de lire le champ valeur du tout premier maillon (c'est même suffisamment simple pour qu'en général on ne s'embarrasse pas d'une fonction pour faire cela). On peut par exemple écrire :

```c
int head(int_list* p_list) {
    return (*p_list).value;
}
```

Toutefois, la fonction présente un risque : si l'on passe en argument ce qui correspond à une liste vide (autrement dit un pointeur NULL), la fonction va tenter de déréférencer ce pointeur NULL et le programme aura donc un comportement indéfini [40].

On ne dispose pas, en C, de mécanisme permettant véritablement de lever une exception ou provoquer une erreur permettant d'interrompre le programme si l'utilisateur tente d'effectuer une opération incorrecte. Il existe cependant une solution qui peut permettre de détecter une partie des comportements malvenus : assert, fournie par « assert.h ». Il s'agit d'une fonction [41] attendant un booléen qui, si ce booléen est false, interrompt le programme.

38. La conversion explicite du résultat de type void* du malloc en un int_list*, nous l'avons déjà évoqué, est généralement évitée en C, mais nous suivons les recommandations du programme ici.
39. Comme déréférencer un pointeur vers une structure puis accéder à un champ de la structure est une opération très fréquente en C, une notation spécifique existe pour effectuer ces deux opérations : on ainsi peut écrire « p_res->value » plutôt que « (*p_res).value ». C'est la façon naturelle de procéder, mais pour éviter pour le moment d'ajouter une notation supplémentaire risquant d'ajouter à la confusion et essayer d'être le plus clair possible dans cette jungle de pointeurs, nous allons pour l'instant lui préférer le déréférencement explicite avec « * », avant un accès à un champ avec « . ».
40. Déréférencer un pointeur nul aboutit quasi systématiquement en une *erreur de segmentation*, on évitera donc un comportement complètement imprévisible, mais la norme ne le garantit pas.
41. Techniquement, une *macro-commande*.

Avec une petite subtilité toutefois : ce comportement est généralement réservé au débogage du programme. Pour ne pas perdre de temps à effectuer des vérifications, lorsque l'on compile une version définitive du programme, un mécanisme permet de désactiver toutes les assertions [42] ! On cherche donc à écrire du code tel que les arguments des asserts soient tous évalués à false. Mais on peut donc s'offrir un brin de sécurité lors du développement du programme en écrivant :

```c
int head(int_list* p_list) {
    assert(p_list != NULL);

    return (*p_list).value;
}
```

Outre l'avantage durant le déboggage du programme, on peut également voir les assert comme un moyen de documenter la fonction. Même si la chose ne sera généralement pas vérifiée lors de l'exécution, la présence de assert dans la fonction précédente suggère que la fonction attend un pointeur non-nul.

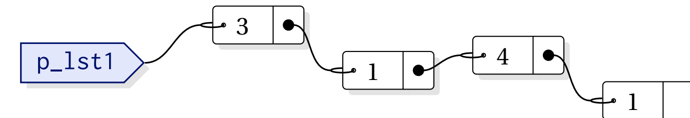
#### Obtention de la queue

Les choses deviennent un peu plus complexes lorsque l'on souhaite obtenir la queue de la liste. Bien évidemment, il s'agit essentiellement de récupérer le champ suivant de la structure. Mais il y a deux problèmes à résoudre.

Le premier problème est de savoir que faire dans le cas d'une liste vide. Comme précédemment, on peut protéger les choses par un assert.

L'autre problème, plus gênant, est de savoir ce qu'il advient de la tête de la liste. En effet, le langage C ne contient pas de ramasse-miettes, et n'est pas en mesure de libérer automatiquement la mémoire allouée et « perdue ». À tout appel à malloc doit correspondre un (unique) appel à free ! Est-il donc de la responsabilité de la fonction fournissant la queue ou de l'appelant de libérer cette mémoire ? La réponse n'est pas évidente, car on peut envisager plusieurs situations.

Supposons par exemple que p_lst1 est un pointeur vers une liste chaînée d'entiers telle que celle représentée ci-dessous :




Le lecteur avisé aura remarqué sur cette représentation l'absence de l'objet « Nil » que l'on avait dans nos représentations de listes chaînées en OCaml. En effet, dans nos listes C, la fin de la chaîne est simplement identifiée par un pointeur NULL dans le champ suivant

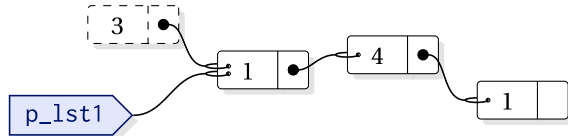
42. En cela, le comportement est identique à ce qui se passe en Python avec la commande Python assert.

de la dernière cellule, il n'y a pas d'objet supplémentaire pour identifier l'extrémité de la liste.

On peut vouloir écrire, afin de retirer la tête d'une liste :

```
p_lst1 = tail(p_lst1);
```

Ce qui conduit donc à cette situation :

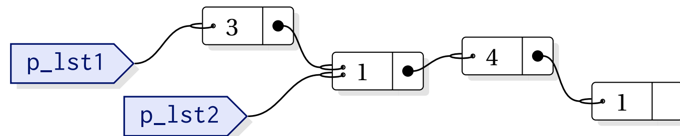



Dans ce premier cas, il est probable que l'on perde tout accès à la cellule précédemment en tête de la liste pointée par p_lst1, et donc que l'on ait ici une fuite de mémoire si la fonction tail ne libère pas la mémoire allouée pour mémoriser la tête de la liste.

Mais on peut également vouloir écrire :

```
int_list* p_lst2 = tail(p_lst1);
```

Ce qui conduit à cette situation :




Dans ce second cas, on cherche à obtenir la queue de la liste (au travers du pointeur p_lst2) *tout en conservant la liste dans son ensemble via le pointeur* p_lst1 ! On ne veut donc *surtout pas* libérer la mémoire correspondant à la première cellule.

Ne sachant pas ici dans quelle situation concrète on se trouvera, on va explicitement ajouter un argument booléen précisant ce qu'il convient de faire (dans un programme réel, en général, on fera un choix éclairé entre les deux possibilités, car on n'a pas souvent besoin des deux comportements) :

```
int_list* tail(int_list* p_list, bool free_head) {
    assert (p_list != NULL);

    int_list* p_res = (*p_list).next;
    if (free_head) {
        free(p_list);
    }
    return p_res;
}
```

En langage C, la question de savoir qui est responsable de la désallocation de zones mémoire allouées dynamiquement est cruciale. Si en C++ il existe des solutions pour clarifier les choses (et des structures pour se charger de ces questions), en C, cela repose uniquement sur la discipline du programmeur et la documentation !

**Longueur de la liste**

Pour obtenir la longueur d'une liste chaînée, on pourrait procéder comme en OCaml et utiliser une fonction récursive, telle que :

```
int length(int_list* p_list) {
    if (p_list == NULL) {
        return 0;
    }
    return 1 + length((*p_list).next);
}
```

On évitera généralement d'utiliser la fonction tail ici, d'abord pour éviter un appel de fonction supplémentaire qui n'est guère nécessaire, mais possiblement aussi parce que la fonction tail pourrait être écrite de façon à libérer la mémoire allouée à la tête, ce dont on ne veut surtout pas ici [43] !

En général, on préférera se tourner vers une solution itérative pour éviter la suite (assez coûteuse) d'appels récursifs. Pour ce faire, nous utiliserons un pointeur (ci-dessous [44] nommé ptr) qui va « visiter » successivement chacun des maillons de la liste, en partant du tout premier, et en passant de proche en proche en utilisant [45] « ptr = (*ptr).next ». Le parcours de la liste s'arrête lorsque le pointeur ptr se voit attribuer la valeur NULL, indiquant qu'il est parvenu au bout de la liste. Pour calculer la longueur de la liste, il suffit alors de simplement compter le nombre de passages d'un maillon au suivant :

```
int length(int_list* p_list) {
    int count = 0;
    int_list* ptr = p_list;
    while (ptr != NULL) {
        count = count + 1;
        ptr = (*ptr).next;
    }
    return count;
}
```

43. On peut utiliser notre propre fonction tail à condition de choisir false comme second argument.
44. La création de la variable ptr n'est pas indispensable, on aurait pu directement modifier la variable p_list. C'est en effet une variable locale, donc la modifier n'a aucun effet sur la liste à l'extérieur de la fonction.
45. Un peu comme si, en OCaml, on utilisait une référence r initialisée avec la liste entière, et qu'à chaque étape on utilisait « r := List.tl r ».

On remarque que l'on retrouve ici tous les éléments constitutifs d'une boucle `for` : initialisation, test d'arrêt et opération pour passer au suivant. Il est donc fréquent d'écrire plutôt :

```
int length(int_list* p_list) {
    int count = 0;
    for (int_list* ptr = p_list; ptr != NULL; ptr = (*ptr).next) {
        count = count + 1;
    }
    return count;
}
```

### Recherche

Cette structure est très idiomatique [46] et nous la rencontrerons souvent. Par exemple, sur le même modèle, déterminer si la liste contient un élément égal à un autre élément passé en paramètre ne pose pas de souci :

```
bool member(int_list* p_list, int elem) {
    for (int_list* ptr = p_list; ptr != NULL; ptr = (*ptr).next) {
        if ((*ptr).value == elem) {
            return true;
        }
    }
    return false;
}
```

De même, pour déterminer l'index dans la liste d'un tel élément (et -1 en cas d'absence), on emploie un compteur :

```
int index(int_list* p_list, int elem) {
    int count = 0;
    for (int_list* ptr = p_list; ptr != NULL; ptr = (*ptr).next) {
        if ((*ptr).value == elem) {
            return count;
        }
        count = count + 1;
    }
    return -1;
}
```

46. Elle a conduit naturellement à la notion d'*itérateur* dans d'autres langages (dont le C++). Un itérateur est un objet associé à autre un objet pouvant contenir plusieurs éléments, et ayant un moyen d'*avancer* et de vérifier s'il est en mesure de continuer à le faire.

### Retournement de liste

Pour créer une *nouvelle* liste [47] correspondant au retournement d'une liste passée en paramètre simplement en itérant sur les cellules de cette liste :

```
int_list* reverse(int_list* p_list) {
    int_list* p_res = NULL;
    for (int_list* ptr = p_list; ptr != NULL; ptr = (*ptr).next) {
        int_list* tmp = malloc(sizeof(int_list));
        (*tmp).value = (*ptr).value;
        (*tmp).next = p_res;
        p_res = tmp;
    }
    return p_res;
}
```

### Destruction de la liste

Lorsque l'on n'a plus besoin d'une liste, on ne peut pas simplement s'en débarasser : comme chaque cellule a été allouée par un appel à `malloc`, il faut impérativement exécuter des appels à `free` pour chacune des cellules. On peut donc créer une fonction qui désalloue tous les éléments d'une liste [48] :

```
void delete(int_list* p_list) {
    while (p_list != NULL) {
        int_list* tmp = p_list;
        p_list = (*p_list).next;
        free(tmp);
    }
}
```

Notons qu'on ne peut pas utiliser la boucle `for` usuelle ici, car la désallocation du maillon ne permet (théoriquement) pas d'utiliser `(*ptr).next` à l'issue de l'itération pour passer au maillon suivant. Il peut être plus simple d'envisager une solution récursive :

```
void delete(int_list* p_list) {
    if (p_list != NULL) {
        delete((*p_list).next);
        free(p_list);
    }
}
```

47. Attention, « `p_lst = reverse(p_lst)` » cause une fuite de mémoire !
48. Il ne faudra évidemment plus déréférencer le pointeur passé en argument après l'appel !

## 5.5 Mutabilité des listes

Si les algorithmes sur les listes chaînées en C seront généralement très similaires à ceux étudiés avec le langage OCaml, il y a une différence majeure entre les deux structures que nous n'avons pas encore souligné : les champs des cellules qui constituent la liste sont, en C, modifiables à tout moment [49]. Nos listes en C sont donc des structures *mutables*.

On peut ainsi modifier le contenu des cellules, par exemple incrémenter toutes les valeurs contenues dans la liste comme dans la fonction ci-dessous :

```
void increment(int_list* p_list) {
    for (int_list* ptr = p_list; ptr != NULL; ptr = (*ptr).next) {
        (*ptr).value = (*ptr).value + 1;
    }
}
```

On notera en particulier que la fonction ne renvoie rien : c'est logique, puisque c'est la liste passée en argument qui est altérée, on n'en construit pas une *nouvelle* comme on l'aurait fait avec une liste OCaml en écrivant :

```
# let rec increment = function
    | [] -> []
    | h::t -> h+1::increment t;;

val increment : int list -> int list = <fun>
```

Mais cela ne s'arrête pas à l'altération des valeurs, on peut très bien modifier l'ordre des cellules dans la liste, en retirer ou en ajouter. Par exemple, la fonction suivante permet d'ajouter un élément *à droite* d'une liste *non-vide* :

```
void add_right(int elem, int_list* p_list) {
    assert( p_list != NULL )

    while ((*p_list).next != NULL) {           // On recherche le
        p_list = (*p_list).next                // dernière cellule
    }
    int_list* tmp = malloc(sizeof(int_list)); // On crée une
    (*tmp).value = elem                        // nouvelle cellule
    (*tmp).next = NULL                         // et on l'accroche
    (*p_list).next = tmp;                      // en bout de liste
}
```

49. En OCaml, avec les définitions proposées, ces champs sont immutables après la création de l'objet.

Nous avons imposé la condition que la liste ne devait pas être vide, car la fonction modifie le pointeur contenu dans la dernière cellule. On peut très bien considérer qu'un ajout à droite dans une liste vide produit une liste à un seul élément. Cependant, cela nécessite de modifier le pointeur désignant la liste : il ne contiendra plus `NULL` mais l'adresse de cette toute nouvelle cellule. Il faudrait donc quelque peu altérer la signature de notre fonction. C'est pour des cas particuliers de ce genre qu'il peut être utile d'avoir une structure contenant la liste plutôt qu'un simple pointeur vers la première cellule (et un pointeur `NULL` si la liste est vide), car cela permet d'éliminer la différence fondamentable entre les listes vides et les listes qui ne le sont pas.

La mutabilité des champs d'un enregistrement permet également de faciliter la copie d'une liste sans nécessiter de recourir à la récursion. Pour ce faire, on conserve non seulement un pointeur vers la première cellule de la copie de la liste [50] que l'on construit, mais également un pointeur vers sa *dernière* cellule, pour pouvoir éventuellement changer le champ `next` lors de l'ajout de la cellule suivante :

```
void copy(int_list* p_list) {
    int_list* p_copy = NULL;
    int_list* p_last = NULL;
    for (int_list* ptr = p_list; ptr != NULL; ptr = (*ptr).next) {
        int_list* tmp = malloc(sizeof(int_list));
        (*tmp).value = (*ptr).value;
        (*tmp).next = NULL;
        if (p_copy == NULL) { // Si c'est la première cellule
            p_copy = tmp;
        } else {
            (*p_last).next = tmp;
        }
        p_last = tmp;
    }
    return p_copy;
}
```

50. Initialisé à `NULL`, et le reste si la liste copiée est vide, ce pointeur est modifié après la création de la première cellule de la liste, et n'est plus modifié ensuite.

# Exercices

## Ex. 6.1 – Rationnels

On définit les rationnels par le type suivant, où $n$ est un entier non nul représentant le numérateur et $d$ un entier non nul représentant le dénominateur :

```
type rationnal = { n: int; d: int };
```

**1.** Proposer une fonction `r_prod` de signature `rationnal -> rationnal -> rationnal` calculant le produit de deux nombres rationnels (on supposera le produit représentable).

**2.** Proposer une fonction `r_inv` de signature `rationnal -> rationnal` calculant l'inverse d'un nombre rationnel non nul (on déclenchera une erreur si l'argument est nul).

**3.** Proposer une fonction `r_sum` de signature `rationnal -> rationnal -> rationnal` calculant la somme de deux nombres rationnels [51].

**4.** Proposer une fonction `r_equals` de signature `rationnal -> rationnal -> bool` déterminant si deux nombres rationnels sont égaux.

## Ex. 6.2 – Nombres de Gauss

Les nombres de Gauss sont les complexes de la forme $a+ib$ où $a$ et $b$ sont des entiers relatifs. On définit un type pour les représenter :

```
type gauss = { re: int; im: int };
```

**1.** Proposer une fonction `g_add` de signature `gauss -> gauss -> gauss` calculant la somme de deux nombres de Gauss.

**2.** Proposer une fonction `g_mul` de signature `gauss -> gauss -> gauss` calculant le produit de deux nombres de Gauss.

## Ex. 6.3 – Quelques fonctions sur les listes

**1.** Sur le modèle de la fonction `last`, écrire une fonction `penultimate` prenant en argument une liste Caml et renvoyant son avant-dernier élément. On déclenchera une erreur (avec `failwith`) si la liste contient moins de deux éléments.

**2.** Écrire une fonction `firsts` prenant en argument une liste et un entier $k$, et renvoie la liste des $k$ premiers éléments. On déclenchera une erreur si la liste est trop courte.

51. Pour cette question et la suivante, on se contentera d'une première version élémentaire ne gérant pas les possibles débordement lors du calcul, mais on pourra réfléchir à la meilleure façon de les éviter.

**3.** Écrire une fonction `remove` prenant en argument une liste et un entier $k$, et renvoie une liste identique à la liste fournie, mais privée de l'élément se trouvant à la position d'index `k`. On déclenchera une erreur si la liste est trop courte.

**4.** Écrire une fonction `insert` prenant en argument une liste, un entier $k$ et un élément, et renvoie une liste similaire à la liste fournie, mais dans laquelle on a inséré l'élément fourni de sorte qu'il se trouve à la position d'index `k` dans la liste renvoyée (les autres éléments sont les éléments de la liste initial, dans leur ordre original). On déclenchera une erreur si la liste fournie ne contient pas au moins `k` éléments.

**5.** Discuter la complexité (en temps) de chacune de ces fonctions.

## Ex. 6.4 – Retirer les répétitions

Proposer une fonction `remove_ditto` de signature `'a list -> 'a list` prenant en argument une liste et renvoyant une liste où ont été retirés les éléments égaux à celui qui les précède. Ainsi, `remove_ditto [ 1; 2; 2; 3; 1; 1; 1; 2 ]` devra renvoyer `[ 1; 2; 3; 1; 2 ]`.

## Ex. 6.5 – Réordonner les éléments d'une liste

**1.** Écrire une fonction `rotG` de complexité linéaire prenant en argument une liste et renvoie une liste dans laquelle est éléments ont subi une permutation circulaire vers la gauche. Par exemple, `rotG [ 1; 2; 3; 4 ]` doit donner `[ 2; 3; 4; 1 ]`.

**2.** Écrire une fonction `rotD` de complexité linéaire prenant en argument une liste et renvoie une liste dans laquelle est éléments ont subi une permutation circulaire vers la droite. Par exemple, `rotD [ 1; 2; 3; 4 ]` doit donner `[ 4; 1; 2; 3 ]`.

## Ex. 6.6 – Suppression de doublons

**1.** Quelle est la signature de la fonction suivante, et que fait-elle ?

```
let rec foo f = function
  | t::q when f t q -> foo f q
  | t::q            -> t::(foo f q)
  | _               -> [];;
```

**2.** En déduire une fonction prenant en argument une liste et renvoyant une liste dans laquelle on a retiré les éléments n'apparaissant pas pour la dernière fois. Le résultat de la fonction sur la liste `[ 1; 2; 3; 2; 4; 5; 4 ]` devra être la liste `[ 1; 3; 2; 5; 4 ]`. Quelle est sa complexité ?

## Ex. 6.7 – Préfixes et suffixes

**1.** Proposer une fonction `suffixes` prenant en argument une liste d'éléments (de type quelconque) et renvoyant la liste de ses suffixes. Par exemple, `suffixes [ 1; 2; 3 ]` doit renvoyer par exemple `[ [ 1; 2; 3 ] ; [ 2; 3 ] ; [ 3 ] ]` (l'ordre des listes dans la liste fourni comme résultat n'a pas d'importance).

**2.** Un peu plus difficile, proposer une fonction `prefixes` prenant en argument une liste d'éléments (de type quelconque) et renvoyant la liste de ses préfixes. Par exemple, `prefixes [ 1; 2; 3 ]` doit renvoyer par exemple `[ [ 1 ] ; [ 1; 2 ] ; [ 1; 2; 3 ] ]` (l'ordre des listes dans la liste fourni comme résultat n'a pas d'importance).

**3.** Quelle est la complexité de ces deux fonctions ? Peut-on envisager mieux ?

## Ex. 6.8 – Booléens et listes

**1.** Écrire une fonction `p_count` de signature `('a -> bool) -> 'a list -> int` prenant un prédicat (une fonction prenant un élément et renvoyant un booléen) et une liste, et renvoyant le nombre d'éléments de la liste vérifiant le prédicat (c'est à dire pour lequel la fonction passée en argument renvoie `true`).

**2.** Écrire une fonction `p_last` de signature `('a -> bool) -> 'a list -> 'a` prenant un prédicat et une liste, et renvoyant le dernier élément de la liste vérifiant le prédicat.

**3.** Écrire une fonction `p_antifilter` de signature `('a -> bool) -> 'a list -> 'a list` prenant un prédicat et une liste, et renvoyant la liste des éléments de la liste ne vérifiant *pas* le prédicat (en en préservant l'ordre).

## Ex. 6.9 – Produit cartésien

Proposer une fonction de signature `'a list -> 'b list -> ('a * 'b) list` prenant en argument deux listes et renvoyant une liste de couples résultat du produit cartésien des deux listes fournies en argument (les couples se trouvant dans un ordre quelconque). `produit [ 'a'; 'b'; 'c' ] [ 1; 2; 3 ]` renverra ainsi par exemple :

```
- : (int * char) list = [(1, 'a'); (1, 'b'); (1, 'c');
(2, 'a'); (2, 'b'); (2, 'c'); (3, 'a'); (3, 'b'); (3, 'c')]
```

## Ex. 6.10 – Factorielle

**1.** Proposer une fonction `multiply` qui prend en argument une liste d'entiers et renvoie le produit de ses éléments (ou 1 si la liste est vide). On pourra réfléchir à une version utilisant le filtrage, et une version utilisant `List.fold_left`.

**2.** Écrire une fonction prenant un entier $n$ et renvoyant la liste des entiers de $n$ à 2 (inclus, rangés par ordre décroissants) si $n \geqslant 2$ et une liste vide sinon.

**3.** En déduire une fonction `fact` prenant en argument un entier $n$ et renvoyant sa factorielle.

## Ex. 6.11 – Éléments communs

On considère deux listes `lst1` et `lst2`, chacune de ces listes contenant des éléments distincts deux à deux ordonnés par ordre croissant.

Proposer une fonction `common` de signature `'a list -> 'a list -> 'a list` prenant en argument ces deux listes `lst1` et `lst2`, de complexité linéaire en la somme des longueurs des deux listes, et renvoyant la liste des éléments communs aux deux listes.

## Ex. 6.12 – Diviseurs

**1.** Proposer une fonction `divisors` de signature `int -> int list` prenant en argument un entier `n` et renvoyant la liste de ses diviseurs positifs, dans un ordre croissant. On attend une complexité sous-linéaire, que l'on précisera.

Un nombre $n$ est dit *abondant* si la somme de ses diviseurs $\sigma(n)$ vérifie $\sigma(n) \geqslant 2n$.

**2.** Proposer une fonction `abundant` de signature `int -> bool` prenant en argument un entier `n` et renvoyant un booléen indiquant s'il est abondant.

**3.** En déduire une fonction `abundants` de signature `int -> int list` prenant en argument un entier `n` et renvoyant la liste des entiers abondants inférieurs ou égaux à `n` rangés par ordre croissant.

## Ex. 6.13 – Différences

**1.** Proposer une fonction `diff1` de signature `int list -> int` prenant en argument une liste $[a_0, a_1, \dots, a_{n-1}]$ et renvoie

$$\prod_{i=1}^{n-1}(a_i - a_{i-1})$$

**2.** Proposer une fonction `diff2` de signature `int list -> int` prenant en argument une liste $[a_0, a_1, \dots, a_{n-1}]$ et renvoie

$$\prod_{j<i}(a_i - a_j)$$

## Ex. 6.14 – Polynomes

On définit deux types :

```
type monomial = float * int
type polynomial = monomial list
```

On suppose qu'un polynôme est représenté par un objet de type « `polynomial` » où les monômes sont rangés par ordre de puissance *croissantes*. Le terme de type « `float` » dans un monôme désigne le coefficient, celui de type « `int` » la puissance. Il ne doit pas y avoir de coefficient nul parmi les coefficients d'un polynôme, ni plusieurs monômes correspondant à la même puissance.

**1.** Proposer une fonction `p_eval` de signature `float -> polynome -> float` prenant en argument un réel $x$ et un polynôme P et renvoyant la valuation $P(x)$ du polynome pour le réel considéré.

**2.** Proposer une fonction `p_sum` sommant deux polynômes.

**3.** Proposer une fonction `p_derivative` dérivant un polynôme.

**4.** Proposer une fonction `m_product` multipliant un monôme avec un polynôme, et en déduire une fonction `p_product` multipliant deux polynômes.

## Ex. 6.15 – Compression RLE

Lorsque, dans une liste, beaucoup d'éléments consécutifs sont identiques, la compression dite RLE, pour « *run-length encoding* » peut être efficace. Elle consiste simplement à mémoriser le nombre de répétitions consécutives d'un élément dans la liste. Pour ce faire, on définit

```
type 'a rle =
  | One of 'a
  | Many of int * 'a
```

**1.** Proposer une fonction encode de signature `'a list -> ('a rle) list` encodant une liste en RLE. Par exemple :

```
# encode [ 1; 1; 1; 2; 3; 3; 4; 4; 4; 4; 1; 2; 2; 2 ];;
- : int rle list = [Many (3, 1); One 2; Many(2, 3);
                                  Many(4, 4); One 1; Many(3, 2)]
```

**2.** Proposer une fonction decode effectuant la transformation inverse.

**3.** Proposer une fonction `length_rle` de signature `'a rle list -> int` renvoyant la longueur qu'aurait la liste une fois décompressée *sans la décompresser*.

**4.** Proposer de même une fonction `sum_rle` de signature `int rle list -> int` renvoyant la somme des éléments (ici supposés entiers) de la liste décompressée sans décompresser la liste.

**5.** Proposer enfin une fonction `nth_rle` de signature `'a rle list -> int -> 'a` renvoyant l'élément d'index `i` de la liste décompressée toujours sans décompresser la liste.

## Ex. 6.16 – Séparation et jointure

**1.** Proposer une fonction `split` de signature `'a list -> 'a list * 'a list` prenant en argument une liste et renvoyant un couple de deux listes, la première contenant les éléments de la liste `lst` aux positions d'index pairs (dans l'ordre), la seconde les éléments aux positions d'index impairs.

**2.** Proposer une fonction `interleave` de signature `'a list * 'a list -> 'a list` prenant en argument un couple de deux listes et construisant une liste consituté du premier élément de la première liste, suivi du premier élément de la seconde liste, puis du second élément de la première liste, et ainsi de suite. Lorsque l'on parvient au bout d'une liste, on place dans le résultat l'ensemble des éléments de la liste restante.

**3.** Proposer une fonction `zip` de signature `'a list * 'a list -> ('a * 'a) list` prenant en argument deux listes et construisant une liste de couples, le premier couple étant constitué du premier élément de chacune des deux listes, le second du second élément de chacune des deux listes, et ainsi de suite jusqu'à épuisement d'une des listes.

## Ex. 6.17 – Algorithme de Chvátal

**1.** Proposer une fonction `transform` de signature `int list -> int list` prenant en argument une liste d'entiers non vide et renvoyant la liste d'entiers obtenue par retournement des $k$ premiers éléments où $k$ est la tête de la liste (`[4; 1; 2; 5; 3; 6]` donnera `[5; 2; 1; 4; 3; 6]`. On lèvera une erreur s'il n'y a pas suffisamment d'éléments ou si $k \leqslant 0$. On attend une complexité en $O(k)$.

**2.** En déduire une fonction `transforme_itere` de même signature qui applique la fonction précédente tant que la tête de la liste n'est pas 1.

**3.** Étudier le déroulement de l'algorithme précédent pour la liste `[6; 3; 1; 7; 2; 5; 4]`.

**4.** Discuter de la terminaison de la fonction `transforme_itere` si l'argument est une liste à $n$ éléments contenant, dans un ordre quelconque les entiers de 1 à $n$. On pourra s'intéresser à la grandeur

$$\sum_{k=1}^{n} 2^k \mathbb{1}_{l[k-1]=k}$$

où $l[k]$ est l'élément à la position d'index $k$ dans la liste.

### Ex. 6.18 – Entrelacement de listes

Proposer une fonction `interleaves` de signature `'a list -> 'a list -> 'a list list` prenant en argument deux listes et renvoyant la liste, dans un ordre quelconque, de toutes les listes résultant d'une fusion des deux listes préservant l'ordre relatif des éléments de chacune de ces deux listes. On aura par exemple :

```
# interleaves [1; 2; 3] [4; 5];;
-: int list list = [[1; 2; 3; 4; 5]; [1; 2; 4; 3; 5]; [1; 2; 4; 5; 3];
[1; 4; 2; 3; 5]; [1; 4; 2; 5; 3]; [1; 4; 5; 2; 3]; [4; 1; 2; 3; 5];
[4; 1; 2; 5; 3]; [4; 1; 5; 2; 3]; [4; 5; 1; 2; 3]]
```

### Ex. 6.19 – Sous-séquences

**1.** Proposer une fonction `subsequences` de signature `'a list -> int -> 'a list list` telle que « `subsequences p lst` » sur une liste $[a_0, a_1, a_2, \ldots, a_{n-1}]$, renvoie la liste de listes $[[a_0, a_1, a_2, \ldots, a_{p-1}], [a_1, a_2, a_3, \ldots, a_p], \ldots, [a_{n-p}, a_{n-p+1}, a_{n-p+2}, \ldots, a_{n-1}]]$.

**2.** Quelle est sa complexité en fonction de $n$ et $p$ ? Peut-on faire mieux ?

### Ex. 6.20 – Fonctions mystérieuses

On propose les fonctions suivantes :

```
let rec foo lst p = function
  | 0 -> lst
  | n -> p::foo lst p (n-1)

let rec bar = function
  | [] -> 0
  | t::q -> 1 + bar (foo q (t-1) t)
```

**1.** Déterminer ce que fait la première fonction.

**2.** Justifier que la seconde fonction termine quel que soit la liste (finie) passée en argument [52].

### Ex. 6.21 – Mélanges de listes

Proposer une fonction `shuffles` de signature `'a list -> 'a list list` prenant en argument une liste et renvoyant la liste de toutes ses permutations, dans un ordre quelconque. On aura par exemple :

```
# shuffle [1; 2; 3];;
-: int list list = [[1; 2; 3]; [1; 3; 2]; [2; 1; 3]; [2; 3; 1];
[3; 1; 2]; [3; 2; 1]]
```

## 5.6 Suite de Conway

On considère la suite de Conway (suite A005150 dans l'OEIS) dite « audioactive » :

$$1 \mapsto 1,1 \mapsto 2,1 \mapsto 1,2,1,1 \mapsto \ldots$$

**1.** Proposer une fonction prenant une liste d'entiers et renvoyant la liste des entiers correspondant à l'itération suivante.

**2.** En déduire une fonction prenant une liste initiale et un nombre d'itérations $n$ et renvoyant la liste obtenue après $n$ itérations.

**3.** Prouver que si l'on part de la liste `[1]`, on ne voit jamais apparaître d'autres entiers que 1, 2 et 3.

La constante de Conway $\lambda$ est la limite de $L_{n+1}/L_n$ où $L_n$ est la longueur de la liste à l'étape $n$. Elle vaut environ 1,303 577 et ne dépend pas de la liste initiale.

### Ex. 6.22 – Dragon curve

On considère une bande de papier. On effectue $n$ plis successifs au centre de cette bande, toujours dans le même sens, puis on déplie la bande, comme illustré ci-dessous. On note 0 et 1 les plis selon leur sens. Les premiers termes sont $[0]$, $[0,0,1]$, $[0,0,1,0,0,1,1]$, ...

(Images issues du site `www.cutoutfoldup.com`, la figure de droite correspond à 5 plis)

**1.** Proposer une fonction `next` de signature `int list -> int list` pour passer d'une liste à la suivante.

52. Si l'on appelle la fonction `bar` avec pour argument les listes `[n]`, on obtient les éléments de la suite A000522 de l'OEIS.

**2.** En déduire une méthode renvoyant le $n^e$ terme.

**3.** Si on part d'une bande de papier de longueur 1 m, et que on ramène tous les plis à 90° (comme ci-dessus), quelle sera la distance entre les extrémités? Proposer une fonction `dist_dragon` de signature `int -> float`, prenant le nombre de plis effectués et renvoyant cette distance.

## Ex. 6.23 – Nombres chanceux (C)

Pour construire la liste des nombres chanceux, on considère la liste des nombres premiers impairs. Le deuxième élément de cette liste étant 3, on élimine les 3ᵉ, 6ᵉ, 9ᵉ... éléments de la liste. Le troisième élément de la liste étant dorénavant 7, on élimine les 7ᵉ, 14ᵉ, 21ᵉ... éléments de la liste. Et ainsi de suite.

On définit le type suivant :

```
struct list {
    int n;
    struct list* next;
};

typedef struct list list;
```

**1.** Proposer une fonction `list* build(int n)` renvoyant une liste contenant les entiers impairs de 1 à $n$ (inclus).

**2.** Proposer une fonction `list* lucky(int n)` renvoyant une liste contenant les entiers chanceux inférieurs ou égaux à $n$ (on prendra garde à éviter les fuites de mémoire).

## Ex. 6.24 – Élimination de doublons (C)

On considère le type suivant :

```
struct list {
    int n;
    struct list* next;
};

typedef struct list list;
```

**1.** Proposer une fonction `list* remove(int n, list* lst)` supprimant toutes les occurrences de `n` dans la liste chaînée `lst` (on désallouera les cellules supprimées) et renvoie un pointeur vers la (possible nouvelle) tête de la liste.

On prendra soin, dans cette question et la suivante, de ne pas provoquer de fuite de mémoire! On supposera que les éléments égaux à $n$ en tête de la liste `lst` peuvent être détruits sans problème.

**2.** En déduire une fonction `list* remove_dupes(list* lst)` qui supprime les doublons dans une liste chaînée d'entiers et renvoie un pointeur vers la (possible nouvelle) tête de la liste.

## Ex. 6.25 – Tris de liste (C)

On considère le type suivant :

```
struct list {
    double val;
    struct list* next;
};

typedef struct list list;
```

**1.** Proposer une fonction `list* insertion(list* lst)` prenant en argument une liste non-vide et insérant sa tête dans le reste de la liste. La fonction renvoie un pointeur vers la tête de la liste après insertion.

Note : aucune allocation/libération de mémoire n'a lieu ici, l'opération consiste en un réarrangement de l'ordre des éléments de la liste!

**2.** En déduire une fonction `list* sort(list* lst)` triant une liste par l'algorithme du tri par insertion.

**3.** Proposer une fonction `list* swap(list* lst)` qui permute les deux premiers éléments de la liste passée en argument et renvoie un pointeur vers la nouvelle tête si la liste contient au moins deux éléments et que l'élément de tête est strictement plus grand que le second élément, et renvoie son argument sans rien changer sinon.

On propose le code suivant :

```
list* foo(list* lst) {
    while (lst != NULL) {
        lst = swap(lst);
        lst = (*lst).next;
    }
}
```

**4.** Quelle opération effectue la fonction `foo`?

**5.** En déduire une autre fonction permettant de trier une liste.

# 7 Programmation OCaml impérative

« Programming languages teach you not to want what they don't provide. »

— *Paul Graham*

## 1 Outils de programmation impérative

### 1.1 Séquences d'instructions

Dans une programmation de style impératif, les instructions se succèdent les unes aux autres. Il est donc nécessaire de pouvoir exécuter plusieurs instructions à la suite. En OCaml, les « instructions » (en réalité des expressions) seront séparées par des points-virgules [1] (même si un retour à la ligne sépare les deux instructions).

```
# let foo x =
    print_string "Le carré de "; print_int x;
    print_string " est "; print_int (x * x);
    print_string "."; print_newline ();;

val foo : int -> unit = <fun>
```

Le « résultat » d'une telle séquence d'expressions est le résultat de la *dernière* expression de la séquence. Donc celui qui sera renvoyé par une fonction constituée d'une séquence d'expressions :

```
# let foo x =
    print_string "Calcul du carré de ";
    print_int x; print_newline ();
    x * x;;

val foo : int -> int = <fun>
```

1. Il n'est donc pas requis de ne mettre une seule instruction par ligne, tant qu'elles sont séparées par des points virgules, même si l'on tend souvent à le faire pour faciliter la lecture des fonctions.

Dans cette dernière fonction, on procède d'abord à un affichage, puis la fonction calcule et renvoie le carré de son argument (entier).

```
# foo 2;;
Calcul du carré de 2
- : int = 4
```

Bien évidemment, les résultats des expressions, excepté celui de la toute dernière, sont perdus. Chacune d'elle ne devrait donc « rien » renvoyer, ou plutôt renvoyer l'élément `()` de type `unit`. Ce n'est pas rigoureusement requis (si ce n'est pas le cas, les calculs intermédiaires sont simplement perdus), quoique Caml fournira un avertissement :

```
# let foo x =
    x + 1; x * x;;

Characters 18-23:
      x + 1; x * x;;
      ^^^^^
Warning 10: this expression should have type unit.
val foo : int -> int = <fun>
```

C'est un simple avertissement, la fonction est définie quand même, mais elle ne renvoie que le carré de l'argument (la première expression, ici, ne sert à rien!). Pour que les expressions, la dernière excepté, aient un intérêt, il faut qu'elles aient un effet (affichage à l'écran, modification de la mémoire...), comme dans le cas d'appels aux fonctions « `print_int` », « `print_string` », « `print_newline` »...

### 1.2 Blocs

Dans certains cas, il n'est pas possible d'utiliser une séquence sans précautions, par exemple dans le cadre de la structure `if ... then ... else ...` : une *unique* expression doit en effet suivre le `then` et le `else` :

```
# let foo n =
    if n mod 2 = 1 then
      print_string "impair"; print_newline ()
    else
      print_string "pair"; print_newline ();;

Characters 95-99:
      else
      ^^^^
Error: Syntax error
```

Il est nécessaire de « grouper » la séquence d'instructions dans un « bloc » qui se comportera comme une expression unique. Cela peut se faire avec de simples parenthèses :

```
# let foo n =
    if n mod 2 = 1 then
      (print_string "impair"; print_newline ())
    else
      (print_string "pair"; print_newline ())

val foo : int -> unit = <fun>
```

Mais cette solution n'est souvent pas assez lisible, aussi OCaml propose une alternative aux parenthèses, grâce aux mots-clés **begin** et **end** :

```
# let foo n =
    if n mod 2 = 1 then
      begin
        print_string "impair"; print_newline ();
      end
    else
      begin
        print_string "pair"; print_newline ();
      end;;

val foo : int -> unit = <fun>
```

C'est également indispensable pour l'expression associée au **else** ici, car sinon le second `print_newline` ne ferait pas partie de la conséquence de l'échec de la condition `n mod 2 = 1`, mais serait exécuté quelle que soit la valeur de `n`, puisque le **else** ne prendra que l'expression qui le suit immédiatement.

Ce type de bloc est également utile lorsque l'on a des filtrages imbriqués, afin de préciser à quel filtrage appartient chacun des motifs !

Considérons par exemple le cas suivant, où les `m1`...`m4` seraient des motifs :

```
match expr1 with
  | m1 -> match expr2 with
            | m2 -> ...
            | m3 -> ...
  | m4 -> ...
```

Contrairement à ce que l'on a pu vouloir écrire (ou que l'on peut comprendre en première lecture), le motif `m4` est un motif pour le *second* filtrage (on rappelle que Caml n'est pas sensible à l'indentation).

Si `m4` est bien un motif pour le *premier* filtrage, comme l'indentation le laisse supposer, il conviendrait d'écrire :

```
match expr1 with
  | m1 -> begin
            match expr2 with
              | m2 -> ...
              | m3 -> ...
          end
  | m4 -> ...
```

ou bien :

```
match expr1 with
  | m1 -> (match expr2 with
             | m2 -> ...
             | m3 -> ... )
  | m4 -> ...
```

## 1.3 Boucles inconditionnelles (for)

Pour effectuer un nombre déterminé de fois une série d'instructions, on écrira

```
for nom = expression_1 to expression_2 do sequence done
```

`expression_1` et `expression_2` doivent donner un résultat entier.

Le `nom` est alors associé successivement à tous les entiers entre `expression_1` et `expression_2` (inclus), et la séquence d'instructions `sequence` est évaluée pour chacun de ces entiers (notons que par la présence de **do** et **done**, il n'est pas besoin de définir nous-même un bloc d'instructions ici s'il faut plus d'une instruction dans la boucle.

Ainsi, l'expression

```
for i = 1 to 4 do expression done
```

est équivalente à la séquence d'instructions suivante :

```
let i = 1 in expression;
let i = 2 in expression;
let i = 3 in expression;
let i = 4 in expression;
```

Précisons que, dans le cas où `expression_1` donne un résultat strictement supérieur à `expression_2`, il ne se passera rien.

On peut ainsi, par exemple, avec une boucle inconditionnelle, écrire une fonction qui imprime une table de multiplication[2] :

```
# let table n =
    for i = 1 to 10 do
      print_int n; print_string " fois ";
      print_int i; print_string " égale ";
      print_int (n*i); print_newline ();
    done;;

val table : int -> unit = <fun>
```

Les expressions dans la boucle devraient renvoyer un `()` de type `unit`, car le résultat de ces expressions sera « jeté » par Caml après chaque itération[3]. Ainsi, la fonction

```
# let foo n =
    for i = 1 to 10 do
      i * n;
    done;;

Characters 40-46:
      i * n;
      ^^^^^^
Warning 10: this expression should have type unit.
val foo : int -> unit = <fun>
```

est acceptée, mais ne renvoie rien, ce qui n'est probablement pas ce que l'on souhaite !

Cette fois encore, si l'expression dans la boucle n'a pas d'effet (affichage, modification de la mémoire...), cette structure de contrôle n'a guère d'intérêt.

Il n'existe pas de structure permettant de choisir le pas lors de l'itération, ou d'itérer sur autre chose que des entiers. On dispose cependant du mot-clé **downto** afin de *décompter* au lieu de compter :

```
for nom = expression_1 downto expression_2 do sequence done
```

2. Bien évidemment, rien n'empêche de le faire avec une écriture purement fonctionnelle et une récursion, nous y reviendrons. On dispose simplement ici d'une *autre* manière d'exprimer une telle opération.

3. Y compris la dernière, une boucle **for** renvoie bien toujours `()` et non le résultat de l'expression de la dernière itération !

### 1.4 Boucles conditionnelles (`while`)

Parfois, le nombre d'itérations à effectuer n'est pas connu à l'avance, et l'on souhaite effectuer une tâche tant qu'une expression est vraie. Pour ce faire, on dispose de la structure de contrôle suivante :

```
while expression do sequence done
```

Avec cette structure, `sequence` sera évaluée autant de fois que nécessaire, tant que `expression` sera vraie. Par exemple :

```
while read_line () <> "Au revoir" do
  print_string "Dites m'en plus !";
  print_newline ();
done;;
```

Encore plus que dans les exemples précédents, on a besoin ici que quelque chose se passe pour que `expression` change après un certain nombre d'itérations, sinon on sera bloqués dans une boucle infinie ! Il devient vraiment indispensable que `sequence` puisse agir sur le contenu de la mémoire.

Il n'est en effet pas possible d'utiliser une définition `let ... = ...` à l'intérieur de la boucle. Tout au plus peut on utiliser une définition locale `let ... = ... in ...`, mais cette définition sera oubliée dès la fin de l'itération dans laquelle elle apparaît.

## 2 Références

### 2.1 Les références

On l'a vu, la programmation impérative n'a de sens que si l'on est capable d'agir sur le contenu de la mémoire, ce qui ne peut être fait avec des définitions. Plutôt que d'associer un nom à un objet (valeur, chaîne, arbre...), il est possible, grâce au mot-clé `ref`, de créer une *référence* vers un objet.

En écrivant par exemple

```
# let a = ref 2.2;;
val a : float ref = {contents = 2.2}
```

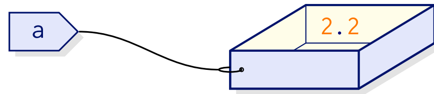
on demande à Caml d'associer le nom « `a` » à une référence vers un flottant (ainsi que l'indique le `float` du type « `float ref` »), ce flottant étant initialement 2.2.

Dans un premier temps[4], on peut imaginer le nom associé à une « boîte » contenant

4. Nous verrons un peu plus tard que cette image peut poser quelques soucis dans certains cas.

l'entier. D'ailleurs, la réponse de Caml suggère bien que l'on manipule une « boîte » avec un contenu. Les noms sont ainsi associés aux « boîtes » plutôt qu'à leur contenu.

La définition `let a = ref 2.2` conduit donc à une situation de la sorte, où l'on voit que le nom a est bien associé (de façon permanente) à la boîte et non au flottant qu'elle contient :




On ne peut utiliser directement une référence comme s'il s'agissait de l'objet qu'elle contient :

```
# a *. 2.0;;

Characters 2-3:
  a *. 2.0;;
  ^
Error: This expression has type float ref
       but an expression was expected of type float
```

Il faut donc préalablement extraire le flottant. Pour ce faire, on fait précéder le nom d'un point d'exclamation :

```
# !a;;
- : float = 2.2

# !a *. 2.0;;
- : float = 4.4
```

Pour modifier le contenu de la case mémoire, on utilise l'opérateur « := » :

```
# a := 3.7;;
- : unit = ()

# !a;;
- : float = 3.7

# a;;
- : float ref = {contents = 3.7}
```

Le contenu de la case mémoire a bien été changé, mais pas la définition de a.

## 2.2 Utilisation

Les références permettent, entre autres choses, de créer des accumulateurs et des compteurs, des objets que l'on retrouve trèès fréquemment dans la programmation impérative. Il est par exemple très simple d'écrire, en style impératif, une fonction factorielle :

```
# let fact n =
    let res = ref 1 in
      for i=2 to n do
        res := !res * i
      done;
      !res;;

val fact : int -> int = <fun>
```

Pour incrémenter l'entier désigné par une référence x, il suffit en principe d'écrire

```
x := !x + 1
```

Comme il s'agit d'une opération courante en programmation impérative, OCaml fournit une fonction `incr` prenant en argument une référence vers un entier et réalisant une telle incrémentation. « `incr x` » est donc équivalent à « `x := !x + 1` ». La fonction `decr`, quant à elle, décrémente d'une unité le contenu d'une référence vers un entier.

Il est possible de créer des références vers des objets de n'importe quel type. Ainsi, dans la fonction suivante, qui compte le nombre de zéros d'une liste d'entiers, on utilise ainsi deux références, l'une, `rest`, recueillant la liste des données restant à traiter, la seconde, `count`, le nombre de zéros déjà identifiés dans la liste :

```
# let count_zeros lst =
    let rest = ref lst          (* éléments restant à examiner   *)
    and count = ref 0 in        (* un compteur de zéros          *)
      while !rest <> [] do
        if List.hd !rest = 0
          then incr count;      (* else () implicite             *)
        rest := List.tl !rest   (* le premier élément est traité *)
      done;
      !count;;                  (* on renvoie le contenu         *)

val count_zeros : int list -> int = <fun>
```

La fonction précédente illustre la façon dont on traite généralement les listes dans un style impératif. Rappelons que `List.tl` ne crée pas une copie de la liste, et est bien une opération en $O(1)$, donc cette fonction n'est pas inefficace.

De même, on peut très bien avoir des références de fonctions, par exemple ici des fonctions des entiers vers les entiers :

```
# let funct = ref abs;;
val foo : (int -> int) ref = {contents = <fun>}

# !funct (-37);;
- : int = 37
```

On peut alors y associer tout objet de type[5] `int -> int` :

```
# funct := fun x -> x*x*x;;
- : unit = ()

# funct := min 0;;
- : unit = ()
```

Une référence peut même contenir une référence :

```
# let b = ref (ref 37);;
val b : int ref ref = {contents = {contents = 0}}
```

Pour accéder à l'entier, il faut alors utiliser deux fois un déréférencement :

```
# ! !b;;
- : int = 37
```

Les références sont en revanche toujours associées à un type bien particulier, et il n'est pas possible de leur associer un objet d'un autre type :

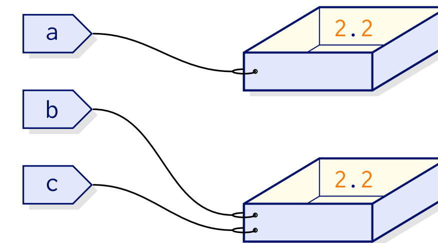
```
# let a = ref 2;;
val a : int ref = {contents = 2}

# a := 4.0;;

Characters 6-9:
  a := 4.0;;
       ^^^
Error: This expression has type float
       but an expression was expected of type int
```

5. En fait, il est même possible d'y associer des fonctions de signature `int -> 'a`, `'a -> 'a`, etc. Cependant, l'objet qui se retrouvera référencé sera un objet de type `int -> int`, qui le restera une fois déréférencé, et qui donc ne correspondra plus à l'objet « original ».

## 2.3 Égalité, identité

Deux noms peuvent aussi bien désigner la même « boîte », comme b et c dans l'exemple ci-dessous :

```
# let a = ref 2.2 and b = ref 2.2;;
val a : float ref = {contents = 2.2}
val b : float ref = {contents = 2.2}

# let c = b;;
val c : float ref = {contents = 2.2}
```




Modifier le contenu de la « boîte » désignée par b aura donc des conséquences sur c mais pas sur a :

```
# b := 3.7;;
- : unit = ()

# a;;
- : float ref = {contents = 2.2}

# c;;
- : float ref = {contents = 3.7}
```

Il est dès lors naturel de se poser, dans ce genre de situation, la question du fonctionnement de l'opérateur d'égalité[6] `=`. En Caml, celui-ci teste une *égalité de valeurs* (parfois qualifiée d'*égalité structurelle*), autrement dit l'opérateur regarde si les *contenus* sont égaux :

```
# a = b;;
- : bool = true

# b = c;;
- : bool = true
```

6. qui n'est pas sans rappeler des difficultés similaires concernant l'égalité/l'identité de deux listes en Python

Notons qu'il n'est, naturellement, possible que de comparer deux éléments de même type, et qu'une référence vers un flottant n'est pas comparable à un flottant :

```
# a = 3.7;;
Characters 6-9:
  a = 3.7;;
      ^^^
Error: This expression has type float
       but an expression was expected of type float ref
```

Pour tester l'*identité* (ou *égalité physique*), on utilisera l'opérateur == :

```
# a == b;;
- : bool = false

# b == c;;
- : bool = true
```

L'opérateur != teste lui la « *non-identité* »[7].

```
# a != b;;
- : bool = true

# b != c;;
- : bool = false
```

Les opérateurs == et != font référence à la manière dont les objets sont rangés en mémoire, et **leur usage est à réserver aux objets mutables**. Leur comportement sur des objets immutables peut être imprévisible (et varier d'un compilateur à l'autre) :

```
# 2.5 == 2.5;;
- : bool = false
```

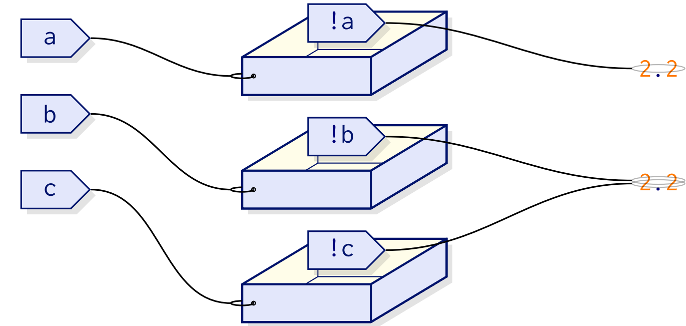
### 2.4 Une dernière remarque pour clore

Si cette image de « boîte » est généralement suffisante, elle peut parfois montrer ses limites. En effet, il est possible de créer des références distinctes vers un même objet (qui se trouverait alors simultanément dans deux « boîtes »!), comme ci-dessous :

```
# let a = ref 2.2 and b = ref 2.2
# let c = ref !b
```

7. On rappelle que c'est l'opérateur <> qui teste si deux valeurs ne sont pas égales.

En réalité, de la même façon que le compilateur associe usuellement aux noms les adresses où ont stockés en mémoire les objets qu'ils désignent, les références contiennent également des adresses d'objets en mémoire. Une meilleure image serait donc d'imaginer que l'on place dans chaque boîte non pas les objets eux-même, mais une étiquette permettant de les retrouver, comme illustré ci-dessous :




Toutefois, tant que les objets auxquels on fait référence sont immutables, et ce sera le cas dans la très grande majorité des situations, cette distinction n'a pas d'importance.

## 3 Objets Caml avec mutabilité

### 3.1 Cas du type « enregistrement »

Supposons que l'on souhaite manipuler un annuaire en Caml, regroupant le nom et le numéro de téléphone de différentes personnes. On peut par exemple écrire un type « enregistrement » associant un nom et un numéro de téléphone[8] :

```
type coord = { name: string ; number: string };;
```

On peut ensuite définir un annuaire comme une liste de tels éléments[9] :

```
let phonebook = [ { name = "Dupont" ; number = "0123456789" } ;
                  { name = "Durand" ; number = "0246813579" } ;
                  { name = "Martin" ; number = "0918273645" } ];;
```

L'ennui, c'est qu'il n'est pas possible de modifier un numéro si la personne en change, sans modifier la liste de façon à retirer l'élément devenu incorrect pour le remplacer par un nouveau.

8. On aurait pu mémoriser le numéro sous la forme d'un entier, mais si la version de Caml utilise des entiers 32 bits, ils ne permettront pas de mémoriser n'importe quel numéro à dix chiffres, et on perdrait les 0 de tête.
9. Nous verrons dans le chapitre suivant une meilleure solution pour définir un annuaire.

Ce que l'on pourrait écrire, dans un style fonctionnel, par[10] :

```
# let rec update name new_number = function
    | h::t when h.name = name
      -> { name = name ; number = new_number }
           :: update name new_number t
    | h::t -> h :: update name new_number t
    | [] -> [];;

val update : string -> string -> coord list -> coord list = <fun>
```

Cela crée un *nouvel* annuaire, intégrant la correction. L'argument n'est lui pas modifié.

On peut donc par exemple utiliser la fonction `modifie` de la sorte :

```
# let new_phonebook = update "Durand" "0000012345" phonebook ;;
val new phonebook : coord list =
  [{name = "Dupont"; number = "0123456789"};
   {name = "Durand"; number = "0000012345"};
   {name = "Martin"; number = "0918273645"}]
```

On peut préférer *modifier* un annuaire existant, et pour ce faire utiliser des références pour le numéro, au prix d'un changement dans la déclaration de l'annuaire :

```
type coord = { name: string ; number: string ref };;

let phonebook = [ { name = "Dupont" ; number = ref "0123456789" } ;
                  { name = "Durand" ; number = ref "0246813579" } ;
                  { name = "Martin" ; number = ref "0918273645" } ];;
```

La fonction de modification peut alors s'écrire :

```
# let rec update name new_number = function
    | h::t when h.name = name
      -> h.number := new_number;
         update name new_number t
    | h::t -> update name new_number t
    | [] -> ();;

val update : string -> string -> coord list -> unit = <fun>
```

10. Notons que si l'on trouve le nom recherché dans l'annuaire, on poursuit la recherche, et si le nom apparaît plusieurs fois dans l'annuaire, *tous* les numéros seront mis à jour.

Le résultat de la fonction est à présent de type `unit`, car on ne construit plus un nouvel annuaire, on se contente de modifier l'existant, en écrivant :

```
# modifie "Durand" "0000056789" annuaire;;
- : unit = ()

# annuaire;;
- : coord list =
[{nom = "Dupont"; numéro = {contents = "0123456789"}};
 {nom = "Durand"; numéro = {contents = "0000056789"}};
 {nom = "Martin"; numéro = {contents = "0918273645"}}]
```

On peut aussi adopter un style plus impératif :

```
# let update name new_number phonebook =
    let rest = ref phonebook in
      while !rest <> [] do
        let coord = List.hd !rest in
          if coord.name = name
            then coord.number := new_number;
          rest := List.tl !rest
      done;;

val update : string -> string -> coord list -> unit = <fun>
```

L'inconvénient de cette approche est que cela change la façon de déclarer l'annuaire (avec des `ref`) et de l'utiliser (avec des `!`), ce qui peut être parfois génant. Il existe cependant une autre façon de procéder : Caml nous offre la possibilité de déclarer un champ du type enregistrement comme étant *mutable* :

```
type coord = { name: string ; mutable number: string };;

let phonebook = [ { name = "Dupont" ; number = "0123456789" } ;
                  { name = "Durand" ; number = "0246813579" } ;
                  { name = "Martin" ; number = "0918273645" } ];;
```

On peut alors modifier l'élément mutable avec l'opérateur `<-` :

```
# (List.hd phonebook).number <- "9876543210";;
- : unit = ()

# List.hd phonebook;;
- : coord = {name = "Dupont"; number = "9876543210"}
```

On écrit alors notre fonction de modification par exemple de la façon suivante :

```
# let rec update name new_number = function
    | h::t when h.name = name
        -> t.number <- new_number;
           update name new_number t
    | h::t -> update name new_number t
    | [] -> ();;

val update : string -> string -> coord list -> unit = <fun>
```

Pour ceux qui se demanderaient pourquoi l'on a créé un opérateur supplémentaire `<-` au lieu d'utiliser `:=`, dans le cas où l'on définit un objet de la sorte

```
type foo = { mutable elem = int ref };;
```

il fallait bien pouvoir distinguer les deux opérations qui sont toutes les deux possibles ici !

## 3.2 Retour sur les références

Les références ne sont en fait qu'un cas particulier de type enregistrement. Il est parfaitement possible de les recréer en écrivant [11] :

```
# type 'a ref = { mutable contents: 'a };;

# let ref x = { contents = x };;
val ref : 'a -> 'a ref = <fun>

# let (!) = function { contents=x } -> x;;
val ( ! ) : 'a ref -> 'a = <fun>
```

Le comportement sera identique à celui des références que nous avons vues précédemment. D'ailleurs, dans le cas des références fournies par OCaml, il est parfaitement permis, si a désigne une référence, d'écrire « `a.contents` » et « `a.contents <- valeur` »...

## 3.3 Un autre objet mutable : les tableaux (`'a array`)

Certains types proposés par Caml sont « naturellement » mutables. Les chaînes de caractères l'ont été (il était possible de « muter » un caractère d'une chaîne), mais ne le sont plus [12] dans les dernières versions de OCaml.

11. Cela explique d'ailleurs l'affichage « normal » des références qui fait mention d'un « `contents` » !
12. Il existe un type `bytes`, très semblable aux chaînes de caractères, qui lui est mutable.

Les listes sont immutables, ce qui rend, on l'a vu, leur utilisation dans un style impératif délicat. On dispse donc d'un autre conteneur, mutable, que l'on utilisera souvent dans un style impératif : les *tableaux* (de type `'a array`). Ce sont des objets qui peuvent contenir un nombre *prédéterminé* d'éléments *de même type*.

On peut les définir explicitement en plaçant différents éléments (impérativement tous de même type), séparés par des points virgules, entre `[|` et `|]` :

```
# let tableau = [| 1.2; 2.3; 3.4 |];;
val tableau : float array = [|1.2; 2.3; 3.4|]
```

On peut également créer un tableau grâce à la fonction `Array.make`, en précisant la taille et l'élément à placer dans chaque case :

```
# Array.make 6 0.0;;
- : float array = [|0.; 0.; 0.; 0.; 0.; 0.|]

# Array.make 3 "Hello";;
- : string array = [|"Hello"; "Hello"; "Hello"|]
```

On peut obtenir la taille d'un tableau avec `Array.length`, et accéder à un élément en indiquant, entre parenthèses précédées d'un point, l'indice de l'élément souhaité :

```
# Array.length tableau;;
- : int = 3

# tableau.(1);;
- : float = 2.3
```

Au contraire de ce qui se passe avec les listes, ces opérations sont toutes deux effectuées en temps constant ($O(1)$).

Par ailleurs, les tableaux étant des objets mutables, il est possible de modifier un élément du tableau avec `<-` :

```
# tableau.(1) <- 10.23;;
- : unit = ()

# tableau;;
- : float array = [|1.2; 10.23; 3.4|]
```

Il existe en OCaml de nombreuses fonctions destinées à la manipulation de tableaux (telles que `Array.copy`, `Array.sub`, `Array.iter`, `Array.map`, `Array.mem`, `Array.to_list`, `Array.of_list`, `Array.sort`...) dont la liste et le fonctionnement sont résumés dans la documentation du langage, et que nous découvrirons en fonction de nos besoins.

Les listes et les tableaux répondent à des besoins différents, comme nous le verrons. Il est très facile d'obtenir une liste résultant de l'ajout ou de la suppression d'un élément en tête d'une liste existante (en O(1)), mais le coût pour accéder à un élément au milieu de la liste est élevé (en $O(n)$).

Par ailleurs, le contenu d'une liste est immutable. À l'inverse, les tableaux sont des objets mutables, mais ont une taille fixe (la changer nécessite de recopier le tableau, avec un coût en $O(n)$).

Ainsi, en fonction des besoins de l'algorithme, on préférera donc l'une ou l'autre de ces structures.

### 3.4 Tableaux bidimensionnels

Pour représenter un tableau en deux dimensions, il n'existe pas de type particulier, mais comme on peut définir des tableaux de n'importe quel type, on peut définir des tableaux de tableaux. Attention toutefois, si l'on souhaite construire une matrice nulle de trois lignes et deux colonnes, **on ne peut écrire** :

```
# let matrice = Array.make 3 (Array.make 2 0.0);; (* incorrect *)
- : float array array = [|[|0.; 0.|]; [|0.; 0.|]; [|0.; 0.|]|]
```

Même si le résultat semble satisfaisant, on a créé ici un tableau *qui contient trois fois la même ligne*[13] ! Modifier un élément sur une ligne quelconque aurait un effet sur toutes les autres, ce qui n'est a priori pas ce que l'on cherche...

Pour définir une matrice nulle de trois lignes et deux colonnes, on pourra en revanche écrire :

```
# let matrice = Array.make 3 [| |];;
val matrice : '_a array array = [|[||]; [||]; [||]|]

# for i=0 to 2 do matrice.(i) <- Array.make 2 0.0 done;;
- : unit = ()

# matrice;;
- : float array array = [|[|0.; 0.|]; [|0.; 0.|]; [|0.; 0.|]|]
```

Caml fournit cependant fort obligeamment un raccourci pour effectuer cette construction, `Array.make_matrix` :

```
# Array.make_matrix 3 2 0.0;;
- : float array array = [|[|0.; 0.|]; [|0.; 0.|]; [|0.; 0.|]|]
```

13. Il s'agit exactement du même problème que lorsque l'on écrit `[ [0.0] * 2 ] * 3` en Python.

On fait référence la ligne d'indice $i$ par :

```
# matrice.(1);;
- : float array = [|0.; 0.|]
```

Et donc à l'élément situé sur la ligne d'indice $i$ dans la colonne d'indice $j$ par :

```
# matrice.(1).(0);;
- : float = 0.
```

Remarquons enfin que rien ne garantit, lorsque l'on a un `'a array array`, que chacune des « lignes » ait la même taille, et qu'il peut très bien ne pas s'agir d'un tableau bidimensionnel dans le sens usuel du terme !

## Exercices

### Ex. 7.1 – Fonction mystérieuse

Déterminer ce que fait la fonction suivante et sa signature :

```
let foo x y =
  x := !x + !y;
  y := !x - !y;
  x := !x - !y ;;
```

### Ex. 7.2 – Liste de couples

Proposer une fonction `couples` de signature `int -> (int * int) list` prenant en argument un entier $n > 0$ et renvoyant la liste de tous les couples d'entiers $(x, y)$ vérifiant $1 \leqslant x \leqslant y \leqslant n$. On pourra envisager une version impérative et une version fonctionnelle.

### Ex. 7.3 – Fonctionnelles

Proposer des équivalents [14] à `List.map` et `List.fold_left` pour des `'a array`.

### Ex. 7.4 – Plateaux

Proposer une fonction `longest_stat` de signature `'a array -> int` prenant en argument un tableau et renvoyant la longueur du plus long sous-tableau ne contenant des valeurs toutes égales.

### Ex. 7.5 – Symétrie

Proposer une fonction `is_symmetric` de signature `'a array -> int` prenant en argument un tableau et renvoyant un booléen indiquant si les valeurs contenues dans le tableau, considérées de gauche à droite et de droite à gauche, sont égales.

### Ex. 7.6 – Miroir

Proposer une fonction `reverse` de signature `'a array -> unit` prenant en argument un tableau et inversant l'ordre de ses éléments en place (la valeur dans la première case se retrouve en dernière place, la seconde en avant-dernière, et ainsi de suite).

14. Ces fonctions existent, sous le nom `Array.map` et `Array.fold_left`.

### Ex. 7.7 – Différences cumulées

On suppose disposer d'une fonction `cumsum` de signature `int array -> unit` calculant, en place, les sommes cumulées d'un tableau.

Proposer une fonction `invcum` de signature `int array -> unit` effectuant l'opération inverse : `cumsum (invcum t)` et `invcum (cumsum t)` doivent tous les deux ne pas avoir d'effet sur le tableau `t`.

### Ex. 7.8 – Réciproque

Écrire une fonction `invert` de signature `int array -> int array` qui renvoie la réciproque d'une bijection $\phi$ décrite par son tableau de valeurs (dans la case d'index $i$ du tableau, on a rangé $\phi(i)$)

### Ex. 7.9 – Mélange de Ficher-Yates

Le mélange de Ficher-Yates, également appelé mélange de Knuth, a pour but d'obtenir une permutation aléatoire des éléments d'un tableau, de sorte que toutes les permutations peuvent être obtenues de façon équiprobable. Il fonctionne de la façon suivante sur un tableau de taille $n$ :

- on choisit un entier $k$ aléatoire dans $[\![0\,..\,n-1]\!]$, et on échange le contenu des cases 0 et $k$ du tableau (si $k = 0$, on peut évidemment ne rien faire) ;
- on choisit un entier $k$ aléatoire dans $[\![1\,..\,n-1]\!]$, et on échange le contenu des cases 1 et $k$ du tableau ;
- on poursuit de suite, pour $i$ allant de 2 à $n-2$, en choisissant un entier $k$ aléatoire dans $[\![i\,..\,n-1]\!]$, et on échange le contenu des cases $i$ et $k$ du tableau.

**1.** Justifier que chaque élément a la même probabilité de se trouver dans la case d'index 0 à l'issue de l'algorithme.

**2.** Faire de même pour l'élément se trouvant dans la case d'index 1 à l'issue de l'algorithme.

**3.** Justifier que l'on peut bien obtenir n'importe quelle permutation, et que la permutation obtenue est choisie aléatoirement avec une loi uniforme parmi toutes les permutations possibles.

**4.** Proposer une fonction `shuffle` de signature `'a array -> unit` réalisant un mélange de Ficher-Yates. On dispose d'une fonction `Random.int` de signature `int -> int` prenant en argument un entier $k$ et renvoyant un entier choisi aléatoirement dans $[\![0\,..\,k-1]\!]$ avec une distribution uniforme.

## Ex. 7.10 – Tableaux auto-référents

Un tableau t est dit *auto-référent* si t contient t.(i) fois la valeur i pour tout i dans $[\![0\,..\,n-1]\!]$ où $n$ est la taille du tableau t.

**1.** Proposer une fonction vérifiant si un tableau est auto-référent.

On se propose d'écrire une fonction construisant un tableau auto-référent de taille $n$. On propose l'algorithme suivant : on part d'un tableau contenant $n$ zéros, et tant que le tableau n'est pas auto-référent, pour tout $i \in [\![0\,..\,n-1]\!]$, on compte le nombre de i et on place le résultat dans t.(i).

**2.** Proposer une implémentation d'une telle fonction.

**3.** Que penser de la correction de cette fonction ? Voyez-vous d'autres solutions ?

## Ex. 7.11 – Master Mind

On considère un jeu basé sur un ensemble de $p$ couleurs numérotés de 1 à $p$. Le but est de deviner une séquence cachée de $n$ de ces couleurs [15], par exemple pour $n = 5$ et $p = 6$, `[|1; 5; 6; 1; 2|]`.

Le jeu procède par hypothèses, qui sont des séquences de même taille que la séquence recherchée, avec les mêmes couleurs possibles. On déterminer pour chaque hypothèse le nombre d'éléments bien placés, et le nombre d'éléments mal placés. Par exemple pour l'hypothèse `[|4; 2; 1; 1; 2|]`, il y a 2 éléments bien placés et deux éléments mal placés (même si le 2 n'apparaît qu'une fois dans la bonne solution, on compte quand même un 2 bien placé et un 2 mal placé).

**1.** Proposer une fonction count de signature `int array -> int array -> int * int` prenant en argument la solution et une hypothèse et renvoyant le couple correspondant.

**2.** Proposer une fonction all de signature `int -> int -> int array list` prenant en argument les entiers $n$ et $p$ et renvoyant une liste de toutes les combinaisons possibles (dans un ordre quelconque).

**3.** Écrire une fonction filter_options de signature `int array list -> int array -> (int * int) -> int array list` prenant en argument une liste de combinaisons possibles (possibilités), ue hypothèse et le couple correspondant et renvoyant la liste de possibilité compatibles avec cette hypothèse.

**4.** Comment fabriquer un automate jouant tout seul à deviner la séquence cachée ?

15. Le principe est similaire aux jeux de type « Wordle », populaires en ligne ces dernières années, excepté que l'on remplace les mots par des séquences de couleurs.

## Ex. 7.12 – Algorithme de Heap

L'algorithme de Heap a été proposé par B. R. Heap en 1963. Il vise à construire successivement les $n!$ permutations des $n$ éléments d'un tableau en minimisant le nombre d'échanges à effectuer, soit $n!-1$ échanges au total [16].

Cet algorithme fonctionne de manière récursive : il génère toutes les permutations de $n-1$ premiers éléments sans toucher au dernier, puis échange le dernier élément avec l'un des $n-1$ premiers éléments et génère à nouveau toutes les permutations des $n-1$ premiers éléments, et ainsi de suite. La difficulté consiste à savoir avec quel élément échanger le dernier élément à chaque fois qu'on a à le faire.

Pour un tableau de taille $n$, en dehors de la génération des permutations de taille $n-1$, il y a donc $n-1$ échanges à faire entre le dernier élément et un autre élément du tableau. Numérotons ces échanges par un entier $i$ allant de 0 à $n-2$.

- si $i$ est pair, on échange le dernier élément avec l'élément d'index $i$ ;
- si $i$ est impair, on échange le dernier élément avec l'élément d'index 0.

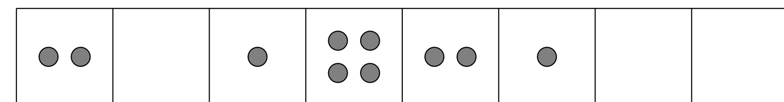
On peut montrer que cette stratégie permet bien d'obtenir l'ensemble des $n!$ permutations du tableau.

**1.** Proposer une fonction all_perms de signature `'a array -> ('a array -> unit) -> unit` prenant en argument un tableau de taille $n$ et une fonction $f$ prenant en argument un tableau, et appelant la fonction $f$ successivement sur les $n!$ permutations possibles du tableau en utilisant l'algorithme de Heap.

## Ex. 7.13 – Awele en solitaire

On s'intéresse à un jeu de solitaire, vaguement inspiré de l'awele, composé de billes et d'un ensemble de réceptacles, alignés, pouvant accueillir celles-ci. Un « coup » consiste à choisir un des réceptacles, à prendre l'ensemble des billes qu'il contient, et à en placer une dans chacun des réceptacles situés à sa gauche. Le coup est « valide » si et seulement si à l'issue de ces replacements, on reste avec une unique bille en main, qui est écartée.

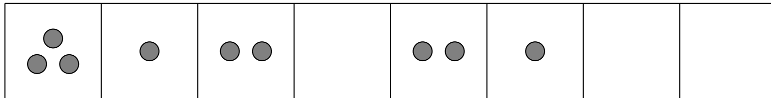
Par exemple, à un instant donné, le jeu peut se présenter de la façon suivante :



Dans cette situation, il est possible déplacer les billes du quatrième réceptacle en partant de la gauche, en plaçant une bille dans chacun des trois premiers réceptacles, et en écartant la quatrième et dernière bille (il s'agit par ailleurs du seul coup valide dans la situation présente).

16. C'est l'algorithme avec le minimum d'échanges possible, même si l'on peut trouver des algorithmes légèrement plus rapides en raison de considérations diverses telle que la gestion du cache du processeur.

Après ce coup, la situation est alors la suivante :



La partie est « gagnée » si l'on parvient à éliminer toutes les billes par une succession de coups valides. L'état précédent ne permet plus de coups valides, aussi la partie est-elle perdue dans le cas proposé.

L'état du jeu sera représenté par un tableau d'entiers (`int array`), indiquant le nombre de billes dans chaque cases (de gauche à droite). Par exemple, les deux états présentés ci-dessus correspondent respectivement aux tableaux `[|2; 0; 1; 4; 2; 1; 0; 0|]` et `[|3; 1; 2; 0; 2; 1; 0; 0|]`.

**1.** Proposer une fonction `is_valid` de signature `int array -> int -> bool` prenant en argument un tableau représentant un état du jeu et un entier `k` désignant une case (0 désigne la case la plus à gauche, 1 la seconde case en partant de la gauche, etc.) et renvoyant un booléen indiquant si le coup consistant à déplacer les pierres de la case indiquée est valide.

**2.** Proposer une fonction `next` de signature `int array -> int -> unit` prenant en argument un tableau et un entier `k` désignant une case, et modifiant le contenu du tableau pour qu'il représente l'état du jeu après le déplacement des billes de la case `k`. On déclenchera une erreur si le coup n'est pas valide.

**3.** Justifier que s'il y a plusieurs coups possibles, un seul coup au plus peut amener vers une victoire, et préciser comment on le détermine.

**4.** En déduire une fonction `possible` de signature `int array -> bool` prenant un état donné et renvoyant un booléen indiquant s'il existe une succession de coups permettant de gagner la partie. Il est permis de modifier le contenu du tableau lors de l'appel.

**5.** Proposer une fonction `solution` de signature `int array -> int list` qui renvoie une liste de « coups » permettant de gagner la partie si c'est possible, et déclenchera une erreur si c'est impossible. Il est permis de modifier le contenu du tableau lors de l'appel. Par exemple, pour le tableau `[|1; 1; 1; 3; 5; 0; 0; 0|]`, la fonction renverra la liste `[0; 4; 0; 1; 0; 3; 0; 2; 0; 1; 0]`.

**6.** Proposer une fonction `problem` de signature `int -> int array` prenant en argument un nombre `n` de billes et renvoyant un tableau, représentant un état à `n` billes pour lequelle une solution existe.

## Ex. 7.14 – Jeu du baguenaudier

On considère un jeu se jouant sur un tableau de $n$ cases, chaque case pouvant chacune contenir au plus un jeton. Les règles sont les suivantes :

- on peut ajouter ou retirer un jeton dans la case 0 ;
- on peut ajouter ou retirer un jeton dans la case $i$ si et seulement si il y a un jeton dans la case $i-1$ et aucun jeton dans les cases plus à gauche (d'index 0 à $i-2$ inclus si elles existent).

C'est une modélisation d'un casse-tête d'origine probablement chinoise constitué d'anneaux et d'une tige qu'il faut libérer.

L'état du jeu est représenté par un `bool Array`, un `true` indiquant la présence d'un jeton. Une séquence de coup représenté par une liste d'entiers entre 0 et $n-1$ (pour un entier $i$, s'il y a un jeton dans la case $i$ on l'enlève, et sinon on ajoute un jeton dans cette même case).

**1.** Proposer une fonction prenant en argument un tableau et une liste d'entiers et exécutant les coups indiqués.

**2.** Montrer qu'il est possible de vider un tableau initialement plein, ou de remplir un tableau initialement vide, et programmer deux fonctions prenant en argument la taille $n$ et produisant une liste de coups permettant d'effectuer ces opérations.

**3.** En déduire une fonction indiquant une solution permettant de vider un tableau initial quelconque passé en paramètre.

# Table des matières


**Introduction au langage C** **1**
Du programme au processeur . . . 1
Des instructions pour le processeur . . . 1
Documenter le code pour les humains . . . 1
Langages de programmation . . . 2
Compilation et interprétation . . . 2
Naissance du C, forces et faiblesses . . . 3
Un premier aperçu d'un fichier source C . . . 4
Déclarations . . . 5
Principe . . . 5
Devenir des variables dans les fichiers binaires . . . 7
Utilisation des varibles . . . 7
Affectation . . . 7
Affectations et conversions implicites . . . 8
Expressions . . . 8
Instructions et expressions . . . 8
Opérateurs arithmétiques . . . 9
Opérateurs de comparaison . . . 9
Opérateurs logiques . . . 10
Précédence . . . 10

**Structures de contrôle en C** **11**
Structure conditionnelles . . . 11
Exécution conditionnelle avec « |if| » . . . 11
Précision sur les tests en langage C . . . 11
Alternative avec « |else| » . . . 12
Grouper les instructions en « blocs » . . . 12
Tests en cascade . . . 13
Blocs d'instructions et portée des variables . . . 14
Principe . . . 14
Un peu plus loin . . . 14
Occultation de variables . . . 15
Boucles conditionnelles . . . 15
Boucles « |while| » . . . 15
Quelques exemples concrets . . . 16
Variation sur le thème . . . 17
Écrire des boucles correctes . . . 18
De l'importance et du bon usage des commentaires . . . 18
Invariants . . . 18
Assertions . . . 19
Correction et terminaison . . . 19
Itérations . . . 20
Principe . . . 20
Particularités des boucles |for| en C . . . 21
Raccourcis d'écriture . . . 21
Interrompre les itérations . . . 22
Séquences d'expressions . . . 22
Omission d'éléments dans un |for| . . . 22
Fonctions . . . 23
Définitions de fonctions . . . 23
Appels de fonction . . . 24
Fonctions et procédures . . . 24
Gestion de la mémoire . . . 25
Variables « globales » et effets de bord . . . 26
Déclarations . . . 27
Découpage du fichier source et liaison . . . 28

**Les tableaux en C** **35**
Création et manipulation de tableaux . . . 35
Déclaration . . . 35
Initialisation . . . 36
Manipulation des éléments . . . 36
Un mot sur le comportement indéfini en C . . . 37
Tableaux, affectations, expressions . . . 38
Tableaux et fonctions . . . 39
Algorithmes élémentaires sur les tableaux . . . 39
Somme des éléments . . . 39
Sommes cumulées . . . 39
Détermination de la valeur maximale . . . 39
Localisation d'un élément particulier . . . 41
Vérification de propriétés . . . 42
Plus longue séquence . . . 43
Quelques exemples plus élaborés . . . 45
Recherche d'un gagnant d'un vote à la majorité . . . 45
Problème du sous-tableau maximal . . . 47
Premiers tris et notion de complexité . . . 50
Le tri par sélection . . . 50

Complexité algorithmique ... 51
Le tri « bulle » ... 54
Le tri « insertion » ... 56
Tableaux multidimensionnels ... 57
Principe ... 57
Tableaux C à plusieurs dimensions ... 58
Représenter des images ... 59

**Adresses et pointeurs** **65**
Adressage ... 65
Adresse d'une variable ... 65
Utilisation de l'adresse ... 66
Retour sur les déclarations ... 67
Validité des adresses ... 67
Pointeurs NULL ... 68
Pointeurs et mémoire adressable ... 68
Allocation dynamique ... 68
Tableaux ... 70
Allocation dynamique de tableaux ... 70
Différences avec les tableaux ... 70
Tableaux et fonctions ... 71
Effets de bords des fonctions ... 71
Tableaux et retours de fonctions ... 72
Tableaux et opérations sur les pointeurs ... 73
Quelques exemples concrets ... 74
Recherche dichotomique dans un tableau trié ... 74
Tri rapide ... 76
Partitionner un tableau ... 78
Retour sur les tableaux multidimensionnels ... 82
Pointeurs vers les tableaux multidimensionnels ... 82
Tableaux multidimensionnels et fonctions ... 83
Tableaux de pointeurs et tableaux « Iliffes » ... 83
Pointeurs de fonctions ... 84
Principe et déclaration ... 84
Fonctions comme paramètres ... 85

**Introduction au langage OCaml** **91**
Présentation de la famille Caml ... 91
Philosophie du langage ... 91
Compilateurs et interpréteurs OCaml ... 91
Premiers pas ... 92
Calculs et expressions ... 92
Typage fort ... 94
Définitions ... 95
Définitions globales ... 95
Différences avec les variables en C ... 95
Définitions multiples ... 96
Définitions locales ... 97
Les fonctions ... 98
Fonctions avec un unique argument ... 98
Le mot-clé « |function| » ... 99
Arguments multiples ... 99
Signature de la fonction ... 100
Le type |unit| ... 101
Filtrage par motif ... 102
Motifs gardés ... 103
Fonctions récursives ... 104
Expressions avancées ... 105
Expressions conditionnelles ... 105
Filtrage ... 106
Couples ... 107
Principe ... 107
Couples et filtrages ... 108

**Structures de données** **115**
Les types construits en OCaml ... 115
Définir ses propres types ... 115
Types « somme » (énumérations, unions) ... 115
Types « produit » (enregistrements) ... 117
Listes chaînées ... 118
Les listes en informatique ... 118
Principe des listes chaînées ... 118
Les listes OCaml ... 121
Création et manipulation ... 121
Immutabilité des listes ... 122
Autres fonctions sur les listes ... 123
Écrire des fonctions sur les listes ... 123
Coût en temps des opérations sur les listes ... 124
Retour sur des algorithmes classiques ... 125
Fonctionnelles agissant sur les listes ... 128
Listes et prédicats booléens ... 132
Créer des types complexes en C ... 133
Déclaration de types ... 133
Enregistrements ... 133
Énumérations, unions ... 135
Les listes en langage C ... 136

Création d'un type récursif . . . 136
Identifier l'extrémité droite de la liste . . . 136
Mémoriser une liste . . . 137
Manipuler une liste . . . 137
Mutabilité des listes . . . 141
Suite de Conway . . . 145

**Programmation OCaml impérative** **147**
Outils de programmation impérative . . . 147
Séquences d'instructions . . . 147
Blocs . . . 147
Boucles inconditionnelles (|for|) . . . 148
Boucles conditionnelles (|while|) . . . 149
Références . . . 149
Les références . . . 149
Utilisation . . . 150
Égalité, identité . . . 151
Une dernière remarque pour clore . . . 152
Objets Caml avec mutabilité . . . 152
Cas du type « enregistrement » . . . 152
Retour sur les références . . . 154
Un autre objet mutable : les tableaux (|'a array|) . . . 154
Tableaux bidimensionnels . . . 155