# TP7 : C, système

De la semaine du 4 décembre à la semaine du 22 janvier

Un indicateur de difficulté est fourni par des étoiles (⋆) sur certaines questions ou exercices.

Certains exercices et certaines questions comportent un symbole (🔧) pour indiquer la présence de difficultés liées au système. La maîtrise de ces subtilités techniques ne sera pas demandé sans la possibilité d'utiliser un ordinateur.

Les exercices qui sont numérottés avec des lettres et non des chiffres sont des exercices spécifiques de la version numérique.

## 1 Programmation de base en C

**Exercice 1. Modulo**

Il s'avère qu'on dispose de l'opérateur modulo % qui calcule le reste de la division euclidienne de sa première opérande par sa seconde opérande.

1. Proposer une implémentation d'une fonction de prototype `int modulo_moins(int a, int b)` qui calcule $a \mod b$ à l'aide d'une boucle et de soustraction.
2. Proposer une implémentation d'une fonction de prototype `int modulo_vid(int a, int b)` qui calcule $a \mod b$ à l'aide de la division entière /.

**Exercice 2. Boucles**

1. Proposer un programme qui affiche tous les nombres premiers de 1 à 100.
2. On se donne la suite de Fibonacci définie par

$$u_0 = 0$$
$$u_1 = 1$$
$$\forall n \in \mathbb{N},\ u_{n+2} = u_{n+1} + u_n$$

Proposer un programme qui affiche les nombres de Fibonacci jusqu'à $n = 100$.

**Exercice 3. Un générateur pseudo-aléatoire**

Les générateurs de nombres pseudo-aléatoire sont des outils qui permettent d'obtenir une succession de nombres qui vérifient certaines propriétés aléatoire qui permettent de s'en servir pour des applications où il peut être intéressant d'avoir un comportement qui est imprévisible.

Ici, on s'intéresse à un générateur congruentiel linéaire qui fonctionne avec le principe suivant :

On commence avec une graine $X_0$, et on se donne des entiers $m$, $a$ et $c$ qui vérifient des propriétés particulières, et on calcule la suite des $(X_n)_{n\geqslant 0}$ de la manière suivante :

$$X_{n+1} = (aX_n + c) \mod m$$

1. On prend les valeurs suivantes :

$$X_0 = 3; a = 15; c = 1; m = 7$$

Calculer les valeurs de $X_1$, $X_2$, $X_3$.

2. Proposer une fonction de prototype `int suivant (int actuel, int a, int c, int m)` qui calcule à partir de la valeur courrante de $X_n$, de $a$, $c$ et $m$, la valeur de $X_{n+1}$.
3. On prend les valeurs suivantes :

$$X_0 = 1; a = 2; c = 0; m = 7$$

Calculer les valeurs de $X_i$ pour $i \leqslant 6$.

Que peut-on remarquer ?

4. Montrer que la suite $(X_n)_{n\geqslant 0}$ est ultimemement périodique, et montrer que sa période est au plus $m$.
5. Des propriétés arthimétiques nous permettent de montrer que si l'on veut que, quelque soit $X_0$, cette période soit égale à $m$, en supposant $c \neq 0$, il faut trouver $a$, $m$ et $c$ tels que :
   — $c$ est premier avec $m$ (c'est-à-dire que $\text{pgcd}(c, m) = 1$) ;
   — pour chaque $p$ premier qui divise $m$, $(a-1)$ est un multiple de $p$ ;
   — $(a-1)$ est multiple de 4 si $m$ est multiple de $m$.

   Proposer une fonction de prototype `bool est_valide_c_nonnul(int a, int m, int c)` qui vérifie que des entiers $a$, $m$ et $c$ vérifient ces propriétés.
6. Trouver un exemple de $a$, $m$, et $c$ qui vérifient ces propriétés.
7. Si $c = 0$, il existe des conditions suffisantes sur $a$ et $m$ tels que la période soit toujours maximale. À savoir les conditions suivantes :
   — $m$ est premier ;
   — $a^{m-1} - 1$ est un multiple de $m$ ;
   — $a^j - 1$ n'est pas un multiple de $m$ pour tout $j \in \{1, 2, 3, ..., m-2\}$.

   Proposer une fonction de prototype `int est_valide(int a, int m)` qui vérifie si une paire $(a, m)$ vérifie ces propriétés.
8. Trouver un exemple de $a$ et $m$ qui vérifient ces propriétés.
9. En pratique, on veut un nombre $m$ le plus grand possible pour avoir des périodes plus longues. Comment peut-on faire pour avoir un nombre dans un interval plus petit à partir de $X_n$ ?

*En pratique, les générateur pseudo-aléatoire ne sont pas utilisés pour les données critiques (chiffrement, sécurité, ...) car encore trop facile à prédire. On a besoin d'avoir recours à d'autres techniques.*

### Exercice 4. Quelques petits jeux pour la console

1. Proposer un programme qui choisit un nombre aléatoirement entre 0 et 100 et qui regarde des entrées en affichant si les nombres en entrée sont supérieurs ou inférieur jusqu'à ce que l'entrée soit égale au nombre choisi.

   Pour générer un nombre aléatoire, on pourra utiliser les bibliothèques suivantes :

```c
#include <time.h>
#include <stdlibs.h>
```

   Il faudra initialiser ensuite le générateur pseudo aléatoire :

```c
srandr(time(NULL))
```

   Par la suite, il sera possible de générer un nombre entre `0` et `RAND_MAX` avec la fonction `rand()`.

   (On pourra autrement prendre le générateur aléatoir de l'exercice précédent avec une graine initialisée grâce au temps.)

2. Proposer un jeu de pendu.

   On pourra utiliser un fichier qui contient une liste de mots possibles (🔑), ou demander à une autre personne de rentrer un mot (puis sauter un certain nombre de lignes pour ne plus voir le nombre).

3. Proposer un jeu de type Wordle.

   On pourra utiliser un fichier qui contient une liste de mots possibles (🔑), ou demander à une autre personne de rentrer un mot (puis sauter un certain nombre de lignes pour ne plus voir le mot).

### Exercice 5. Variante de i++

On rappelle que l'instruction `i++;` incrémente de 1 la valeur contenue dans `i`.

1. Il s'avère que l'expression `i++` a une valeur de type entière. Quelle est la valeur de cette expression ?
2. Que fait la syntaxe `++i;` ?
3. Quelle est la différence entre ces deux syntaxes ?

### Exercice 6. Opérateurs d'affectation

1. Proposer un programme qui détermine ce que fait la syntaxe `a+=k;` où `a` est une variable et `k` est une expression de type entier.
2. En déduire la fonction des opérateurs $-=$, $*=$, $/=$ et $\%=$.

### Exercice 7. Calcul d'exponentiation

On dispose d'une fonction d'exponentiation dans le module `<math.h>`, mais celle-ci fonctionne sur les flottants. Nous aimerions avoir à notre disposition une fonction d'exponentiation sur les entiers.

1. Proposer une fonction de prototype `int puissance(int x, int n)` qui calcule $x^n$. (On ne passera bien sûr pas par les flottants)
2. On remarque que :
$$x^n = \begin{cases} \left(x^{\frac{n}{2}}\right)^2 & \text{si } n \text{ est pair,} \\ x \cdot \left(x^{\frac{n-1}{2}}\right)^2 & \text{si } n \text{ est impair.} \end{cases}$$
   Proposer une fonction qui calcule $x^n$ mais plus rapidement en utilisant une fonction récursive.
3. Désormais, on essaye d'avoir une complexité du même ordre, mais en utilisant une implémentation impérative.
   (a) En fonction de l'écriture binaire de $n$, quels sont les cas successifs des appels récursifs dans la fonction précédente ?
   (b) En déduire une implémentation impérative.
4. En écrivant $n = \sum_{k=0}^{N} a^k 2^k$ en base deux, et en remarquant que :
$$x^n = \prod_{k \mid a_k = 1} x^{2^k}$$
   Proposer une implémentation impérative de complexité similaire.

### Exercice A. Récursivité terminale en C (🔧)

On propose le code suivant en C :

```c
int factorielle(int n, int accumulateur)
{
    if (n==0)
    {
        return accumulateur;
    }
    return factorielle(n-1, accumulateur * n);
}
int main() {
    factorielle(10, 1);
}
```

En observant les codes assembleurs fournis par les deux commandes suivantes, observer que dans un cas, il n'y a pas d'appels récursifs :

```
gcc -std=c99 -S -O2 nom_du_fichier.c
```

```
gcc -std=c99 -S -O1 nom_du_fichier.c
```

*Il peut être difficile de prévoir si l'optimisation de récursivité terminale a lieu ou non en C en fonction des optimisations de gcc. En règle générale, on évitera d'utiliser de la récursivité en C.*

### Exercice 8. Évaluation paresseuse en C

On rappelle que l'évaluation paresseuse est un procédé qui permet à certains opérateurs de ne pas évaluer tous ses opérandes.

Par exemple, lors de l'évaluation de `true || (1/0 == 0)`, aucune erreur n'est levée car la partie de gauche permet déjà de conclure sans réaliser le calcul de la partie de droite.

1. Proposer un exemple de code en C qui montre que l'opérateur logique `&&` utilise l'évaluation paresseuse.
2. Est-ce que l'opérateur bit-à-bit `&` utilise l'évaluation paresseuse sur les booléens ?

### Exercice 9. Cohérence des opérations sur les booléens et les entiers

1. Proposer une fonction de prototype `bool vers_bool(int n)` qui convertit un entier vers un booléen en renvoyant `true` si $n \neq 0$ et `false` sinon.
2. Montrer que pour toute paire d'entier $n$ et $m$, les expressions suivantes sont égales :

```
vers_bool(n | m)
```

```
vers_bool(n)||vers_bool(m)
```

3. Est-ce que pour toute paire d'entier $n$ et $m$, les expressions suivantes sont égales ?

```
vers_bool(n & m)
```

```
vers_bool(n)&&vers_bool(m)
```

### Exercice 10. Déclaration de fonction sans définition pour des fonctions mutuellement récursives

Grâce à une déclaration sans définition, on peut définir des fonctions mutuellemnt récursives :

```c
bool est_pair(int n);
bool est_impair(int n){
    if (n== 0) {return false;}
    return est_pair(n-1);
}
bool est_pair(int n){
    if (n==0) {return impair;}
    return est_impair(n-1);
}
```

1. Pour chaque fonction, dire à quelle ligne elle est déclarée, et à quelle ligne elle est définie.
2. On se donne les suites $F$ et $M$ pour calculer

$$\begin{aligned} F(0) &= 1 \\ M(0) &= 0 \\ \forall n > 0\ F(n) &= n - M(F(n-1)) \\ \forall n > 0\ M(n) &= n - F(M(n-1)) \end{aligned}$$

Proposer des fonctions en C qui permettent de calculer $F$ et $M$.

# 2 Représentation des nombres

## 2.1 Représentation des entiers

**Exercice 11. Taille des entiers en C**

On prendra soin d'importer la bibliothèque pour gérer les différentes tailles d'entiers en C.

```c
#include <stdint.h>
```

1. En observant quand a lieu un dépassement, pour chacun des types suivant, calculer la plus grande valeur et la plus petite valeur possible pour ce type.
   (a) `int8_t`
   (b) `uint8_t`
   (c) `int16_t`
   (d) `uint16_t`
   (e) `unsigned`
   (f) `int`
2. Est-ce qu'il est possible de faire de même pour le type `int64_t` ?

**Exercice 12. Droits dans un système Unix**

Dans les système Unix, concernant les permissions, il existe différentes droits associées à chaque fichier :
— `r` ( *Read* en anglais) : le droit de lecture qui autorise à lire le fichier ;
— `w` ( *Write* en anglais) : le droit en écriture qui permet de modifier le fichier ;
— `x` ( *eXecute* en anglais) : le droit en exécution qui permet d'exécuter un fichier.
On décide de noter les droits au format `rwx`. La lettre est représenté quand on a l'autorisation de faire l'action concernée, ou bien – si l'action n'est pas autorisée. Par exemple `r--` décrit le fait d'avoir le droit de lire le fichier, mais pas de le modifier ou de l'exécuter.

On peut encoder ces droits à l'aide d'un entier en utilisant une écriture binaire. Ainsi, le droit `r-x` est représenté par l'entier $\overline{101}^2 = 5$.

1. À quel entier correspond le droit −wx ?
2. À quel droit correspond l'entier 7 ?
3. Proposer une fonction de prototype `void afficher (int droit)` qui affiche les droits associés à un entier.
4. De même proposer une fonction de prototype `int convertir (char [] droit)` qui convertit une chaîne de caractère représentant des droits en l'entier associé.

En pratique, les droits des fichiers sont représentés sur 9 bits au format rwxrwxrwx. Les trois bits de poids faibles (ceux le plus à droite dans la représentation) indiquent les droits du ou de la propriétaire du fichier, les trois bits du milieu représentent les droits du groupe du ou de la propriétaire du fichier, et enfin les trois bits de poids forts correspondent aux droits du restes des utilisateurs et des utilisatrices.

Ainsi, les droits r−−rw−rwx précisent que le ou la propriétaire a tous les droits sur le fichiers, que son groupe a les droits en écriture et en lecture, et que le reste a le droit en lecture seulement.

5. Proposer une fonction de prototype `void afficher_tout (int droit)` qui affiche les droits sur 9 bits.
6. De même proposer une fonction de prototype `int convertir_tout (char [] droit)` qui convertit une chaîne de caractère à 9 caractères représentant des droits en l'entier associé.

### Exercice 13. Adresses IPv4

Le protocol IPv4 est la première version d'Internet Protocol.

On représente les adresse IP sous la force de 4 entiers représentés sur un octet au format a.b.c.d. Les valeurs des adresse IP varient de 0.0.0.0 à 255.255.255.255.

Comme une adresse IPv4 est sur 4 octets, on peut utiliser un entier `unsigned`.

1. Proposer une fonction `void afficher (int n)` qui affiche la chaîne de caractère au format a.b.c.d.

Dans une adresse IP, une partie est réservé au réseau, et l'autre partie à l'hôte au sein du réseau.

Un masque de sous réseau est un entier qui s'écrit en binaire sous la forme $\overline{1...10...0}^2$ et qui permet de savoir quels bits de l'adresse sont réservés au sous-réseau.

Par exemple, le masque 255.128.0.0 correspond à garder les 9 premiers bits. Si on applique ce masque à l'adresse 192.168.0.1, on obtient le sous-réseau 192.128.0.0.

2. Proposer une fonction de prototype `int sousreseau (int adresse, int masque)` qui calcule l'adresse du sous-réseau avec le masque en entrée.
3. Parfois on représente une adresse au format a.b.c.d/e où $e$ indique le nombre de bit égaux à 1 dans le masque de sous réseau.

   Proposer une fonction de prototype `int calculer_masque (int n)` qui calcule le masque à partir du nombre de bits égaux à 1 dans le masque.

### Exercice 14. Représentation d'ensembles

On désire représenter des ensembles d'entiers à l'aide d'entiers naturels. L'idée est d'utiliser la représentation binaire d'un nombre $n = \sum_{i=0}^{k} a_i 2^i = \overline{a_k a_{k-1} ... a_1 a_0}^2$. L'ensemble $E_n$ qui correspond à cet entier $n$ est l'ensemble des entiers $i$ tels que $i \leqslant k$ et $a_i = 1$.

Ainsi :

$$E_n = \{i \in \mathbb{N} | i \leqslant k \text{ et } a_i = 1\} \text{ où } n = \sum_{i=0}^{k} a_i 2^i = \overline{a_k a_{k-1} ... a_1 a_0}^2$$

Par exemple, l'ensemble $\{1, 3, 4\}$ peut être représenté par l'entier $\overline{11010}^2 = 26$.

On utilisera des entiers non signés.

1. (a) Quel entier représente l'ensemble $\{0, 1, 5\}$ ?
   (b) Quel ensemble représente l'entier 37 ?
2. (a) Quel est l'ensemble des entiers qui peuvent être représentés avec le type `unsigned` ?
   On notera $U$ cet ensemble. Ainsi, pour chaque $n$, $E_n \subset U$.
   (b) Quel est l'entier $n$ tel que $E_n = U$ ?
3. (a) Proposer une fonction de prototype `int singleton (int k)` qui construit l'entier qui représente le singleton $\{k\}$.
   (b) Proposer une fonction de prototype `int range (int k)` qui construit l'entier qui représente l'ensemble {0, 1, ..., k-1} des entiers de 0 à $k-1$.
   (c) Proposer une fonction de prototype `void afficher (int n)` qui affiche l'ensemble $E_n$.
4. On cherche à représenter des opérations sur les ensembles par des opérations sur les entiers.
   Pour chacune de ces questions, on proposera une implémentation en C qui permet de réaliser l'opération concernée.
   (a) Si $n$ et $m$ sont des entiers qui représentent des ensembles $E_n$ et $E_m$, comment calculer l'entier qui représente l'union de ces ensembles $E_n \cup E_m$ ?
   (b) Si $n$ et $m$ sont des entiers qui représentent des ensembles $E_n$ et $E_m$, comment calculer l'entier qui représente l'intersection de ces ensembles $E_n \cap E_m$ ?
   (c) Si $n$ est un entier qui représente l'ensemble $E_n$, comment calculer l'entier qui représente le complémentaire de cet entier $E_n^c = \{x \in U | x \notin E_n\}$ ?
   (d) Comment vérifier si $E_n \subset E_m$ pour deux entiers $n$ et $m$ ?
   (e) Comment calculer la différence $E_n \setminus E_m = \{x \in E_n | x \notin E_m\}$ ?
   (f) Comment calculer la différence symmétrique $E_n \Delta E_m$ l'ensemble des éléments qui appartiennent exactement à un de ces deux ensembles ?
   *On pourra utiliser un seul opérateur logique, ou l'une des formules suivantes :*

$$A \Delta B = (A \cup B) \setminus (A \cap B)$$
$$A \Delta B = (A \setminus B) \cup (B \setminus A)$$

5. On s'intéresse aux effets de certaines opérations binaires sur les ensembles représentés par des entiers.
   (a) Si $n$ est un entier qui représente un ensemble, quel est l'ensemble représenté par `n>>1` ?
   (b) Si $n$ est un entier qui représente un ensemble, quel est l'ensemble représenté par `n<<1` ?
   (c) Si $n$ est un entier qui représente un ensemble, quel est l'ensemble représenté par `x& (~x + 1)` ?
6. On cherche à calculer la somme des cardinaux des entiers que l'on peut représenter sur $p$ bits.
   (a) Si $n$ est un entier non nul qui représente un ensemble, quel est l'ensemble $E_{n-1}$ représenté par $n-1$ ?
   (b) Si $n$ est un entier non nul, que vaut `n&(n−1)` ?
   (c) En déduire une fonction de prototype `int card (unsigned n)` qui calcul le cardinal d'un ensemble représenté par un entier.
   Quelle est la complexité de cette fonction ?

   On nomme $u_p$ la somme des cardinaux des ensembles représentés par les entiers naturels codés sur $p$ bits. On cherche à calculer $u_p$.
   (d) Proposer une fonction de prototype `int somme_card(int p)` qui énumère sur tous les nombres entre 0 et $2^p - 1$ pour calculer $u_p$.
   (e) Combien vaut $u_0$ ?
   (f) Montrer que, pour tout $p \geqslant 0$, $u_{p+1} = 2^p + 2u_p$.
   (g) En déduire que, pour tout $p \geqslant 0$, $u_p = p2^p$.

**Exercice 15. Opérations bit pour l'opposé d'un nombre**

En utilisant seulement l'addition + et la négation bit à bit ~, proposer une implémentation de la fonction qui renvoie l'opposé d'un entier $x \mapsto -x$ de prototype :

```c
int oppose(int x)
```

**Exercice 16. Décalage à droite arithmétique**

En utilisant la conversion de variable en C montrer que le comportement de l'opérateur >> dépend du type de sa première opérande.

## 2.2 Flottants

**Exercice 17. Erreurs de flottants**

On cherche à exhiber quelques erreurs communes qui surviennent lors de calculs sur les flottants.
Pour chaque calcul suivant, donner le résultat de la machine, et proposer une interprétation du comportement observé.

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

int main() {
    double a;
    a = pow(2.0, -1073.0);
    if (a== 0.)
        printf("a est nul.\n");
    else
        printf("a est non nul.\n");
    if (a*a== 0.)
        printf("a*a est nul.\n");
    else
        printf("a*a est non nul.\n");
    return 0;
}
```

2.
```c
#include <stdio.h>
#include <stdlib.h>
#include <math.h>

int main() {
    double a; double b;
    a = pow(2.0, -60.0);
    b = 1.0;
    a = (b + a) - b;
    if (a== 0.)
        printf("a est nul.\n");
    else
        printf("a est non nul.\n");
    return 0;
}
```

3.
```c
#include <stdio.h>
#include <stdlib.h>
#include <math.h>

int main() {
    double a = 0.1; double b;
    a = pow(2.0, -60.0);
    b = 1.0;
    a = (b + a) - b;

    return 0;
}
```

4.
```c
#include <stdio.h>
#include <stdlib.h>
#include <math.h>

int main() {
    double a = 0.1;
    double b = 0.3;
    if (a + a + a == b)
        printf("a + a + a n'est pas egal a b");
    else
        printf("a + a + a n'est pas egal a b");

    return 0;
}
```

5.
```c
#include <stdio.h>
#include <stdlib.h>
#include <math.h>

int main() {
    double a;
    double b = 0.0;
    a = pow(10.0, -8);
    for (int i = 0; i<100000000; i++)
    {
        b = b + a;
    }
    printf("b est egal a %.10f", b);
    return 0;
}
```

**Exercice 18. Conversion depuis les flottants**

On peut convertir un flottant vers les entiers à l'aide de la syntaxe `(int) f` où `f` est une expression flottante.

1. Que se passe-t-il quand on convertit 2.3 ? 2.8 ?
2. Que se passe-t-il quand on convertit −2.3 ? −2.8 ?
3. Que se passe-t-il quand on convertit un flottant plus grand que la capacité mémoire des entiers ? Que se passe-t-il quand on convertit NaN ou un infini ?
4. Les comportements liés à la conversion de flottants en dehors de la capacité mémoire des entiers, ou des valeurs spéciales des flottants ne sont pas spécifiés.

   Proposer une fonction de prototype `bool convertir_depuis_flottant(double f, int * cible)` qui convertit si le comportement est spécifié un flottant vers un entier et mets cette valeur à l'adresse `cible`, et qui renvoie si la conversion a pu avoir lieu.

**Exercice 19. Opérateur $\mathcal{S}$.**

On s'intéresse à l'interprétation d'un flottant positif comme un entier non signé sur 64 bits. On se donne $x$ un flottant, et on note $s$ le bit de signe, $E$ l'exposant sur 11 bits, ainsi que $f$ la fraction sur 53 bits.

On propose la fonction $\mathcal{S}$ des flottants dans les flottants qui prend un flottant en entrée, l'interprète comme un entier, lui ajoute un, puis le reconvertit en flottant.

1. Supposons $s = 0$, $E = 0$, et $f \neq \overline{1...1}^2$. Que vaut $\mathcal{S}(x)$ ?
2. Supposons $s = 0$, $1 \leqslant E \leqslant 2^{11} - 2$, et $f \neq \overline{1...1}^2$. Que vaut $\mathcal{S}(x)$ ?
3. Supposons $s = 0$, $E = 0$, et $f = \overline{1...1}^2$. Que vaut $\mathcal{S}(x)$ ?
4. Supposons $s = 0$, $0 \leqslant E \leqslant 2^{11} - 2$, et $f \neq \overline{1...1}^2$. Que vaut $\mathcal{S}(x)$ ?
5. Supposons $s = 0$, $0 \leqslant E \leqslant 2^{11} - 2$, et $f = \overline{1...1}^2$. Que vaut $\mathcal{S}(x)$ ?
6. Conclure sur $\mathcal{S}$.

**Exercice 20. Caractérisation des flottants** (⋆)

Montrer que les flottants sur 64 bits sont exactements les rationels de la forme $n2^e$ avec $n$ et $e$ entiers tels que $|n| < 2^{53}$ et $-1074 \leqslant e \leqslant 970$.

**Exercice B. Preuves sur les flottants** (⋆)

On note $\mathbb{F}$ l'ensemble des flottants. On remarque que $\mathbb{F} \subset \mathbb{R}$.

On définit la fonction $\mathcal{N} : \mathbb{R} \to \mathbb{F}$ qui arrondi un réel au flottant le plus proche (en arrondissant à la fraction qui finit par un zéro en cas d'égalité).

On note $\oplus$ l'opération sur les flottants de la manière suivante :

$$x \oplus y = \mathcal{N}(x + y)$$

De même, on définit les opératations internes sur les flottants $\ominus$ et $\otimes$ :

$$x \ominus y = \mathcal{N}(x - y)$$

$$x \otimes y = \mathcal{N}(x \times y)$$

*On pourra utiliser le résultat de l'exercice précédent sans le démontrer.*

1. Calculer $(-2^{52} \oplus 2^{52}) \oplus \frac{1}{8}$ et $-2^{52} \oplus (2^{52} \oplus \frac{1}{8})$.
   Que peut on en déduire sur les propriétés de $\oplus$ ?
2. Proposer un flottant non nul $x$ tel que $x \otimes x = 0$.
3. Quel est le plus grand flottant $x$ tel que $1 \oplus x = 1$.
4. Prouver que pour tout $x, y \in \mathbb{F}^2$, si $|x + y| \leqslant 2^{-1022}$, alors $x \oplus y = x + y$.
5. Prouver que pour tout $x, y \in \mathbb{F}^2$, si $\frac{y}{2} \leqslant x \leqslant 2y$, alors $x \ominus y = x - y$.

# 3 Gestion de la mémoire

**Exercice 21. Botanique des erreurs de pointeurs**

Pour chaque code suivant, exécuter le programme et expliquer pourquoi le code ne respecte pas la spécification, ou bien les différentes erreurs que vous pourriez rencontrer :

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

void incrementer (int a)
{
    a = a + 1;
}
int main() {
    /* Initialise a a 0, puis l'incremente, et enfin l'affiche */
    int a = 0;
    incrementer(a);
    printf("a vaut 1, en effet : %d\n", a);
    return 0;
}
```

2.
```c
#include <stdio.h>

void incrementer (int *a)
{
    *a = *a + 1;
}
int main() {
    /* Initialise a a 0, puis l'incremente, et enfin l'affiche */
    int a = 0;
    incrementer(a);
    printf("a vaut 1, en effet : %d\n", a);
    return 0;
}
```

3.
```c
#include <stdio.h>

void incrementer (int *a)
{
    *a = *a + 1;
}
int main() {
    /* Initialise a et b a 0, puis les incremente, et enfin l'affiche */
    int* a_ptr, b_ptr;
    int a = 0; int b = 0;
    a_ptr = &a;
    b_ptr = &b;
    incrementer(a);
    incrementer(b);
    printf("a vaut 1, en effet : %d\n", a);
    printf("b vaut 1, en effet : %d\n", b);
    return 0;
}
```

*Pour cette raison, certains standard déconseillent de déclarer plusieurs variables à la fois. C'est aussi pourquoi certains standard préfèrent écrire* `int *a` *que* `int* a`.

### Exercice 22. Pointeurs

1. Proposer une fonction de prototype `void mettre_a_zero (int * cible)` qui met à zéro un entier à une adresse donnée.
2. Proposer une fonction de prototype `void incrementer (int * cible)` qui augmente de 1 la valeur contenue dans un pointeur.
3. Proposer une fonction de prototype `int nouvelle_id (int * cible)` qui incrémente la valeur contenue dans un pointeur et renvoie la nouvelle valeur.
4. Proposer une fonction de prototype `void echanger(int * a, int * b)` qui échange le contenu de deux entiers aux adresses $a$ et $b$.

### Exercice 23. Permutation de deux entiers

On suppose qu'on a deux variables entières $a$ et $b$ dont on veut échanger les valeurs.

```
int a,b;
```

L'idée est de proposer une fonction qui prend les deux adresses des entiers, et échangent les valeurs contenues dans les variables. Le prototype attendu est donc le suivant :

```
void f (int *aptr, int bptr)
```

On s'attend au comportement suivant :

```
int a = 0;
int b = 1;
f(&a, &b);
printf("a vaut %d et b vaut %d.",a, b);
```

```
a vaut 1 et b vaut 0
```

1. Proposer un programme de prototype `void echanger_temp(int * aptr, int * bptr)` qui échange le contenu des deux variables à l'aide d'une variable temporaire.
2. On cherche désormais à utiliser des propriétés arithmétiques pour pouvoir se passer d'une variable temporaire. Compléter le code suivant de sorte à ce que la fonction suivante réalise l'échange des valeurs contenues dans les deux variables :

```
void echanger_arithmetique(int * aptr, int * bptr)
{
    *aptr = *aptr + *bptr;
    *bptr = ... ;
    *aptr = ... ;
}
```

3. On veut désormais utiliser une propriété logique sur les bits.
   (a) Montrer que pour tout $x$, $y$ et $z$ trois entiers, on a les propriétés suivantes :
   - `x^y` est égal à `y^x`
   - `x^0` est égal à `x`
   - `x^x` est égal à `0`
   - `(x^y)^z` est égal à `x ^(y^z)`

   (b) En déduire une fonction de prototype `echanger_logique(int * aptr, int * bptr)` qui utilise le ou exclusif pour pouvoir échanger deux variables.

### Exercice 24. Allocation mémoire

1. Proposer une fonction `void liberer (int * cible)` qui libère la mémoire d'un entier sur le tas dont l'adresse est en argument.
2. Proposer une fonction `int liberer_et_renvoyer (int * cible)` qui libère la mémoire d'un entier sur le tas et renvoie la valeur qui était contenue à cette adresse.

### Exercice C. Mot-clef static

Les variables statiques sont des variables allouée de manière globale au programme.

On peut s'en servir de différentes manière, à différents usages, mais l'idée principale est que la variable n'est initialisée qu'une seule fois, et ensuite sa valeur est conservée à travers les différents appels.

```c
int compteur()
{
    static int compte = 0;
    compte++;
    return compte;
}
```

On ne peut cependant pas avoir accès à la variable en dehors de l'espace de nommage concerné :

```c
int compteur()
{
    static int compte = 0;
    compte++;
    return compte
}
int main()
{
    int i;
    compteur();
    printf("%d\n", compte)
    /* Meme si la variable compte est quelque part dans la memoire, nous ne pouvons
    pas avoir acces a cette variable. */
    return 0;
}
```

Ainsi, il faut faire la distinction entre la *durée de vie* de la variable statique (elle survit à son bloc d'activation) et sa portée (elle n'est accessible qu'au sein de la partie du code où elle est définie).

1. Afficher les valeurs successives des appels à `compteur()`.
2. Proposer une fonction de prototype `int somme(int n)` qui renvoie un entier égal à la somme de tous les entiers qui ont été utilisés en argument de cette fonction.

   Ainsi, lors du premier appel à `somme(2)`, la réponse est 2, mais si un deuxième appel est effectué à `somme(2)`, le résultat serait 4. Si un troisième appel `somme(8)` est effectué, on obtiendrait 12.
3. Proposer une variante du générateur pseudo aléatoire de l'exercice 2 pour n'avoir aucun argument (on pourra fixer $a$, $c$, et $m$).

   Le prototype attendu est le suivant :

   ```c
   int generateur()
   ```

*En pratique, les variables statiques sont enregistrées dans le segment de donnée, tous comme les variables globales. La différences principales est donc principalement la portée.*

### Exercice 25. Utilisation de pointeurs pour renvoyer plus d'une valeur

On peut utiliser des pointeurs pour permettre à une fonction de renvoyer plusieurs résultats.

1. Proposer une fonction de prototype `void division_euclidienne (int dividende, int diviseur, int * quotient, int * reste)` qui calcul le résultat de la division euclidienne du dividende par le divisieur et modifie les valeurs pointées par `quotient` et `reste` pour y mettre le résultat.
2. En déduire un programme qui demande deux nombres en entrée et qui affiche le quotient et le reste de la division euclidienne du premier par le second.

   *On pourra utiliser `scanf` ou les arguments du programme directement.*

### Exercice 26. Cas d'une fuite mémoire

On propose la fonction suivante :

```c
int fibonacci(int n){
    int *memoire = (int*) malloc(sizeof(int) * (n+1));
    if (n<1) {return 0;}
    memoire[0] = 0;
    memoire[1] = 1;
    for (int i = 2; i<(n+1);i++)
    {
        memoire[i] = memoire[i-1] + memoire[i-2];
    }
    return memoire[n];
}
```

1. (🔧) À l'aide de la commande top (ou htop) dans le terminal et d'un programme qui utilise cette fonction, mettez en évidence qu'il y a une fuite de mémoire dans le code.

   *Attention à ne pas faire planter votre machine.*
2. Proposer une version corrigé de ce code sans fuite mémoire.

**Exercice D. Boucle sans rien du tout en utilisant un dépassemement de mémoire tampon.**

On propose la fonction suivante :

```c
void f()
{
    char tampon[16];
    scanf("%s", tampon);
    return;
}
```

1. Que se passe-t-il sur la pile lorsque une entrée plus grande que la mémoire tampon est utilisée dans cette fonction ?
2. Comment pourrait-on utiliser ce dépassement de mémoire tampon pour modifier l'adresse de retour dans la pile des appels ?
3. (🔧🔧) Proposer une fonction qui boucle sur elle-même en utilisant un dépassement de mémoire tampon.

   *Pour faire marcher cette fonction là dans la pratique, on pourra avoir besoin de modifier les paramètre du noyau pour retirer la randomization des adresses.*

   *Cette opération ne peut être réalisée qu'avec les droits administrateurs sur la machine sur laquelle vous travailler, justement pour des raisons de sécurités.*

# 4 Types structurés en C

## 4.1 Struct

**Exercice E. Utilisation sur la pile de structures**

À la place d'utiliser le complément à deux pour les entiers relatifs, on utilise une structure qui contient la valeur absolue du nombre ainsi que le signe du nombre.

Le type que l'on utilise est le suivant :

```c
typedef struct _relatif {
    unsigned int absolue;
    bool signe;
} relatif_t;
```

Le booléen signe vaut vrai si le nombre est négatif.

1. Comment représenter le nombre 8 de cette manière ? −4 ?
2. Que représente le relatif suivant :

```
relatif_t a = {12, true}
```

   *On peut initialiser une structure avec un générateur. Cela ne peut pas être utilisé pour modifier une valeur déjà déclarée.*
3. Quelles sont les représentations de 0 ?
4. Proposer une fonction de prototype `void afficher(relatif_t rel)` qui affiche un entier relatif en entrée.
5. Proposer une fonction de prototype `relatif_t ajouter(relatif_t a, relatif_t b)` qui calcule la somme de deux entiers relatifs.

### Exercice F. Utilisation des structures pour renvoyer plus d'une valeur

On se donne le type suivant :

```
struct _resultat {
    int quotient ;
    int reste;
};
```

1. Comment utiliser `typedef` pour créer un type `resultat_t` qui soit équivalent à `struct _resultat` ?
2. Comment aurait-on pu faire cette définition en une seule instruction ?
3. En utilisant ce type, proposer une fonction de prototype `resultat_t division_euclidienne(int dividende, int diviseur)` qui renvoie le quotient et le reste de la division euclidienne du dividende et du diviseur à l'aide du type `resultat_t`.

### Exercice G. Ordre de priorité des opérateurs

On se donne les type suivants :

```
struct _a {
    int c1;
};

struct _b {
    int * c2;
};

typedef struct _a a_t;
typedef struct _b b_t;
```

1. On se donne `a_ptr` de type `a_t *`, et `b` de type `b_t`.
   Expliquez la différence entre les deux codes suivants :

```
(*a_ptr).c1;
```

```
*b.c2;
```

2. Dans quel cas aurions nous pu utiliser l'opérateur `->` ?

### Exercice H. Allocation et modification d'une structure sur le tas

On se donne les types suivant :

```c
struct _somme {
    int total;
    int dernier_ajout;
};
typedef struct _somme somme_t;
```

L'objectif est de se servir de ce type pour pouvoir représenter un accumulateur dans lequel on ajoute des entiers. On veut pouvoir se servir du champs dernier_ajout pour pouvoir revenir d'une étape en arrière dans le calcul.

L'application est donc la suivante : initialement, les deux grandeurs sont nulles :

| Total : 0 | Dernier Ajout : 0 |
|---|---|

Je peux ajouter des entiers, par exemple 2 puis 3, ce qui donne successivement :

| Total : 2 | Dernier Ajout : 2 |
|---|---|

puis

| Total : 5 | Dernier Ajout : 3 |
|---|---|

Si je veux revenir en arrière, je peux le faire sur une étape :

| Total : 2 | Dernier Ajout : 0 |
|---|---|

Mais je ne peux pas revenir plus en amont à cause du fait que je ne me souvenais que de la dernière étape.

1. Proposer une fonction de prototype somme_t * nouveau(); qui alloue sur le tas un objet de type somme_t vide (avec un total de 0 et un dernier ajout de 0) et qui renvoie un pointeur vers cette structure.
2. Proposer une fonction de prototype void ajouter(somme_t * s_ptr, int n) qui ajoute un entier dans la somme et mets à jour le dernier ajout.
3. Proposer une fonction de prototype void retour_arriere(somme_t * s_ptr) qui modifie une somme pour pouvoir revenir d'une étape en arrière. On modifiera le champs dernier_ajout pour obtenir 0.
4. Proposer une fonction de prototype void detruire(somme_t * s_ptr) qui libère la mémoire associée à une somme passée en argument.
5. Dans cette application, nous ne pouvons revenir que d'une étape en arrière. Comment aurait-on pu procéder pour pouvoir revenir d'un nombre arbitraire d'étapes en arrière ?

### Exercice I. Taille d'une structure (🔧)

1. On se donne les type suivant :

```c
struct _a {
    int16_t c1;
    int8_t c2;
}
```

```c
struct _b {
    int8_t c1;
    int8_t c2;
    int8_t c3;
}
```

Quelle est la taille minimale qu'il faudrait en théorie pour stocker un élément de chacune de ces structures ? Quelle est la taille donnée par `sizeof` ?

2. À l'aide de la notion d'alignement, expliquer les raisons de cette différence.

*L'alignement des données en pratique dépend de l'implémentation du compilateur.*

## 4.2 Tableaux

**Exercice 27. Quelques exercices sur des tableaux**

On prendra soin de tester chacune des fonctions implémentées.

1. Proposer une fonction de prototype `void afficher(int tableau[], int taille)` qui à l'aide d'un tableau et de sa taille affiche le contenu de ce tableau.
2. Proposer une fonction de prototype `void initialiser(int tableau[], int taille)` qui à l'aide d'un tableau et de sa taille, mets toutes les valeurs de ce tableau à 0.
3. Proposer une fonction de prototype `int somme(int tableau[], int taille)` qui à l'aide d'un tableau et de sa taille, renvoie la somme de tous ses éléments.
4. Proposer une fonction de prototype `int max(int tableau[], int taille)` qui à l'aide d'un tableau et de sa taille, renvoie le maximum de ses éléments.

**Exercice 28. Rercherche Dichotomique**

Proposer une implémentation de la recherche dichotomique de prototype `int dichotomie(int tableau[], int taille, int x)`. On cherche $x$ dans le tableau supposé trié, et on renvoie l'indice obtenu ou $-1$ si on ne trouve pas l'élément.

**Exercice 29. Tris classique**

Proposer une implémentation des tris classiques suivants (on pourra se référer aux exercices en OCaml).

1. Tri par insertion, tri par sélection ;
2. Tri à bulle ;
3. Tri rapide, tri fusion ;
4. Bogo-tri.

**Exercice J. Crible d'Ératosthène**

Le crible d'Ératosthène est un algorithme qui permet de trouver les nombres premiers sur une plage donnée.

L'idée est de procéder de la manière suivante :

— On commence avec un tableau de booléens où chaque case correspond avec un entier de 1 à $n$. Si le booléen vaut Faux, le nombre est rayé, et on sait qu'il n'est pas premier.
— On commence par rayer un, puis on parcours le tableau. À chaque fois qu'on rencontre un nombre, et on raye tous ses multiples dans le tableau.
— Une fois qu'on a atteint la fin du tableau, tous les nombres qui n'ont pas été rayés sont premiers.

1. Montrer la correction de cet algorithme.
2. Proposer une fonction `void eratosthene (int n)` qui affiche tous les nombres premiers de 1 à $n$ sur la sortie standard.
3. Le souci d'utiliser un booléen est que les booléens sont stockés sur un octet en C, et on utilise donc 8 fois trop de mémoire pour stocker notre tableau.

   On cherche à représenter le tableau par un `unsigned int *`, mais dont chaque élément permet d'encoder si plusieurs nombres ont été rayés ou pas.

   Par exemple, si on note $b_k$ le booléen qui indique si un nombre est rayé, le premier entier du tableau correspond au tableau suivant, si les entiers sont représentés sur 4 octets :

| $b_1$ | $b_2$ | $\cdots$ | $b_{29}$ | $b_{30}$ | $b_{31}$ | $b_{32}$ |
|---|---|---|---|---|---|---|

   De manière générale, l'entier à la position $k$ du tableau aura les informations suivantes :

| $b_{32k+1}$ | $b_{32k+2}$ | $\cdots$ | $b_{32k+29}$ | $b_{32k+30}$ | $b_{32k+31}$ | $b_{32k+32}$ |
|---|---|---|---|---|---|---|

   (a) En supposant qu'on travaille sur un seul octet, quelles doivent être les valeurs successives du tableau ?
   (b) Proposer une fonction `int trouver(int n)` qui à partir d'un entier $n$ retrouve l'indice $k$ dans le tableau de sorte à ce que l'entier à l'indice $k$ contienne l'information sur $b_n$.
   (c) Proposer une fonction `bool b(int l, unsigned int m)` qui renvoie si le bit à l'indice $32 - l$ est égal à 1.
   (d) Proposer une fonction `bool valeur(int n, unsigned int * tableau)` qui renvoie la valeur de $b_n$ dans le tableau `tableau`.
   (e) Proposer une fonction `void rayer (int n, unsigned int * tableau)` qui raye l'entier à la case `n` dans le tableau `tableau`.
   (f) Proposer une fonction `int prochain(int dernier, unsigned int * tableau)` qui renvoie le prochain nombre strictement plus grand que `dernier` qui ne soit pas rayé dans le tableau `tableau`.
   (g) En déduire une fonction `void eratosthene2 (int n)` qui affiche tous les nombres premiers de 1 à $n$ sur la sortie standard en utilisant cette méthode.

**Exercice 30. Nombre de comparaisons dans le calcul du minimum et le maximum d'un entier sur un tableau**

1. Proposer une fonction de prototype `void min_et_max(int tableau[], int taille , int *min_ptr, int *max_ptr )` qui trouve le minimum et le maximum dans un tableau de taille donné et qui mets cette valeur dans
2. Combien de comparaisons effectue votre fonction sur un tableau de taille $n$ ?
3. On cherche à produire un code qui effectue ce calcul avec un nombre d'opérations qui soit optimal.

   On propose d'utiliser la méthode suivante :

   On compare les deux premiers éléments : le plus grand des deux est notre valeur actuelle pour le max, tandis que le plus petits des deux est notre valeur actuelle pour le min.

À chaque étape on considére les deux prochains éléments qu'il nous reste à voir et on les compare entre eux. Le plus grand des deux est comparé avec le max actuel (et on mets éventuellement à jour le max), et le plus petit des deux est comparé avec le min actuel (et on mets éventuellement à jour le min).

Si le tableau est de taille impaire et qu'il nous reste donc un seul élément à la fin, on le compare avec le min et le max (en les mettant éventuellement à jour).

Proposer une implémentation de cette méthode.

4. Combien de comparaisons effectue cette fonction sur un tableau de taille $n$ ?
5. (⋆)Montrer que ce nombre est optimal.

   *On pourra considérer l'ensemble des candidats pour être le maximum, et l'ensemble des candidats pour être le minimum, et voir ce qui se passe quand on réalise une comparaison.*

**Exercice 31. Dégradation en pointeur**

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

void f(int t[]){
    printf("La taille de t est %d\n", sizeof(t));
    printf("La valeur de t est %o\n", t);
}
int main(){
    int t[100];
    printf("La taille de t est %d\n", sizeof(t));
    printf("L'adresse du premier element est %o\n", &(t[0]));
    f(t);
    return 0;
}
```

Expliquez la sortie de ce code, ainsi que les éventuels messages de mise en garde du compilateur.

**Exercice 32. Arithmétique des pointeurs** (🔧)

L'arithmétique des pointeurs désigne une méthode de programmation qui, en utilisant des opérations arithmétiques sur les adresses, de manipuler les données de la mémoire.

1. Que se passe-t-il lorsque l'on incrémente un pointeur vers un entier de type `int *` à l'aide de l'instruction `ptr++` ?

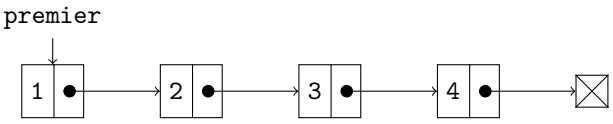
   *On pourra s'intéresser à la différence entre `ptr` et `ptr+1` convertie dans les entiers.*
2. Que se passe-t-il lorsque l'on incrémente un pointeur vers un entier sur un octet de type `int8_t *` à l'aide de l'instruction `ptr++` ?
3. Dans un tableau d'entiers dont le premier élément est à l'adresse donnée par le pointeur `ptr`, quelle est la valeur stockée à l'adresse `ptr++` ?
4. En déduire une fonction qui parcours tous les éléments d'un tableau (par exemple pour les sommer) sans utiliser la syntaxe `t[k]`.
5. Montrer que que `t[k]` est équivalent à `*(t+k)`.
6. En déduire pourquoi `t[k]` est équivalent à `k[t]` en C.

*Dans un certain sens, on dit que l'opérateur [] n'est qu'un sucre syntaxique pour un calcul arithmétique sur les pointeurs.*

## 4.3 Structures chaînées

**Exercice 33. Listes chaînées**

On s'intéresse aux listes chaînées en C.




On utilisera les types suivants :

```c
typedef struct cellule {
    int valeur;
    struct cellule * suiv;
} cell;

typedef cell* liste_chainee;
```

Ainsi, une cellule est un enregistrement qui contient une valeur et un pointeur vers la cellule suivante (potentiellement égal à NULL), et une liste chaînée est un pointeur vers sa première cellule (potentiellement égal à NULL).

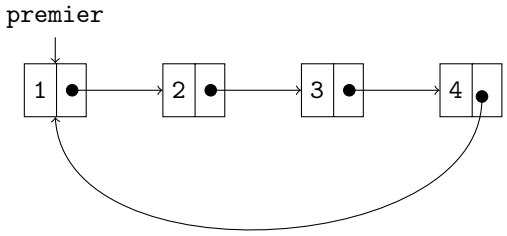
1. Comment construire une liste qui contient les entiers 1, 2 et 3 ?
2. Proposer une fonction de prototype `liste_chainee creer ()` qui crée une liste chaînée vide.
3. Proposer une fonction de prototype `int longueur (liste_chainee lc)` qui renvoie la longueur d'une liste chaînée.
4. Proposer une fonction de prototype `void afficher (liste_chainee lc)` qui affiche le contenu d'une liste chaînée passée en argument.
5. Proposer une fonction de prototype `void ajouter_debut (liste_chainee * plc, int x)` qui ajoute un élément en tête d'une liste chaînée.
6. Proposer une fonction de prototype `void ajouter_fin (liste_chainee * plc, int x)` qui ajoute un élément à la fin d'une liste chaînée.
7. Proposer une fonction de prototype `void retirer_premier (liste_chainee * plc)` qui retire le premier élément d'une liste chaînée.

   *On n'oubliera pas de libérer la mémoire allouée à la cellule ainsi retirée.*
8. Proposer une fonction de prototype `void retirer_dernier (liste_chainee * plc)` qui retire le dernier élément d'une liste chaînée.
9. Proposer une fonction de prototype `void retirer (liste_chainee * plc, int x)` qui retire une cellule de valeur $x$ dans une liste chaînée.

   *On pourra supposer que l'élément existe exactement une fois dans la liste chaînée.*
10. Proposer une fonction de prototype `liste_chainee range (int n)` qui renvoie la liste chaînée dont les éléments dans l'ordre sont les entiers de 0 à $n-1$.
11. Proposer une fonction de prototype `void liberer (liste_chainee lc)` qui détruit une liste chaînée et en libère la mémoire.
12. Proposer une fonction de prototype `void inverser (liste_chainee lc)` qui inverse une liste chaînée passée en argument.
13. Proposer une fonction de prototype `void concatener (liste_chainee lc1, liste_chainee lc2)` qui modifie `lc1` pour qu'elle devienne la liste concaténée de `lc1` et `lc2`.
14. Proposer une fonction de prototype `void modifir (liste_chainee lc, int indice, int valeur)` qui modifie une liste chaînée à l'indice `indice` pour lui donner la valeur `valeur`.
15. Quelle est la complexité de ces fonctions ?

**Exercice 34. Listes cycliques**

On s'intéresse aux listes chaînées cycliques, c'est-à-dire aux listes chaînées dont le dernier élément pointe vers le premier élément.




```c
typedef struct cellule {
    int valeur;
    struct cellule * suiv;
} cell;

typedef cell* liste_chainee;
```

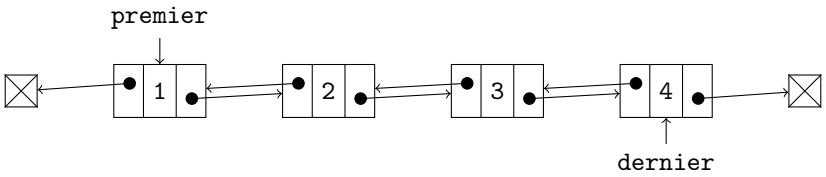
Ainsi, une cellule est une valeur ainsi qu'un pointeur vers une cellule, et une liste est un pointeur vers la première cellule.

Par ailleurs, on remarque que dans une liste non vide, aucun élément ne pointe vers NULL.

Quels sont les changements avec une liste chaînée classique ?

### Exercice 35. Listes doublement chaînées

On s'intéresse aux listes doublement chaînées, c'est-à-dire aux listes là où chaque élément pointe vers le suivant comme le précédent.




```c
typedef struct cellule {
    int valeur;
    struct cellule * suiv;
    struct cellule * prec;
} cell;

typedef struct lc {
    cell * premier;
    cell * dernier;
} liste_chainee;
```

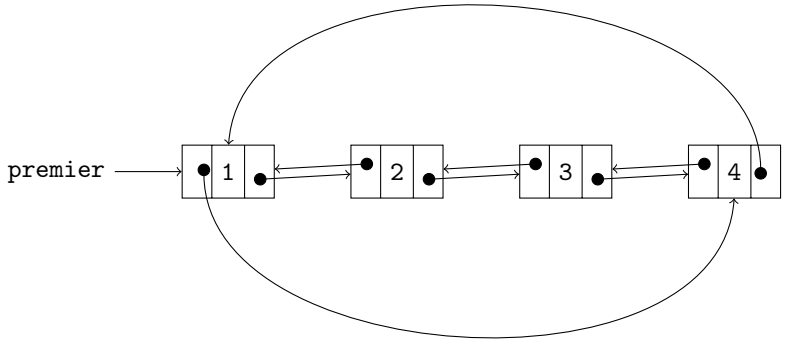
Ainsi, une cellule est une valeur ainsi que deux pointeurs, l'un vers la précédente, et l'autre vers la suivante. Une liste est un pointeur vers la première cellule ainsi qu'un pointeur vers la dernière cellule.

On remarque que si la liste n'est pas vide, le premier élément et le dernier élément pointent vers NULL.

Quels sont les changements avec une liste chaînée classique ?

### Exercice K. Listes doublement chaînées cycliques

On s'intéresse aux listes doublement chaînées cycliques, c'est-à-dire aux listes là où chaque élément pointe vers le suivant comme le précédent, et là où le premier et le dernier pointent l'un vers l'autre.




```c
typedef struct cellule {
    int valeur;
    struct cellule * suiv;
    struct cellule * prec;
} cell;

typedef cell * liste_chainee;
```

Ainsi, une cellule est une valeur ainsi que deux pointeurs, l'un vers la précédente, et l'autre vers la suivante. Une liste est un pointeur vers la première cellule.

Par ailleurs, on remarque que dans une liste non vide, aucun élément ne pointe vers NULL.

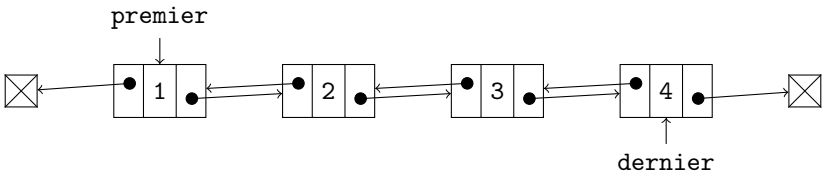
Quels sont les changements avec une liste chaînée classique ?

### Exercice 36. Liste chaînée XOR

Une liste chaînée XOR est une liste doublement chaînée qui, au lieu de sauvegarder l'adresse dans chaque cellule de la suivante et de la précédente, sauvegarde le XOR de ces deux adresses.

Ainsi, l'idée est qu'à partir d'une cellule et de sa précédente dans le parcours d'une liste chaînée, on peut faire un XOR de l'adresse de la précédente et du champs dans la cellule actuelle pour retrouver l'adresse de la cellule suivante.

L'intérêt de cette méthode est d'avoir une liste doublement chaînée avec moins d'espace. Le prix à payer est de ne pas pouvoir faire le parcours depuis une case quelconque sans savoir l'adresse de la précédente.




```c
typedef struct cellule {
    int valeur;
    struct cellule * xoradresse;
} cell;

typedef struct lc {
    cell * premier;
    cell * dernier;
} liste_chainee;
```

Quels sont les changements avec une liste chaînée classique ?

**Exercice 37. Détection de cycle avec l'algorithme du lièvre et de la tortue**

On reprend le type pour créer une liste mutable :

```c
typedef struct cellule {
    int valeur;
    struct cellule * suiv;
} cell;

typedef cell* liste_chainee;
```

On remarque qu'on a pas besoin d'avoir une structure strictement linéaire en rajoutant par exemple des cycles (très justement ce que l'on a fait pour les listes cyclique), et on aimerait pouvoir détecter ces cycles.

Un cycle est une suite de cellule $a_0 ... a_n$ telle que $a_0 = a_n$ et telle que pour tout $0 \leqslant i \leqslant n$ $a_i$ ait pour suivant $a_{i+1}$. Si les $a_i$ sont à deux à deux distincts sauf pour $a_0$ et $a_n$, on dit que ce cycle est de longueur $n$.

Attention : un cycle ne commence pas nécessairement à la première cellule de la liste chaînée.

1. Proposer une implémentation d'un cycle de longueur 1.
2. Proposer une fonction de de prototype `bool a_un_cycle (liste_chainee lc)` qui renvoie si une liste chaînée contient un cycle. On utilisera une liste qui contient toutes les cellules déjà vues depuis le début du parcours.
3. Quelle est la complexité temporelle de la fonction proposée dans le pire des cas en fonction du nombre de cellules accessibles dans la liste chaînée ?
4. Quelle est la complexité spatiale dans le pire des cas en fonction du nombre de cellules accessible dans la liste chaînée ?

Un algorithme un peu plus efficace est l'algorithme du lièvre et de la tortue qui consiste à utiliser deux parcours de la liste en parrallèle, l'un qui avance d'une cellule à la suivante à chaque étape (la tortue), et l'autre qui avance d'une cellule à la suivante de sa suivante à chaque étape (le lièvre). L'algorithme conclut que la liste comprenait un cycle si jamais le lièvre et la tortue arrivent sur la même case simultanément à une étape après la première. Si l'algorithme arrive à la fin de la liste (c'est-à-dire lorsque l'une des cellules suivantes est vide), il conclut qu'il n'y a pas de cycles.

5. On dit qu'une suite $(u_n)$ est ultimement périodique s'il existe $N \in \mathbb{N}$ et $k \in \mathbb{N}$ tels que pour tout $n \geqslant N$, on ait $u_{n+k} = u_n$.

   Ainsi, une suite ultimement périodique est une suite qui est périodique à partir d'un certain rang.

   On considère les éléments dans l'ordre $(c_n)$ des cellules en prenant $c_0$ la première cellule de la liste chaînée, et en prenant $c_{n+1}$ égal à la cellule suivante de la cellule $c_n$.

   Montrer que les $(c_n)$ forment une suite ultimement périodique si et seulement si la liste chaînée comprend un cycle.

   Pour une suite ultimement périodique, on nomme $\lambda$ le plus petit $N$ à partir duquel la suite est périodique, et on nomme $\mu$ le plus petit $k$ tel qu'à partir de ce $\lambda$, on ait $c_{n+k} = c_k$.

   $\lambda$ correspond aux éléments à l'extérieur du cycle, et $\mu$ correspond à la plus petite période dans le cycle, c'est-à-dire au nombre d'éléments dans le cycle.
6. On se donne une suite $(u_n)$ telle que pour tout $i$ et $j$, si $u_i = u_j$ alors $u_{i+1} = u_{j+1}$.

   Montrer que $(u_n)$ est ultimement périodique si et seulement si il existe $n \geqslant 1$ tel que $u_n = u_{2n}$.

7. En déduire que l'algorithme du lièvre et de la tortue est correcte et termine.
8. Montrer que le plus petit $n \geqslant 1$ tel que $u_n = u_{2n}$ noté $n_0$ est majoré par $\lambda + \mu$.
9. En déduire une complexité asymptotique pour la complexité temporelle de l'algorithme du lièvre et de la tortue. Quelle est sa complexité spatiale ?
10. Proposer une implémentation du lièvre et de la tortue qui vérifie s'il y a un cycle dans une liste chaînée.

## 4.4 Caractères, Chaînes de caractères

**Exercice 38. Quelques exercices sur les chaînes de caractère**

On prendra soin de tester les fonctions proposées sur des exemples.

1. Proposer une fonction de prototype `int zip(char * chaine, char c)` qui renvoie le nombre d'occurence de `c` dans `chaine`.
2. Proposer une fonction de prototype `void zip(char * chaine1, char * chaine2)` qui affiche les caractères des chaînes en entrée en alternant entre les deux chaînes.

   Si une chaîne est plus petite que l'autre, on n'arrêtera de lire dans celle-ci, et on affichera tous les caractères restants de l'autre chaîne.

   Par exemple, sur les entrées "Bonjour !" et "Au Revoir !", la sortie suivante sera affichée :

   ```
   BAoun jRoeuvro i!r !
   ```

3. Proposer une fonction de prototype `char * depuis_tableau(char t[], int taille )` qui, à partir d'un

   *On fera attention à conserver une propriété liée aux chaîne de caractère.*

**Exercice 39. Manipulation sur les caractères**

En utilisant des opérations arithmétiques sur les caractères, proposer des fonctions de prototypes et de spécifications suivantes.

1. Une fonction de prototype `char vers_majuscule(char c)` qui renvoie la majuscule associée à une minuscule.
2. Une fonction de prototype `char vers_minuscule(char c)` qui renvoie la minuscule associée à une majuscule.
3. Une fonction de prototype `char * convertir_en_majuscule(const char * str)` qui renvoie une chaîne de caractère identique où les minuscules ont été remplacées par des majuscules.
4. Une fonction de prototype `char * convertir_en_minuscule(const char * str)` qui renvoie une chaîne de caractère identique où les majuscules ont été remplacées par des minuscules.
5. Une fonction de prototype `int vers_chiffre(char c)` qui renvoie la valeur d'un caractère qui représente un chiffre.
6. Une fonction de prototype `int vers_entier(const char *str)` qui renvoie la valeur d'une chaîne de caractère qui représente un entier.

**Exercice L. Numérotation avec des lettres**

On considère le système de numération suivant : $A$, $B$, $C$, ..., $Y$, $Z$, $AA$, $AB$, ..., $AZ$, $BA$, $BB$, ..., $ZZ$, $AAA$, $AAB$, ..., $AAAA$, ....

Proposer une fonction de prototype `char *numeration_lettres(int n)` qui renvoie une chaîne de caractères qui correspond au numéro dans ce système de numération d'un entier $n$.

## 4.5 Autres structures

**Exercice 40. Type Somme en C**

Le C dispose d'un type somme, c'est-à-dire, d'un type dont les valeurs sont un nombre précisés à la définition.

```
enum jour{Lundi, Mardi, Mercredi, Jeudi, Vendredi, Samedi, Dimanche};
```

Ainsi, le type `enum jour` peut prendre exactement 7 valeurs.

1. Proposer une fonction de prototype `void afficher (enum jour j)` qui affiche un jour de la semaine dans la sortie standard.
2. Proposer une fonction de prototype `enum jour suivant(enum jour j)` qui renvoie le jour suivant dans la semaine d'un jour passé en argument.

   *On rappelle que le Lundi suit le Dimanche.*
3. (🔧) En utilisant des conversions vers les entiers, proposer une explication sur la manière dont les types d'énumération sont représentés dans la mémoire.

   On pourra utiliser le code assembleur pour confirmer ces observations.
4. À l'aide de cette remarque, proposer une implémentation de la fonction suivante plus courte.

**Exercice 41. Tableau dynamique en C**

On désire implémenter un tableau dynamique d'entiers en C à l'aide du type suivant :

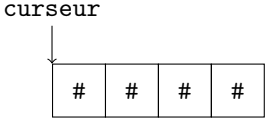
```
typedef struct _dynamique {
    int * memoire;
    int curseur;
    int taille_allouee;
} dynamique
```

Le `curseur` indique le nombre d'éléments affectés dans le tableau, et la `taille_allouee` est la taille de la mémoire totale disponible pour les éléments affectés dans le tableau.

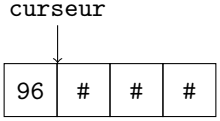
Un tableau dynamique est une structure linéaire à taille variable dans laquelle on peut accéder rapidement aux éléments quelqu'en soit l'indice.

L'idée est la suivante : on dispose d'une mémoire (un tableau) de taille fixe et d'un curseur qui indique jusqu'à où les éléments sont importants.

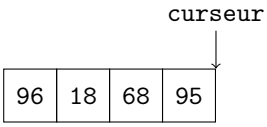
Lorsque le tableau dynamique est vide, aucun élément n'est important, et le curseur pointe vers l'indice 0 (la dièse indique que l'élément peut être n'importe quel élément du bon type) :



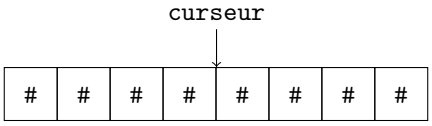

Quand on ajoute un élément, on modifie l'élément sous le tableau, et on déplace le curseur. Ainsi, en ajoutant 96 :



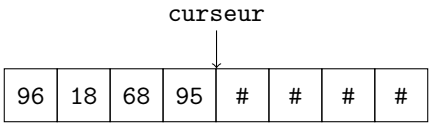
Ainsi, si on ajoute 18, 68 puis 95, on obtient le tableau dynamique suivant :



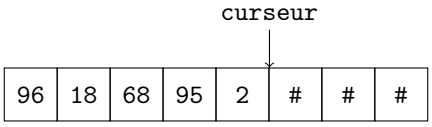

Mais si on doit ensuite ajouter un autre nombre (par exemple 2), il faut modifier le tableau qui sers de mémoire. On crée donc un tableau de taille doublée (si le tableau de la mémoire était vide, on crée un tableau de taille 1) :




On recopie les éléments qui étaient dans le précédent tableau :

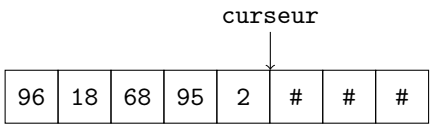



Enfin, on peut rajouter l'élément que l'on voulait ajouter en mettant à jour le curseur :




1. À partir d'un tableau vide, en ajoutant les éléments 4, 8, 15, 16, 23 et 42, à quoi ressemble la mémoire avant et après chaque ajout ?
2. Comment construite un objet du type dynamique avec ces données ?
3. Proposer une fonction de prototype dynamique creer_dynamique() qui construit un nouveau tableau dynamique vide.
4. Proposer une fonction de prototype int taille (dynamique td) qui renvoie la taille d'un tableau dynamique, c'est-à-dire le nombre d'éléments qui sont dans ce tableau dynamique.

   Par exemple la taille du tableau suivant est 5.




5. Proposer une fonction de prototype bool est_vide (dynamique td) qui renvoie si un tableau dynamique est vide.
6. Proposer une fonction de prototype void ajouter(dynamique * td_ptr, int el) qui ajoute un élément dans un tableau dynamique.

   On fera attention à redimensionner le tableau si cela est nécessaire, c'est-à-dire quand le tableau est rempli.

**Exercice M. Retrait dans un tableau dynamique**

On reprend les tableaux dynamiques de l'exercice précédent.

On cherche à réduire la taille de la mémoire quand celle-ci devient occupée au quart : on divise par deux la taille de la mémoire à ce moment là.

1. Pourquoi faire le changement de taille mémoire quand le tableau est occupé au quart et pas à moins de la moitié ?
2. Proposer une implémentation de prototype `int retirer_dernier(dynamique * td_ptr)` qui retire le dernier élément du tableau et renvoie sa valeur en changeant éventuellement la taille de la mémoire.
3. Montrer que pour toute succession de $n$ ajouts et de $n$ retraits, la complexité totale de ces $2n$ opérations est en $O(n)$, quelque soit l'ordre.

## 4.6 Tableaux multidimensionnels

**Exercice N. Tableaux multidimensionnels de taille statique**

En C, on peut déclarer des tableaux multidimensionnels de taille statique à l'aide de la syntaxe suivante :

```
int tableau[x][y];
```

Ici, $x$ et $y$ doivent être des constantes ou des littéraux (c'est-à-dire des entiers en chiffres).

Par exemple, on peut déclarer un tableau de taille $4 \times 3$ à l'aide de la syntaxe :

```
int tableau[4][3];
```

On peut normalement modifier et accéder à ses éléments à l'aide des syntaxes suivantes :

```
tableau[2][1] = 4;
```

```
printf("La valeur a la case (2,1) est %d\n", tableau[2][1]);
```

L'idée derrière cette syntaxe est qu'un tableau multidimensionel est un tableau de tableaux.

1. ...

**Exercice O. Linéarisation de tableaux multidimensionnels**

# 5 Fichiers, Entrées, Sorties

**Exercice 42. Lire et écrire dans un fichiers**

Dans chacune des fonctions suivantes, on pourra suposer que les entiers écrits ne dépassent pas 10 caractères.

1. Proposer une fonction de prototype `void liste_entier(int n)` qui écrit les entiers de 0 à $n-1$ dans un fichier en les séparant par des sauts de lignes.
2. Proposer une fonction de prototype `int somme_entiers(char *nom)` qui lit un fichier de nom `nom` composés d'entiers séparés par des espaces et renvoie la somme de ces entiers.
3. Proposer une fonction de prototype `void ecrire_maximum(char *nom)` qui lit un fichier de nom `nom` composés d'entiers séparés par des espaces et écrit à la fin de ce fichier sur une nouvelle ligne le maximum des entiers rencontrés.

**Exercice 43. Quine en C et en OCaml**

Un **quine** est un programme en informatique dont la sortie est le contenu du programme lui-même.

La plupart des quines dans les langages de programmations fonctionnent d'une manière similaire : on construit une chaîne de caractère que l'on affiche deux fois en jouant avec les différentes subtilités du langage pour recoller les caractères avant et après.

1. Que doit-on mettre dans `**A REMPLACER**` pour que le code suivant soit un quine en C ?

```c
#include <stdio.h>
int main(){
char*a="**A REMPLACER**";
printf(a,10,10,34,a,34,10,10);
}
```

*Le caractère de code* 10 *est le caractère de retour à la ligne, tandis que le code* 34 *correspond aux guillemets doubles droits :* ".

2. Que doit-on mettre dans `//A REMPLACER//` pour que le code suivant soit un quine en OCaml ?

```ocaml
(fun s -> Printf.printf "%s %S;;" s s) "//A REMPLACER//";;
```

*Le caractère* %s *premet d'insérer une chaîne de caractère dans une chaîne caractère, et* %S *permet d'insérer une chaîne de caractère dont les caractères ont été modifiés pour correspondre à la syntaxe OCaml.*

3. Proposer un programme en C et en OCaml qui affiche deux fois son code source.

### Exercice 44. Quine dans un langage minimaliste

Un **quine** est un programme informatique dont la sortie est le contenu du programme lui-même.

Dans cet exercice, nous construisons un langage minimaliste pour lequel nous construisons un interpréteur, puis pour lequel nous construisons un quine.

On se donne un langage dont la syntaxe est décrite de la manière suivante :
— Chaque ligne comprend exactement une instruction ;
— Chaque instruction est soit `afficher M` où $M$ est un entier, soit `recopier M` où $M$ est un entier.

Lors de l'exécution d'un tel programme, les lignes sont interprétées dans l'ordre de la manière suivante :
— `afficher M` affiche les $M$ lignes suivantes du programme dans la sortie et les ignore dans l'exécution ;
— `recopier M` affiche les $M$ dernières lignes qui ont été affichées.

Ainsi, le code suivant :

```
afficher 0
afficher 1
afficher 2
recopier 1
```

a pour sortie :

```
afficher 2
afficher 2
```

Le code suivant :

```
afficher 3
afficher 0
afficher 1
recopier 1
recopier 2
```

a pour sortie :

```
afficher 0
afficher 1
recopier 1
afficher 1
recopier 1
```

On remarque que les codes et les sorties ont la même syntaxe.

1. On commence par quelques propriétés sur le langage.
   (a) Montrer que pour toute sortie, il existe un programme qui a cette sortie.
   (b) Certains programmes n'ont cependant pas de sortie dans le cas où le programme n'est pas valide.

       Proposer un programme qui n'est pas valide dans ce langage.
   (c) On note $u_n$ le nombre de lignes maximum de la sortie d'un code de $n$ lignes.
       Combien vaut $u_0$ ? $u_1$ ? $u_2$ ? $u_3$ ?
   (d) Montrer que pour tout $n \geqslant 1$, $u_{n+1} \geqslant 2u_n$.
   (e) On considère un programme $P$ de longueur $n+1$ tel que sa sortie soit de taille $u_{n+1}$. On suppose que la dernière ligne du programme est exécutée et qu'il s'agit d'un `recopier`. Montrer que $u_{n+1} = 2u_n$.
   (f) (⋆) Montrer que pour tout $n \geqslant 2$, $u_{n+1} = 2_u n$.
   (g) En déduire une forme explicite pour $u_n$.

2. Pour représenter un programme ou une sortie, on utilise une énumération pour choisir entre `afficher` et `recopier`.

```c
enum fonction_t {A, R};
```

   On utilise une commande qui est une fonction et un entier qui correspond à l'argument de la fonction.

```c
typedef struct _commande {
    enum fonction_t f;
    int arg;
} commande_t;
```

   Enfin, un programme est un tableau de commande auquel on a rajouté la taille de ce tableau :

```c
typedef struct _programme {
    commande_t * commandes;
    int taille;
} programme_t;
```

   (a) Proposer une fonction de prototype `int taille_sortie (programme_t p)` qui calcule la taille de sortie d'un programme en entrée.
   (b) En déduire une fonction de prototype `programme_t executer (programme_t p)` qui calcule la sortie d'un programme en entrée.
   (c) Proposer une fonction de prototype `void afficher (programme_t p)` qui affiche un programme en entrée.
   (d) Proposer un programme C tel qu'il exécute le programme sous la forme d'un fichier passé en argument.

       *On pourra supposer que les arguments des programmes sont de longueur au plus 2.*

3. On cherche désormais à construire un quine sur ce langage.
   (a) Quelle est la sortie du code suivant ?

```
afficher 4
recopier 4
afficher 4
recopier 4
afficher 4
recopier 4
```

   (b) Proposer un code dont les deux dernières lignes sont les suivantes :

```
...
afficher 4
recopier 4
```

       Et qui a pour sortie lui-même sauf sa dernière ligne.

(c) En déduire un quine sur ce langage.

*Cette construction d'un quine est liée aux fichiers zip qui se décompressent en eux-même.*

### Exercice 45. Un code offusqué (🔧)

Le code suivant doit être compilé de sorte à ce que le nom du programme soit Fizz_Buzz.

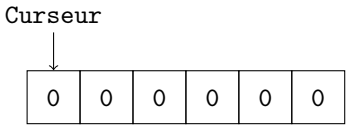
```c
#include <stdio.h>
#include <stdlib.h>

int main (int argc, char* argv[])
{
    return (((*argv)[4] = 0) || ( (!(argc%3) && printf(*argv)) + (!(argc%5) &&
    printf((*argv)+5))) || printf("%d", argc)) != printf("\n")!= argc++ < 100 &&
    main(argc, argv);
}
```

1. Compiler et exécuter ce code.
2. Comment fonctionne ce code ?
3. Proposer une implémentation décente de ce code.

### Exercice 46. Compilation vers C du langage BF

Le langage BF est un langage exotique qui est composé de huits instructions et qui opère sur une bande mémoire.

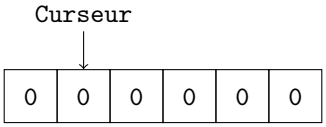
Initialement, la bande mémoire est composée intrégralement d'entiers égaux à 0. On suppose qu'elle est de taille $n$ fixe. Initialement, un curseur pointe vers l'élément d'indice 0.




Les entiers sont supposés positifs.

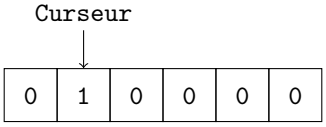
Un programme BF est une chaîne de caractère qui contient les instructions suivantes. Lors de l'exécution on lit les caractères dans l'ordre jusqu'à arriver à la fin du programme.

> déplace le curseur vers la droite d'une case. Ainsi, depuis la situation précédente, on obtient la disposition suivante :




< réalise l'opération inverse, décalant le curseur vers la gauche d'une case.

\+ incrémente la case indiquée par le curseur de 1. À partir de la situation précédente, on obtient la situation suivante :



– réalise l'opération inverse qui décrémente la case indiquée par le curseur de 1.

. imprime sur la sortie standard le contenu de la case indiquée.

[ et ] permettent de faire des boucles. Le crochet ouvrant saute au crochet fermant associé (en terme de parenthèsage) si la valeur de la case sous le curseur est 0, et le crochet fermant saute au crochet ouvrant associé si la valeur de la case sous le curseur est différente de 0.

, demande une entrée et met le résultant dans la case indiquée par le curseur.

*Usuellement, les entrées et sorties en BF sont codés en ASCII. Dans notre cas, on affichera simplement les nombres tels quels, et de même pour récupérer des informations.*

Par exemple le programme BF ,[–>+<]>. lit un nombre en entrée, fait une boucle pour le déplacer dans la case suivante (en mettant la première case à 0), puis renvoie cette valeur.
Le programme ,>,<[–>+<]>. lit deux entiers mis dans les deux première case, puis augmente la deuxième case d'une valeur égale à la première puis renvoie la somme des entiers.

On cherche à compiler le langage BF vers le C, c'est-à-dire que, à partir d'un fichier contenant un code en langage BF, on cherche à construire le code qui fait la même chose en langage C.

1. On cherche à ce que les éléments de la mémoire soit nécessairement des entiers positifs représentés sur un octet. Et que la mémoire elle-même soit un tableau de tels octets.

   Quel type doit on utiliser pour chaque case mémoire ? Quel type doit on utiliser pour la mémoire ?
2. Si on suppose qu'on a accès à un tel tableau, ainsi qu'à un entier qui correspond à la position actuelle du curseur, comment peut-on compiler les instructions $<$, $>$, $+$ et $-$ ?
3. Comment peut-on compiler les instructions , et . ?
4. Comment peut-on compiler les instructions [ et ] ?
5. Proposer un code C qui transforme une chaîne de caractère qui est un programme BF vers le C.

   On pourra supposer qu'il existe une constante définie en début du code produit qui définit la taille du tableau de la mémoire.

```c
const int taille_memoire = 16
```

6. Proposer un code C qui compile le langage BF vers le C.

### Exercice P. Cacher du texte dans une image (🔧)

Le format d'image PPM est un format de fichier d'image minimaliste qui permet de stocker des images dont chaque pixel est stocké au format RGB.
L'objectif de l'exercice est de pouvoir cacher un message secret dans les bits de poids faible des pixels de l'image. En n'altérant que le bit de poids faible pour chaque valeur de couleur, on va altérer de manière minimaliste l'image.
*Vous aurez besoin d'ouvrir le fichier en mode binaire.*

1. Proposer une fonction de prototype `void cacher(char *nom, char *message)` qui modifie une image au format PPM pour que la suite des bits de poids faibles de chacune des valeurs RGB des pixels dans l'ordre de l'image devienne l'écriture en binaire de la chaîne message.
2. Proposer une fonction de prototype `void reveler(char *nom)` qui affiche le message caché dans l'image dont le nom est passé en argument.

**Exercice Q. Ensemble de Mandelbrot** (🔧)

On définit l' ensemble de Mandelbrot comme l'ensemble des $c \in \mathbb{C}$ tels que la suite définie par $z_0 = 0$ et $\forall n \in \mathbb{N}, z_{n+1} = z_n^2 + c$ soit bornée.

Pour représenter les complexes, on utilise la structure suivante :

```
typedef struct _complexe {
    double re ;
    double im ;
} complexe;
```

1. Montrer que s'il existe $n_0$ tel que $|z_{n_0}| > 2$, alors la suite $z_n$ diverge.
2. On souhaite donc procéder de la manière suivante pour calculer une approximation de l'ensemble de Mandelbrot : on prend un nombre d'itération maximale :

   ```
   const int max_iter = 100;
   ```

   Pour déterminer si un nombre complexe $c$ est dans notre approximation de l'ensemble de Mandelbrot, on calcule un nombre de termes de la suite $z_n$ au plus égal à max_iter, et on dit que l'élément est dans notre approximation de l'ensemble de Mandelbrot si et seulement si lors de ce calcul, il n'existe pas de $n_0$ tel que $\text{abs}(z_{n_0}) > 2$.

   Proposer une fonction qui détermine si un complexe est dans l'approximation de l'ensemble de Mandelbrot.
3. En utilisant la bibliothèque graphics.h et en particulier la fonction putpixel, proposer une implémentation qui affiche une approximation de l'ensemble de Mandelbrot.

   *Il est possible que la bibliothèque graphics ne soit pas présente sur la machine sur laquelle vous travaillez. Dans ce cas, vous pouvez ignorer cette question.*
4. Sur la plupart des représentations de l'ensemble de Mandelbrot, on utilise une couleur qui correspond à la vitesse à laquelle la suite des $z_n$ diverge en utilisant un gradient.

   Modifiez votre code pour proposer un gradient de couleur dans la représentation de l'approximation de l'ensemble de Mandelbrot.

# 6 Système

**Exercice R. Manipulation de base du terminal**

1. Créer un fichier fichiera à l'aide de la commande mkdir.
2. Se déplacer dans ce dossier à l'aide de la commande cd.
3. À l'aide de l'éditeur de texte nano créer un fichier et écrivez un programme C qui affiche "Bonjour" sur la sortie standard.
4. À l'aide de la commande cat, afficher le contenu du fichier.
5. Compiler le programme à l'aide de gcc.
6. Executer le programme compilé en redirigeant la sortie vers un fichier a.out.
7. Afficher le contenu du fichier a.out à l'aide de la commande cat.

**Exercice S. Chemins en UNIX**

**Exercice T. Manipulation de fichiers**

**Exercice U. Utilisation d'un tube pour chaîner des programmes**

**Exercice V. Implémentation de commandes de base en C**