INF 03105 – Programmation
TP n° 1 : Initiation à Git et C

Les notes de cours sont à l'adresse suivante :
https://perso.ens-lyon.fr/francois.schwarzentruber/teaching/l3-prog/book/welcome.html

I) Initialisation d'un dépôt Git

Tout au long du semestre, vous allez utiliser Git pour sauvegarder et synchroniser les programmes que vous écrivez. Nous allons configurer Git ensemble, car cela est un peu long et déroutant quand on ne l'a jamais fait. Git est un outil central dans le monde de l'informatique, vous en aurez très souvent besoin, autant apprendre à s'en servir le plus tôt possible. Voici les grandes étapes à réaliser :

1. Se connecter à la palteforme `git` d'Aliens (dans votre vie future, il s'agira peut-être du `gitlab` de l'Inria, celui du CNRS, de `github`, ou d'une autre plateforme basée sur Git)
2. Créer un projet
3. Ajouter vos encadrant·es au projet
4. Ajouter un fichier via l'interface graphique
5. Générer une clé RSA dans votre répertoire personnel
6. Ajouter cette clé au `git` d'AliENS
7. Cloner le dépôt
8. Synchroniser le dépôt

Voici un guide un peu plus détaillé de ce qu'il faut faire à chacune de ces étapes :

1. Se connecter au `git` d'Aliens :
   - Allez sur `https://git.aliens-lyon.fr`.
   - Cliquez sur Sign In (en haut à droite).
   - Cliquez sur Sign In with AliENS.
   - Connectez-vous avec votre compte ENS.
2. Créer un projet :
   - Cliquez sur le `+` en haut à droite (Create...) puis New repository.
   - Rentrez le nom du projet dans Repository name (par exemple, `tp-prog`).
   - Mettre Visibility Level à Private.
   - Validez en cliquant sur Create Repository.
3. Ajouter vos encadrant·es au projet :
   - Cliquez sur Settings puis Collaborators, dans le menu à gauche.
   - Ajoutez vos encadrants avec cette interface
   - Les pseudos de vos encadrant·es sont `alaouar`, `mnicolis` et `fschwarz`.
4. Ajouter un fichier via l'interface graphique :
   - Cliquez sur Upload File dans l'onglet Code du Repository.
   - Choisissez le fichier à ajouter.
   - Remplissez Commit Message avec une description succincte du fichier ajouté (par exemple, "Premier TP de Prog").
   - Cliquez sur Commit Changes.

Maintenant, le but est de configurer Git pour pouvoir ajouter et mettre à jour des fichiers en ligne de commande.

5. Générer une clé RSA dans votre répertoire personnel :
   - Dans un terminal de commande, tapez `ssh-keygen -t rsa`
   - Appuyez sur entrer pour que la clé soit créée dans l'emplacement par défaut (`~/.ssh/id_rsa`). Si vous en avez déjà une, il vous demandera si vous voulez l'écraser (overwrite) – dites non ! Si vous n'en avez pas, il vous demande un mot de passe (passphrase) ; vous pouvez appuyer sur Entrée pour ne pas en mettre. (Vous devez valider le mot de passe ou l'absence de mot de passe une seconde fois).

6. Ajouter cette clé au `git` d'AliENS :
   - Cliquez sur votre image de profil en haut à droite (une figure géométrique stylisée en couleur) puis Settings.
   - Cliquez sur SSH/GPG Key dans le menu à gauche.
   - Cliquez sur Add key.
   - Donnez un nom à la clé (par exemple "Ordinateur Portable")
   - Tapez `cat ~/.ssh/id_rsa.pub` dans un terminal et copiez-collez ce qui s'affiche dans Content.
   - Validez avec Add Key.

Enfin, on va pouvoir utiliser Git en ligne de commandes depuis un terminal.

7. Cloner le dépôt :
   - Retournez à votre projet dans la page `git` (cliquez sur le logo en haut à gauche, puis sur le nom du projet).
   - Dans l'onglet Code copiez l'*adresse du projet*, qui est à côté du bouton SSH, qui doit être activé (soit à la main, soit en cliquant sur le bouton de copie à droite). C'est de la forme :
     `ssh://git@git.aliens-lyon.fr:"votre_nom"/"nom_du_projet".git`
   - Dans un terminal, à l'endroit où vous voulez mettre votre répertoire de TP, tapez :
     `git clone "adresse_du_projet"`
     L'adresse du projet est ce que vous venez de copier-coller. Cette commande permet de créer un nouveau dossier, qui contient tout ce qui est sur votre projet sur `gitlab`, et ce dossier pourra être synchronisé facilement.
8. Synchroniser le dépôt :
   - Dans le dossier créé par le clonage, créez un fichier vide, juste pour tester.
   - Dans un terminal, toujours dans le même dossier, tapez `git add -a`
   - Tapez `git commit -m "message"`
     La première fois, Git n'est pas content, faites ce qu'il dit (vous devez renseignez votre nom et votre adresse mail).
   - Tapez `git push`

   À tout moment, si vous ne savez plus où vous en êtes avec Git, vous pouvez taper `git status`

La prochaine fois, nous verrons comment récupérer le contenu synchronisé *sans perdre tout son travail*. En effet, il est possible de faire des fausses manipulations et de perdre du travail qui n'est pas encore synchronisé avec Git. Nous ferons tout pour éviter ça.

II) Exercice d'introduction au C

Exercice 1 : Comprendre le C.

Voici un exemple de programme écrit C :

```c
#include <stdio.h>

int somme(int tableau[], int taille){
    int i;
    int accumulateur;
    accumulateur = 0;
    for (i = 0; i < taille; i++) {
        accumulateur = accumulateur + tableau[i];
    }
    return accumulateur;
}

int main(){
    int notes[5] = {12,18,15,13,20};
    int total = somme(notes, 5);
    printf("Le total est : %d. Au revoir.\n", total);
    return 0;
}
```

1. Recopiez ce programme dans un éditeur, compilez ce programme puis exécutez-le.
2. Supprimez la ligne 1 et essayez de compiler. Ça ne fonctionne pas. Déduisez-en à quoi sert cette ligne.
3. D'après cet exemple, comment fait-on pour déclarer une fonction en C ? En particulier, comment précise-t-on le type de retour de la fonction ?
4. À quoi servent les lignes 4 et 5 ?
5. Modifiez la ligne 15 en remplaçant le nombre 5 par 4. Essayez ensuite les valeurs 6 et 123456. Avez-vous une idée de ce qui se passe ?
6. Que fait la ligne 16 ?
7. En vous inspirant de la fonction `somme`, écrivez une fonction `affiche` qui affiche le contenu d'un tableau. En C, pour écrire une fonction qui ne renvoie rien, on utilise le type `void`. La déclaration de la fonction sera donc `void affiche(int tableau[], int taille)`.

III) Exercices intermédiaires

Exercice 2 : Le crible d'Ératosthène.

On rappelle qu'un entier naturel p est dit *premier* s'il n'existe aucun entier naturel autre que 1 et p lui-même qui le divise. Un nombre qui n'est pas premier est *composé*. Par convention, 1 n'est ni premier ni composé. Dans cet exercice, nous écrirons une fonction calculant la liste de tous les nombres premiers plus petits qu'un certain nombre n donné en entrée. Pour ce faire, on utilisera une méthode connue sous le nom de *crible d'Ératosthène*, qui consiste à déterminer les nombres premiers par filtrages successifs.
L'idée est de repérer les nombres composés et de déduire les nombres premiers par complémentarité : pour chaque nombre k supérieur ou égal à 2, on marque chaque multiple de k autre que k lui-même comme étant *composé*. En faisant ça jusqu'à une certaine borne, on déduit tous les nombres premiers inférieur ou égaux à n.

1. Écrivez un programme `eratosthene` qui prend en entrée un entier `n` et affiche tous les nombres premiers de 2 à n calculés en appliquant le crible d'Ératosthène. N'hésitez pas à factoriser votre programme en plusieurs fonctions auxiliaires.
2. Modifiez votre programme pour qu'il affiche aussi la fonction $\pi(n)$ de compte des nombres premiers, c.à-d. le nombre de nombres premiers strictement inférieurs à `n`.[1]

Exercice 3 : Calculatrice.

On veut écrire un programme `calc_sum` qui, lorsque l'utilisateur rentre une ligne contenant des entiers (en représentation décimale) séparés par des symboles `+`, comme `"7+52+3\n"` ou `"2+6+74+13\n"`, le programme affiche le résultat de la somme. Si par contre la ligne rentrée n'a pas le bon format, `calc_sum` doit se terminer en affichant `"Bye."`. Par exemple, si on rentre la chaine `"3+8\n"` le programme affiche `"11"` et attend une nouvelle ligne, mais si on rentre la chaine `"+7+8\n"` ou `"4+7+8+\n"` le programme termine en affichant `"Bye."`.

1. Écrivez le programme `calc_sum`.
2. Modifiez le programme `calc_sum` afin qu'il accepte les nombres décimaux.

1. Estimer de façon aussi précise que possible ce nombre $\pi(n)$ est un des problèmes centraux de la branche des mathématiques appelée *théorie des nombres*. Par exemple, si vous arrivez à démontrer que $\left|\pi(n) - \int_0^n \frac{dt}{\log t}\right| < \sqrt{n}\log n$ pour tout $n \geqslant 2$, vous aurez atteint la gloire éternelle (ainsi que gagné 1 million de dollars). Du point de vue numérique, nous avons vu comment calculer "rapidement" $\pi(n)$ pour n de l'ordre de millions (essayez `eratosthene(1000000)` et regardez combien de temps cela prend). Pour aller au delà, il faut des méthodes plus sophistiquées.

IV) Exercices avancés

Exercice 4 : Labyrinthes

On veut créer un algorithme aléatoire qui puisse générer des labyrinthe tels que celui ci-dessous :

```
XXXXXXXXXXXXXXX
X X     X     X
X XXX X X X XXX
X X   X X X   X
X X XXXXX XXX X
X X   X     X X
X X X X XXXXX X
X X X X X   X X
X XXX X X X X X
X X   X X X   X
X X XXX X XXX X
X X     X X X X
X XXXXXXX X X X
X         X   X
XXXXXXXXXXXXXXX
```

Pour ce faire, on va produire un tableau 2D nommé A, de taille $n \times m$, où n et m sont des entiers impairs. On impose les contraintes suivantes :

- chaque cellule $A[i, j]$ vaut `'X'` ou `'␣'`,
- pour tout i, j, les cellules $A[0, j], A[i, 0], A[n-1, j]$ et $A[i, m-1]$ contiennent `'X'`,
- si i et j sont tous les deux pairs, on a $A[i, j] =$ `'X'`,
- si i et j sont tous les deux impairs, on a $A[i, j] =$ `'␣'`,
- le graphe G est un arbre, où G est le graphe non orienté (V, E), avec V qui est l'ensemble des couples (i, j) dans $\{1, 3, \ldots, n-2\} \times \{1, 3, \ldots, m-2\}$ (c'est-à-dire tels que i et j sont tous les deux impairs) et E qui contient les arêtes $\{(i, j), (i', j')\}$ telles que :
  - $i' = i + 2$ et $j' = j$ et $A[i + 1, j] =$ `'␣'`,
  - ou $i' = i$ et $j' = j + 2$ et $A[i, j + 1] =$ `'␣'`.

1. Vérifiez que l'exemple donné ci dessus respecte bien les contraintes énoncées.

2. Écrivez un programme qui permet de générer de tels labyrinthes.

Exercice 5 : Un peu de Caml en C.

On veut implémenter certaines fonctionnalités de Caml en C. Une des fonctions phares de Caml est la fonction `map` de signature `('a → 'b) → 'a list → 'b list` qui applique une fonction à l'ensemble des éléments d'une liste. On se propose d'implémenter une variante de cette fonction, qui applique une liste de fonctions à un même élément. Sa signature en Caml serait `('a → 'b) list → 'a → 'b list`. On va implémenter cette fonction dans le cas `'a = 'b = int`.

1. Écrivez une fonction qui prend en argument une fonction f : `int → int` ainsi qu'un entier n et qui renvoie $f(n)$. En C, pour déclarer un objet f de type `int → int`, on déclare `"int␣(*f)(int);"`.

2. Définissez une structure de liste chainée. On aura à la fois besoin de listes chainées d'entiers et de liste chainées de fonctions de type `int → int`. Faites en sorte que la même structure puisse être utilisée dans les deux cas en utilisant `void*`.

3. Écrivez les fonctions classiques sur les listes chainées, à savoir la création d'une liste vide, l'ajout d'un élément en tête, le test qui renvoie si une liste est vide, la fonction qui récupère l'élément en tête de liste et celle qui renvoie la queue de la liste.

4. Écrivez une fonction *récursive* qui prend en paramètre une liste chainée L de fonctions $L[i]$: `int → int` ainsi qu'un entier n et qui renvoie la liste chainée des applications de chaque fonction $L[i]$ à n.

Conseils : D'abord, lisez toujours un énoncé jusqu'au bout, au cas où il y aurait des indications à la fin. Ensuite, pour ce genre d'exercice un peu déroutant, il ne faut pas hésiter à revenir vers des choses connues : si les listes chainées de `void*` vous impressionnent, commencez avec des listes chainées d'entiers. De même, vous pouvez commencer par implémenter `map` (qui n'utilise que des listes chainées d'entiers).

INF 03105 – Programmation
TP n° 2

Avant de commencer...

L'exercice 5 nécessite l'installation d'une librairie appelée `raylib`, ce qui peut prendre du temps. Allez jeter un œil maintenant et commencez la procédure d'installation, vous ferez les (vrais) premiers exercices pendant les temps de téléchargement ou de compilation.

I) Exercices d'introduction

Exercice 1 : Types structurés.

Il est possible de créer ses propres types en C. Voici un exemple avec les entiers de Gauss, qui sont des nombres complexes dont les parties réelle et imaginaire sont toutes les deux un entier :

```
1  struct gaussint {
2      int re;
3      int im;
4  };
5
6  struct gaussint addition(struct gaussint z_1, struct gaussint z_2){
7      struct gaussint z;
8      z.re = z_1.re + z_2.re;
9      z.im = z_1.im + z_2.im;
10     return z;
11 }
```

Notez le point-virgule après l'accolade à la ligne 4. Le type créé se note `struct gaussint`. Notez que ce type est constitué de deux mots, là où les types habituels (`int`, `float`, etc.) n'en ont qu'un. Pour gagner de la place et du temps, on peut *renommer* un type. Ainsi, le code `typedef struct gaussint gi;` permet de créer le type `gi` qui est équivalent au type `struct gaussint`, mais qui est bien plus rapide à écrire.

1. Implémentez une fonction qui renvoie le résultat de la multiplication de deux entiers de Gauss.
2. Implémentez une fonction qui affiche un entier de Gauss.
3. Implémentez une fonction qui calcule la somme d'un tableau d'entiers de Gauss.
4. *optionnel, n'y passez pas trop de temps* Implémentez une fonction qui prend en argument une chaine de caractère et qui renvoie l'entier de Gauss correspondant. Par exemple, si l'argument est `"2+3i"`, la fonction renvoie l'entier de Gauss z tel que `z.re = 2` et `z.im = 3`.

Exercice 2 : Pointeurs

Les pointeurs permettent de manipuler l'adresse mémoire des variables. Si `x` est une variable, `&x` est l'adresse mémoire de `x`. À l'inverse, si `ptr` est un pointeur vers une adresse mémoire, `*ptr` représente le contenu de l'adresse mémoire `ptr`. Voici un exemple :

```
1  #include <stdio.h>
2
3  int main() {
4      int n = 17;
5      printf("La␣valeur␣de␣n␣est␣:␣%d\n", n);
6      printf("L'adresse␣de␣n␣est␣:␣%p\n", &n);
7      int* ptr = &n;
8      printf("L'adresse␣pointee␣par␣ptr␣est␣:␣%p\n", ptr);
9      printf("Le␣contenu␣de␣l'adresse␣pointee␣par␣ptr␣est␣:␣%d\n", *ptr);
10     return 0;
11 }
```

1. Exécutez ce programme puis faites un dessin contenant `x`, `ptr`, la valeur de ces deux variables et l'adresse de `x`.
2. Exécutez à nouveau ce programme. Normalement, l'adresse de `x` a changé. Savez-vous pourquoi ?
3. Modifiez le programme pour qu'il affiche l'adresse de `ptr`.

Exercice 3 : Renvoyer deux valeurs en C.

On veut écrire une fonction `somme_moyenne` qui prend en argument une liste d'entiers et qui renvoie deux valeurs : un entier qui est la somme des entiers du tableau, ainsi qu'un nombre décimal qui est la moyenne des entiers du tableau.

1. Créez un type qui contient un entier et un flottant, puis programmez la fonction `somme_moyenne`. Testez-la sur un exemple.

Cette approche est en fait très mauvaise. Le type ainsi créé ne sert que pour une seule fonction. La façon classique de faire est la suivante : `somme_moyenne` ne renvoie rien, mais prend deux arguments en plus : il s'agit de deux pointeurs qui permettront de stocker les deux résultats. Elle a donc la déclaration suivante :
`void somme_moyenne(int tableau[], int taille, int* ptr_somme, float* ptr_moyenne)`
La fonction `somme_moyenne` va modifier directement les valeurs pointées par `ptr_somme` et `ptr_moyenne` et elle sera appelée de la façon suivante :

```
int main() {
    int tableau[5] = {15,18,20,13,16};
    int somme;
    float moyenne;
    somme_moyenne(tableau, 5, &somme, &moyenne);
    printf("La somme est %d et la moyenne est %f\n", somme, moyenne);
    return 0;
}
```

2. Programmez à nouveau la fonction `somme_moyenne`. Testez-la sur l'exemple ci-dessus.

II) Exercices intermédiaires

Le but de ces exercices est de programmer le jeu de la vie de Conway avec un affichage dans le terminal, puis d'installer une librairie d'affichage graphique appelée `raylib` afin d'avoir un affichage joli de ce jeu.

Exercice 4 : Le jeu de la vie, version texte.

Le jeu de la vie est un jeu sans aucun joueur (c'est une simulation). Il se déroule sur une grille 2D de taille théoriquement infinie. Dans notre cas, on se limitera à une grille de taille 60 par 60. Chaque case de la grille est appelée cellule. Elle peut être morte ou vivante. La simulation passe de l'instant t à l'instant $t+1$ via les trois règles suivantes :

- Si à l'instant t, une cellule est morte et possède parmi ces huit cellules voisines exactement trois cellules vivantes, elle devient vivante à l'instant $t+1$.
- Si à l'instant t, une cellule est vivante et possède parmi ces huit cellules voisines un nombre de cellules vivantes différent de 2 ou 3, elle devient morte à l'instant $t+1$.
- Dans tous les autres cas, la cellule ne change pas d'état entre l'instant t et l'instant $t+1$.

Avant d'implémenter ce jeu, on se pose quelques questions pratiques. Tout d'abord, puisque notre simulation ne se déroule pas sur une grille infinie, il va falloir gérer les bords, notamment en ce qui concerne l'étude des huit cellules voisines.

1. Proposez une idée d'implémentation de la grille pour ne pas avoir à gérer les bords.

Ensuite, plutôt que de garder en mémoire tous les instants de 1 à t, on ne conserve que le dernier instant t, qui suffit à construire l'instant $t+1$. On a donc besoin de ne conserver que deux grilles en mémoire. Cependant, vous serez amener à devoir échanger leur contenu.

2. Comment faire pour "échanger le contenu" de deux tableaux de taille arbitraire en temps constant ? On rappelle que la syntaxe pour déclarer un pointeur p vers un tableau d'entiers est `int (*p)[]`.

3. Implémentez le jeu de la vie. On passera de l'instant t à l'instant $t+1$ en appuyant sur la touche entrée (cela peut se faire en appelant `getchar()` et en ignorant le résultat).

Exercice 5 : Raylib

Installez la librairie `raylib`. Pour ce faire, suivez le tutoriel correspondant à votre OS :
- Windows : `https://github.com/raysan5/raylib/wiki/Working-on-Windows`
- Linux : `https://github.com/raysan5/raylib/wiki/Working-on-GNU-Linux`
- Mac : `https://github.com/raysan5/raylib/wiki/Working-on-macOS`

Une fois la libraire installée, considérons le programme minimal ci-dessous :

```c
#include <raylib.h>

int main() {
    InitWindow(800, 450, "exemple");
    SetTargetFPS(60);
    while (!WindowShouldClose()) {
        BeginDrawing();
        ClearBackground(RAYWHITE);
        DrawText("coucou", 200, 200, 20, BLACK);
        DrawLine(0, 0, 200, 200, RED);
        EndDrawing();
    }
    CloseWindow();
    return 0;
}
```

1. Recopiez le code ci-dessus dans un fichier `game.c`, puis compilez-le en utilisant la ligne de commande `gcc game.c -lraylib -lGL -lm -lpthread -ldl -lrt -lX11`
   Exécutez le programme pour vérifier qu'il fonctionne.
2. À votre avis, que fait l'instruction suivante : `if (IsKeyPressed(KEY_SPACE)){x += 10;}`
3. Modifiez votre programme pour que le texte `"coucou"` se déplace de 10 pixels vers la droite à chaque fois que vous appuyez sur la touche espace.

Exercice 6 : Le jeu de la vie, version graphique.

La page `https://www.raylib.com/cheatsheet/cheatsheet.html` liste toutes les fonctions de `raylib`.

1. D'après cette page, comment dessine-t-on un rectangle de taille 5×5 avec `raylib` ?
2. Créez un nouveau programme qui implémente le jeu de la vie avec `raylib`. On passera de l'instant t à l'instant $t+1$ en appuyant sur la touche espace. Chaque cellule sera représentée par un carré.

III) Exercices avancés

Exercice 7 : Marche aléatoire.

On considère un point de coordonnées (x, y) sur $\mathbb{Z}^2$. À chaque pas de temps, ce point se déplace d'une case, c'est-à-dire que soit x augmente ou diminue de 1, soit y augmente ou diminue de 1.

1. En utilisant `raylib`, faites un programme qui affiche ce point. Celui-ci se déplace à chaque *frame*.

Le but est maintenant d'ajouter des fonctionnalités à ce programme simple. Vous pouvez les ajouter dans l'ordre de votre choix.

2. Faites en sorte que si le point sort de la fenêtre, il réapparaisse de l'autre côté.
3. Soit n fixé. Faites en sorte que l'on voit les n dernières positions du point et pas simplement sa dernière position. Utilisez pour cela un tableau (circulaire) de taille n.

4. Faites en sorte que l'on voit tout le parcours du point. Utilisez de l'allocation dynamique pour gérer la croissance du tableau.
5. Faites en sorte que l'écran soit constamment centré sur la position actuelle du point.
6. Faites en sorte que le déplacement du point soit biaisé par la position de votre curseur de souris, c'est-à-dire que le point va essayer de se rapprocher de votre curseur, tout en se déplaçant de façon presque aléatoire.

Exercice 8 : Langage BrainFuck.

Le BrainFuck est un langage de programmation exotique très proche d'une *machine de Turing* (cf. cours de FDI). Dans ce langage, on imagine que l'on a une mémoire infinie, organisée sous forme d'un tableau bi-infini, où chaque case est de la taille d'un octet. On a un unique pointeur qui pointe vers une case de la mémoire. Les quelques instructions qui existent permettent de déplacer ce pointeur, de lire la valeur pointée, de la modifier, de gérer les entrées/sorties et de faire des boucles d'exécution.

`>`	Incrémente de 1 le pointeur.
`<`	Décrémente de 1 le pointeur.
`+`	Incrémente de 1 l'octet pointé.
`-`	Décrémente de 1 l'octet pointé.
`.`	Affiche dans la console l'octet pointé sous forme ASCII.
`,`	Lit l'entrée standard et stocke l'octet lu à la position pointée.
`[`	Si l'octet pointé est égale à 0, saute à l'instruction après le `]` correspondant.
`]`	Retourne au `[` correspondant.

FIGURE 1 – Liste des instructions du BrainFuck.

```
++++++++++[>+++++++>++++++++++>+++>+<<<<-]
>++.>+.+++++++..+++.>++.<<+++++++++++++++.>.+++.---.----.>+.>.
```

FIGURE 2 – Exemple de programme BrainFuck. Le retour à la ligne ne fait pas partie du code.

1. Programmez un *traducteur* BrainFuck vers C. Votre programme doit lire un fichier `in.bf` contenant du BrainFuck et doit créer un ficher `out.c` contenant du C. Si on compile `out.c` et qu'on exécute le binaire produit, on doit obtienir la même chose que si on avait exécuté `in.bf`.

Conseil : d'abord, en supposant que l'on ait un tableau de taille bi-infini et une variable `pointeur`, trouvez comment on pourrait traduite chacune des 8 instructions du BrainFuck en du code C. Ensuite, gérez le fait que le tableau soit fini en le redimensionnant dès que `pointeur` sort du tableau.

2. Programmez un *interpréteur* BrainFuck. Votre programme doit lire un fichier `in.bf` et exécuter son code instruction par instruction.
3. On veut visualiser ce qui se passe quand le code est interprété. Modifiez votre programme pour afficher à chaque instant le contenu du tableau bi-infini qui se trouve autour de l'octet pointé ainsi que l'endroit où l'on est dans le code.