NUMP2 - Projet d'Informatique
2025 - 2026

Yannick Bornat - yannick.bornat@enseirb-matmeca.fr
Guillaume Bourmaud - guillaume.bourmaud@enseirb-matmeca.fr
Rémi Giraud - remi.giraud@enseirb-matmeca.fr
Daoud Karakolah - daoud.karakolah@enseirb-matmeca.fr

TP3 : Débogage avec *Valgrind*
Structures chaînées

Les objectifs de cette séance sont :
- Savoir manipuler l'outil de débogage *Valgrind*.
- Se familiariser avec la manipulation des structures chaînées.

Comme prétexte, nous allons nous intéresser à des représentations vectorielles d'images simples et à des manipulations de base. Pour cela nous allons reprendre les types personnalisés du TP 2.

1 Débogage avec *Valgrind*

Valgrind est un outil d'analyse d'accès mémoire de code C. Il peut être utilisé conjointement avec *GDB* pour le débogage et l'analyse de code. Cet outil est basé sur un émulateur de langage machine, qui interprète pas à pas et vérifie chaque instruction du programme binaire (exécutable). Son exécution peut être assez lente car *Valgrind* produit une analyse complète du code, sauf erreur de segmentation, auquel cas le programme s'arrête et livre son analyse jusqu'alors. Cette analyse est affichée dans le terminal, et est au début difficilement interprétable. Cependant, contrairement à *GDB*, *Valgrind* peut détecter les instructions d'accès mémoire illicites, qu'elles entraînent ou non des erreurs de segmentation ainsi que les fuites mémoires (ex. : oubli de `free`).

En pratique, si l'instruction donnant une erreur de segmentation n'est pas indiquée par *GDB*, on peut lancer une analyse avec *Valgrind*. On peut ensuite éventuellement revenir à *GDB* et placer des breakpoints pour comprendre l'état du système avant l'erreur. Régulièrement au cours du développement ou au moment de valider son programme, on peut également lancer une analyse avec *Valgrind* pour détecter les potentielles fuites mémoires.

À noter que tous les systèmes sur lesquels vous pourrez être amenés à développer ne disposent pas de sortie standard type écran et donc de moyens avancés de débogage. Ces outils ne vous dispensent donc pas de faire preuve de rigueur dans votre programmation.

Compilation :	`gcc -g code.c`
Exécution :	`valgrind ./a.out`
Liste des options :	`valgrind --help`

Testez le programme de *code_valgrind.c* en le compilant d'abord simplement avec la commande `gcc code_valgrind.c` puis exécutez-le avec `./a.out`. Le programme ne devrait pas s'exécuter entièrement.

Compilez avec l'option `-g` et exécutez-le avec *GDB*. Qu'observez-vous ?

Exécutez cette fois le programme avec *Valgrind*. Prenez le temps de lire le compte rendu affiché pour localiser l'erreur. Quelle autre information vous donne le programme ?

Corrigez le programme.

2 Tracé de ligne

Écrivez la fonction `draw_line()` qui reçoit un `struct picture`, quatre entiers (les coordonnées de deux points) et une couleur. Cette fonction trace une ligne de la couleur donnée entre les deux points dont les coordonnées sont fournies, et utilise logiquement la fonction `set_pixel()` du TP 2.

Notons x1, y1 et x2, y2 les coordonnées des deux points définissant le segment à tracer. Le nombre de pixels à tracer est : n = max(|x1-x2|, |y1-y2|) + 1. Il ne reste alors qu'à dessiner les n pixels en calculant leurs coordonnées dans une boucle allant de 0 à n-1. Le calcul des coordonnées se fait simplement en partant de x1,y1 et en itérant n points pour aller jusqu'à x2,y2.

Testez la fonction `draw_line()` en recréant l'image de la figure 2 (dimensions 10x10), basée sur :
- un fond blanc,
- une ligne rouge entre (2,2) et (7, 7),
- une ligne bleue entre (2,6) et (7,3),
- deux lignes vertes entre (1,2) et (1,7), et entre (8,2) et (8,7)
- deux lignes magenta entre (2,1) et (7,1), et entre (2,8) et (7,8)

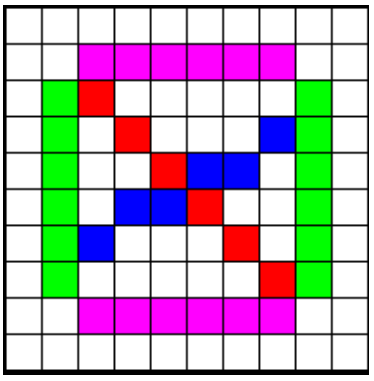

Figure 1: Image test pour le tracé de ligne

La composition des couleurs les plus courantes est donnée dans la table 1.

Table 1: Composition des couleurs principales

Couleur	**Composante Rouge**	**Composante Verte**	**Composante Bleue**
Noir	0	0	0
Rouge	255	0	0
Vert	0	255	0
Jaune	255	255	0
Bleu	0	0	255
Magenta	255	0	255
Cyan	0	255	255
Blanc	255	255	255

3 Images vectorielles

Définition : Une image vectorielle se caractérise par une description géométrique (lignes cercles) indépendante de toute échelle de représentation (par opposition à une image dite *bitmap*, dont la description ne s'intéresse qu'à sa représentation sous forme de pixels). Le format que nous allons utiliser ici n'est pas standard, mais rend bien compte des avantages et inconvénients de ce genre d'image.

Nous allons travailler à partir de deux images données comme ressources sur la page du projet : *cat.txt* et *kang.txt*.

Le format de ces fichiers est le suivant :

- Chaque ligne contient une succession de 4 flottants en représentation ASCii sur 9 caractères chacun.
- Les quatre flottants de chaque ligne représentent respectivement les valeurs x1, y1, x2 et y2 d'un segment à tracer.
- Les 10e, 20e et 30e caractères de chaque ligne sont toujours des espaces (numéros 9, 19 et 29) et servent de séparateurs entre les flottants.
- Chaque ligne a une taille fixe de 40 caractères (en comptant le saut de ligne final).

Les valeurs de chaque point sont exprimées en pixels, mais sont représentées en flottants pour permettre une plus grande souplesse lors de leur manipulation. En effet, comme l'image est codée comme une succession de lignes, il reste possible d'effectuer plusieurs types d'opérations (zoom, rotation, ...) sans en altérer la qualité.

Cette première partie permet de vérifier votre bonne compréhension du format de fichier.

Écrivez un programme qui crée une image de 500x500 pixels et qui trace l'ensemble des lignes de l'image *cat.txt* en utilisant au maximum les fonctions du TP3. Le fichier à lire obéit à un format précis, il est donc possible d'utiliser `fscanf` pour s'éviter des manipulations trop complexes.

4 Mémorisation en structure chaînée

Nous allons maintenant nous intéresser à une série de fonctions permettant d'effectuer diverses transformations. Pour appliquer ces transformations, il est nécessaire de mémoriser les coordonnées de chaque ligne. Nous allons le faire dans une structure chaînée pour garder un maximum de souplesse.

Une structure chaînée est une structure de donnée constituée d'éléments distincts mais dont chacun contient un pointeur vers un élément suivant. Toute fonction ayant accès au premier élément peut donc parcourir l'ensemble des éléments en suivant les références successives. L'organisation constituée par ces éléments chaînés est appelée *liste*. Elle est représentée figure 1.

Déclarez un type `struct vector`, qui contient quatre champs de type double (x1, y1, x2, y2) ainsi qu'un pointeur vers un `struct vector` (qui s'appellera `next`). Chaque élément `struct vector` décrit un segment à tracer. L'organisation constituée par l'ensemble des `struct vector` liés un à un constituera la figure complète.

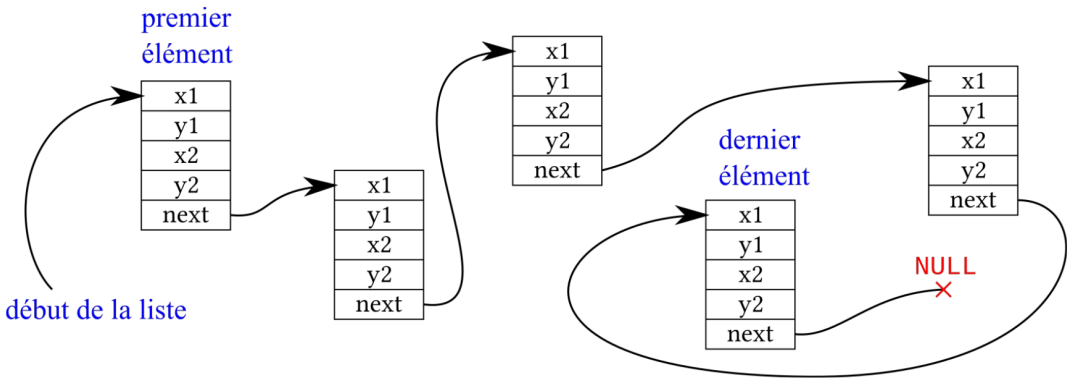

Figure 2: Représentation d'une liste de 5 lignes

Écrivez une fonction `read_vector_file()` qui reçoit un nom de fichier sous forme de chaîne de caractères, et qui renvoie un pointeur vers un `struct vector`. Pour créer la liste des segments, voici l'algorithme le moins pénible (également illustré figure 2):

- créer deux pointeurs : un destiné à référencer le premier élément (initialisé à NULL : pas de premier élément), l'autre référencer temporairement chaque nouvel élément (Fig. 2.a)
- pour chaque élément à ajouter :
  - réserver la place pour créer un nouvel élément (Fig. 2.b)
  - remplir les champs classiques du nouvel élément
  - faire pointer le champ next vers l'ancien premier élément (qui devient le deuxième) (Fig. 2.c)
  - copier l'adresse du nouveau premier élément dans le pointeur destiné à cet effet (Fig.2.d).
- Retourner le pointeur vers le premier élément.

Le prix de la simplicité de cet algorithme est que la liste est créée à l'envers. Ce n'est pas important pour notre application puisqu'il n'y a pas de relation d'ordre entre les différents segments d'une figure.

Écrivez une fonction `draw_vector()` qui reçoit un pointeur vers un `struct vector` (une chaîne de `struct vector` donc), un `struct picture` et une couleur, et qui trace dans l'image et avec la couleur fournie, la figure constituée des segments de la chaîne.

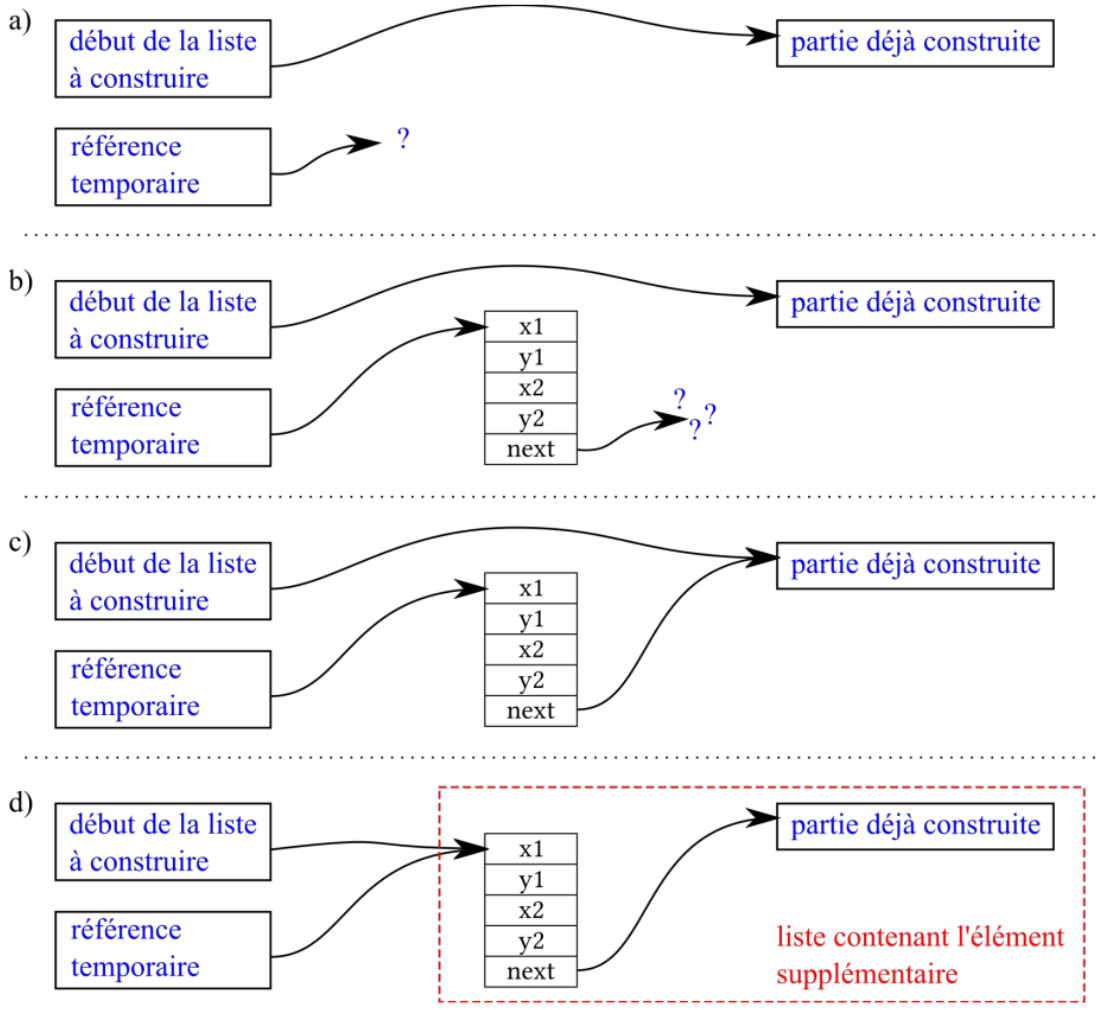

Figure 3: Étapes d'insertion d'un nouvel élément en début de liste. (a) Situation de départ ; (b) création du nouvel élément ; (c) enchaînement de la partie existante à la suite du premier élément ; (d) référencement du nouveau premier élément

Testez vos fonctions avec un programme qui crée une image à partir de *cat.txt*. À ce stade, vous devriez obtenir la même image qu'à la partie 1. L'avancée semble faible, mais cette structure va nous permettre de jouer un peu avec la figure...

5 Transformation de la figure

Écrivez une fonction `scale_vector()` qui reçoit un `struct vector *` et un double (`scale`), et qui multiplie l'échelle de la figure par `scale`. Cette mise à l'échelle se fera sans considération de point de référence. On peut donc se contenter de multiplier chaque coordonnée de chaque segment par `scale`.

Testez la fonction `scale_vector()` en modifiant la taille de la figure *cat.txt*.

Écrivez une fonction `shift_vector()` qui reçoit un `struct vector *` et deux `double` (`x`, `y`), et qui déplace la figure selon le vecteur x,y (dit plus simplement, les coordonnées x,y sont ajoutées à chaque point de coordonnée dans la figure).

Testez la fonction `shift_vector()`.

Écrivez une fonction `flip_vector()` qui reçoit un `struct vector *` et qui applique un effet miroir horizontal. Encore une fois, nous ne prendrons pas de considérations sur la position de l'image, il est donc suffisant de changer le signe des coordonnées x de chaque point.

Testez les fonctions précédentes en affichant deux chats qui se font face l'un l'autre, chacun ayant la moitié de la taille de la figure originale.

Vous pouvez maintenant utiliser la figure *kang.txt* qui reprend le logo du concours kangourou des mathématiques. Ce logo est un pavage. Il devrait donc occuper tout l'écran. Le symbole de base est donné dans *kang.txt* et occupe les dimensions de 100x100.

- Créez la ligne de départ en affichant quatre fois la figure de base aux coordonnées (0, 0), (100, 0), (200, 0) et (300, 0)
- Créez une deuxième ligne identique aux coordonnées suivantes (0, 200), (100, 200), (200, 200) et (300, 200)
- Créez la ligne intermédiaire en calculant le miroir de la figure de base (qui sera alors positionné en (-100, 0), puis en l'affichant aux coordonnées (50, 100), (150, 100) et (250, 100)

Vous pouvez bien sûr étendre le pavage comme bon vous semble...

6 Bonus : Gestion des copies...

À la fin de la partie 3, vous aurez remarqué qu'il serait très pratique de créer une copie de la figure au lieu de toujours modifier l'original.

Écrivez une fonction `duplicate_vector()` qui reçoit un `struct vector *`, et qui crée une copie de la figure en renvoyant un pointeur vers le premier élément de la copie.

Testez votre programme en créant une nouvelle version du programme créant le pavage kangourou, mais utilisant un algorithme plus simple.

Si on peut créer une copie d'une figure, il faut aussi pouvoir effacer une copie pour éviter les fuites mémoires.

Écrivez la fonction `free_vector()` qui reçoit un `struct vector *` , et qui détruit la figure (en libérant la mémoire qu'elle utilise).

7 Bonus : Tracé de lignes récursif (Sierpiński)

Dans cette partie, l'objectif est d'utiliser la récursivité dans un programme qui permet une approche visuelle. Comme prétexte nous allons nous intéresser à la représentation du triangle de Sierpiński (encore une fractale, mais d'un autre type), dont un exemple est donné figure 4.

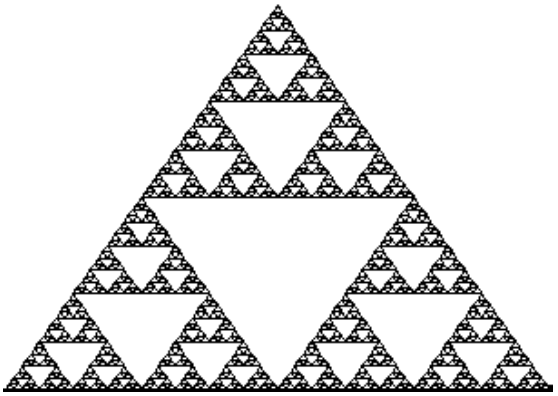

Figure 4: Représentation du triangle de Sierpiński

Le triangle de Sierpiński est, dans sa représentation la plus commune, un triangle équilatéral délimité, à chaque sommet, par un triangle qui lui est identique à l'échelle près (2 fois plus petit).

Écrivez la fonction `sierpinski()` qui reçoit un `struct picture`, trois doubles (les coordonnées d'un point et une taille) et une couleur. Cette fonction trace un triangle équilatéral dont un des sommets est aux coordonnées fournies, et dont le côté mesure taille.

Testez votre fonction en créant une image de dimensions 400 x 350 contenant un triangle de largeur 400 dont un des sommets est en (x=0, y=349). Il s'agit de la représentation de départ.

En toute logique, les coordonnées des autres sommets auront pour coordonnées (taille-1, 349) et (taille/2, 349 – taille*$\sqrt{3}$ / 2).

7.1 Application de la récursivité pour le tracé

Cette partie est destinée à progresser pas à pas vers un algorithme récursif. Chacun peut donc, selon son aisance, passer directement à la création du triangle complet, ou suivre les étapes détaillées.

7.1.1 Utilisation d'une fonction intermédiaire

Écrivez la fonction `sierpinski_div()` qui reçoit les mêmes arguments que `sierpinski()`, mais qui cette fois trace trois triangles, chacun deux fois plus petit que la taille demandée. Chacun de ces trois triangles sera tracé en utilisant la fonction `sierpinski()` aux coordonnées :

- (x0, y0) : triangle en bas à gauche
- (x0+taille/2, y0), : triangle en bas à droite
- (x0+taille/4, y0-taille*$\sqrt{3}$ /4) : triangle du haut

Vérifiez le comportement de la fonction `sierpinski_div()`

7.1.2 Adaptation en fonction de la taille demandée

Modifiez la fonction `sierpinski_div()` pour que le triangle tracé soit composé :

- d'un seul triangle si la taille demandée est inférieure à 300 (comportement identique à `sierpinski()`)
- de trois triangles si la taille demandée est supérieure à 300 (comportement d'origine)

En regardant bien, lors du tracé de la figure en trois triangles, la fonction `sierpinski_div()` peut appeler indifféremment `sierpinski()` ou `sierpinski_div()`. En effet, lorsqu'elle est appelée en interne, la fonction de tracé des "sous-triangles" reçoit une taille inférieure à 300, dans ce cas, les deux fonctions se comportent de façon identique. Ce test sur la taille sera appelé **condition d'arrêt**.

Vérifiez le comportement de la fonction `sierpinski_div()`.

7.1.3 Faible récursivité

En fusionnant `sierpinski()` et `sierpinski_div()`, transformez la fontion `sierpinski()` en une fonction récursive dont la condition d'arrêt est taille<300.

Pour qu'une fonction récursive produise un résultat correct, il faut remplir les conditions suivantes :
- La fonction ne doit pas s'appeler elle-même lorsque la condition d'arrêt est vraie.
- À chaque fois que la fonction s'appelle elle-même, elle doit le faire avec des paramètres qui la rapprochent de la condition d'arrêt.

Gardez le même programme, mais abaissez la condition d'arrêt à taille<150, puis à taille<75.

7.1.4 Figure complète

Déterminez la condition d'arrêt pour tracer le triangle complet. En toute rigueur, la structure est infinie et ne peut donc pas être tracée complètement, mais en informatique, la précision de la représentation des images est limitée à la taille du pixel. On considérera donc que l'image est tracée complètement si les détails de l'ordre du pixel sont visibles, sans considérer les détails plus petits qu'un pixel.

Il n'y a plus qu'à ...

7.1.5 Un peu de couleur...

Pour améliorer le rendu et mettre en évidence le caractère infini des subdivisions, il est intéressant de tracer les triangles avec une couleur d'autant plus sombre qu'ils sont petits.

Modifiez votre programme pour permettre ce résultat.

7.1.6 De la compréhension du phénomène...

Histoire de s'amuser un peu (sic!), modifiez le motif du dessin, tout en gardant les règles d'héritage identiques (par exemple, vous pouvez tracer un carré, un rond, ou juste une croix...)

La figure est perturbée, mais le schéma principal reste le même, ce qui prouve que les règles d'héritage de la récursivité sont plus importantes que les dessins unitaires par eux-mêmes.