Documentation en c : pour le développeur

Sommaire :

Préambule :

Un des principaux objectifs de notre code était de produire le plus clair et petit « main » possible, afin de rendre explicite notre code pour une personne venant à l'utiliser. Pour cela nous avons écrit dans notre « main » les principales fonctions qui seront en charge de la simulation de notre projet avec des noms très explicites. Un des principaux avantages de cette méthode est la détection presque automatique des éventuelles erreurs, en effet chaque fonction contient des potentiels messages d'erreur si la donnée venait à être illogique. Cela pourrait paraitre redondant mais le programme permet de détecter rapidement de quelle fonction provient l'erreur, cette méthode s'est avérée très efficace et nous a permis de gagner beaucoup de temps.

I) Répartition du travail :

Pour réaliser notre projet nous avons d'abord bien discuté entre nous et essayé de bien comprendre le sujet. Nous avons ensuite élaboré un plan afin de se visualiser quelles fonctions seraient nécessaires dans notre code. De manière générale, nous avons procédé comme suit : d'abord nous avons défini les différentes fonctions principales dont nous allions avoir besoin puis pour chaque fonction principale on expose nos idées respectives, et on décide des éventuelles sous-fonctions dont on a besoin, ensuite nous nous sommes répartis les fonctions à coder selon nos convenances (bien-sûr de manière équitable), puis chacun a codé de son côté en effectuant des éventuelles améliorations, enfin nous nous somme regroupé et avons vérifié si tous les éléments fonctionnaient correctement .

II) Explication des différentes fonctions (choix des fonctions et difficultés rencontrées)

D'abord, définissons la structure principale de notre code que nous avons appelé « animale ». Celle-ci prend sept paramètres en compte : sa position x (hauteur), sa position y (largeur), sa direction, son énergie, son numéro de famille qui sont tous de types entiers.

Ainsi qu'un tableau d'entiers qui représente les huit gènes de l'animal et un pointeur sur « l'animal suivant ».

1)Fonctions générales

Beaucoup de ces fonctions sont de types « void », elles ne renvoient donc rien et prennent en argument des pointeurs et effectuent des changements par référence.

La fonction « créer monde » crée une matrice de hauteur et largeur connue dont chaque case contient potentiellement de la nourriture, l'avantage de cette fonction est qu'elle nous permet d'accéder à une case très rapidement grâce aux coordonnées, celle-ci permet aussi de facilement supprimer la nourriture ou d'en rajouter.

Une des principales difficultés rencontrées durant la programmation de notre code fût la manière de gérer la Beauce et la manière de se déplacer dans ce monde. Nous avons utilisé plusieurs schémas pour nous permettre de visualiser au maximum cette zone si particulière car il ne faut pas oublier que le monde est un tore et ceci affecte donc la position de la Beauce. Cette étape nous a permis de bien avancer dans notre projet. Nous avons tout d'abord codé une fonction qui permet de réajuster la Beauce si par exemple cette zone est amenée à s'étendre au-delà de la hauteur et de la largeur du monde pour respecter la propriété d'un tore.

Ensuite, nous avons créé une fonction « ajouter_nourriture_alea_beauce » qui permet de rajouter de la nourriture en sélectionnant d'abord une case aléatoire suivant la hauteur et largeur de la Beauce (fonction définie précédemment). La difficulté rencontrée se situe au niveau du placement de la Beauce dans un monde qui est un tore.

Ainsi, nous avons considéré la Beauce comme un sous monde à part, puis nous avons effectué une translation grâce à ses coordonnées de son coin supérieur gauche (il suffit d'additionner la case aléatoire obtenue avec les coordonnées de la Beauce).

Des difficultés ont également été présentes dans la fonction « deplacer_animale » qui déplace l'animal en fonction du gène activé, nous avons donc dû réfléchir à comment se déplacer dans ce tore. Pour cela, nous avons considéré le cas où l'animal déplacé à l'instant t d'une case à une autre dépasse la dimension du monde, alors il suffit de le placer à hauteur-1 (s'il passe de la hauteur : 0 à -1) ou 0 (cas contraire), on a raisonné de la même manière pour la largeur.

La fonction « gene_aleatoire », qui prend en paramètres le tableau de chromosomes, a fait l'objet de beaucoup de réflexions. Afin de trouver la méthode la plus efficace possible de sélection d'un gène aléatoire proportionnel à la valeur du gène, nous avons procédé comme suit : on additionne toutes les valeurs du tableau de chromosome et on génère un nombre aléatoire entre 0 et la somme obtenue. Afin d'accéder à ce gène, on soustrait la valeur de chaque case du tableau en entrée en commençant par la première case (chromosomes [0]), tant que la différence entre le nombre aléatoirement choisi et la valeur du tableau n'est pas nulle (en incrémentant bien sur l'indice de parcours du tableau).

(NB : Nous expliquons en détail cette fonction car elle pourrait ne pas être comprise lors de la lecture du code).

Afin de vérifier la validité de ce code, nous avons effectué quelques tests. Lors de nos essais, les valeurs des gènes les plus élevées revenaient le plus souvent. De plus on s'est assuré de bien vérifier les cas extrêmes (premier gène et dernier gène).

2)Fonction sur Liste animale

Nous avons décidé de créer une structure « liste_animale » qui possède comme argument le nombre d'animaux de la liste chainée ainsi qu'un pointeur sur la tête de la liste chainée « animale ». Après réflexion, nous avons décidé d'utiliser une liste simplement chainée au lieu d'une doublement chainée,

ce choix augmente néanmoins la complexité qui est en O(n) lors de la suppression d'un animal où « n » est la taille de notre liste. Ce choix reste cependant pratique puisqu'il simplifie l'écriture ainsi que la manipulation des autres fonctions.

(On considère « n » la taille de notre liste chainée).

La fonction « dupliquer_ou_tuer_animale_liste » a fait l'objet de réflexions, nous nous sommes finalement décidés de faire une fonction « deux en un » qui permet de tuer ou de dupliquer un animal. Ces deux cas sont différenciés selon l'énergie de l'animal : si son énergie dépasse le seuil de reproduction on le duplique, s'il est égal à 0 on le supprime. Cette fonction a un rôle important car elle renvoie la valeur 1 si l'animal est supprimé, 0 dans le cas contraire. Ceci permet de savoir lors du parcours de notre liste chainée dans le « main » si un animal a été supprimé ; si oui, on libère la place mémoire de cet animal. Ainsi, l'allocation dynamique est bien gérée lors de la simulation.

La fonction « nourrir_et_baisser_energie _animale » nourrit l'animal d'une unité de nourriture s'il en existe dans sa position et enlève cette unité du monde, et dans tous les cas elle baisse son énergie de moins 1 sauf si l'énergie de l'animal est nulle. Cette fonction s'exécute en O (1).

3)Fonctions sur les fichiers

La partie sur les fichiers représente une étape assez importante dans notre projet.

Tout d'abord, nous avons décidé d'écrire deux fonctions qui manipulent notre fichier d'entrée pour éviter d'avoir une fonction trop longue, cela permet de rendre le code plus claire et compréhensible. La première qu'on appelle « verifier_syntaxe_fichier » s'occupe de vérifier si notre fichier d'entrée est syntaxiquement correct et la deuxième « recupe_donnes_fichier » traite d'éventuelles erreurs de logique et arithmétique (ex : nourriture négative, coordonnées Beauce qui dépasse le monde, etc.) et rempli en même temps notre liste chainée d'animaux.

4)Fonctions sur familles

Afin de gérer les familles d'animaux, nous avons choisi de faire un tableau de pointeur de type « liste animal » dont chaque case contient un pointeur de type « liste animal » et chaque indice du tableau indique le numéro de la famille, ce qui permet de placer les animaux dans leurs familles très facilement grâce à l'indexation du tableau. Ainsi pour remplir notre tableau on parcourt une fois notre liste chainée et chaque animal est placé dans sa famille en un temps de O (1), finalement on obtient un temps en O(n). Ce tableau permet également de remplir notre fichier de sortie. Notons que l'allocation du tableau est gérée de manière dynamique ce qui permettra par la suite de libérer cet espace mémoire de manière simple.

Pour trier les animaux, nous avions tout d'abord pensé à un tri en manipulant les pointeurs de la liste chainée mais cela s'est avéré très difficile à faire. Nous avons finalement dépassé cette difficulté en optant pour un tri à bulle, car celui-ci échange deux valeurs lors du tri, cela permet de trier par changement de valeur et simplifie grandement notre code. Pour ce faire, nous avons créé une sous-fonction « echange_animale » qui prend en argument deux pointeurs de types structure « animal » et permute les caractéristiques (les champs de la structure « animal ») de deux animaux. Ce tri à bulle s'effectue en $O(n^2)$.

5)Fonctions sur Images

D'abord définissons la structure « pixel » ainsi que la structure « Image » qui sont les bases de nos images. La première structure « pixel » qui contient 3 « int » : R, V, B pour les couleurs rouge, vert et bleu.

Ensuite, nous avons dû penser à une structure image qui nous servira à représenter notre « monde » virtuel de pixels, après réflexions nous avons pensé à la structure suivante :

Celle-ci prend en paramètres deux entiers qui serviront de dimensions pour notre monde (la hauteur et la largeur).

Ensuite un tableau à deux dimensions de type pixel, où chaque pixel représente une case de notre monde. Nous avons choisi de créer un tableau de pixel car il permet de placer facilement nos animaux grâce à leurs coordonnées. Cette structure est ainsi cohérente avec le reste de notre programme.

Pour créer une image à l'instant t. Nous avons eu besoin de quatre fonctions principales :

1) Init_image_monde :

Cette fonction initialise l'allocation nécessaire pour notre structure « image » et rempli notre monde de pixel(l'image) par une couleur grise dans toutes les cases, cette image représente le monde vide.

2) Remplir_beauce_image :

Cette fonction affiche dans l'image la Beauce en vert, le centre de la Beauce est représenté par un vert plus clair.

3) Remplir_nourriture_image :

Pour représenter la nourriture dans notre monde nous avons décidé de la représenter en noir. On sélectionne les cases non nulles du monde (et donc contiennent de la nourriture) et on colore le pixel de la case en noir.

4) Remplir_liste_animaux_image :

Pour créer cette fonction il a fallu au préalable affecter à chaque famille une couleur. Pour cela nous avons créé un tableau de pixel dont la taille est le nombre de famille. Chaque indice du tableau représente le numéro de famille et chaque case contient une couleur aléatoire qui représente la famille.

Pour coder notre fonction principale, nous avons rencontré quelques difficultés. En effet, le sujet indique que l'on doit représenter les animaux par leur couleur de famille, et dans le cas où deux animaux seraient sur la même case, c'est celui qui a le plus d'énergie qui serait représenté. Cette condition nous a posé quelques difficultés car on ne doit pas parcourir la liste des animaux deux fois sinon la complexité de l'algorithme augmente. Nous avons donc pensé à l'idée suivante :

On parcourt notre liste d'animaux, on rajoute dans le monde la couleur chaque animal selon la couleur de sa famille, et on stocke dans un tableau à deux dimensions l'énergie de l'animal. De cette manière, il suffit de comparer l'énergie du prochain animal qui serait sur la même case à celle stockée dans le tableau, cela nous évite de parcourir deux fois notre liste chainée et on gagne par conséquent en complexité.

Notons que pour obtenir l'image finale de notre monde à l'instant t, l'ordre d'enchainement des fonctions est important car il permet d'écraser les couleurs précédentes au fur et à mesure de l'appel de chacune de ces fonctions.

Nous avons choisi d'écrire ces fonctions séparément pour ne pas avoir une trop grande fonction, ce qui permet de rendre le code plus clair, lisible et compréhensible.

III) Tableau de complexité des fonctions (principales)

Ce tableau représente la complexité de nos fonctions principales.

NB : on inclut bien évidement dans le calcul la complexité des sous fonctions utilisées dans les fonctions principales.

(On considère « n » la taille de notre liste chainée et on considère « hauteur », « largeur » les dimensions de notre monde).

Fonctions principales	Complexité
« deplacer_animale »	**O (1)**
« nourrir_et_baisser_energie_animale »	**O (1)**
« dupliquer_ou_tuer_animale_liste»	**O (n)**
« trier_liste_animale »	**O (n^2)**
« creer_fichier_image_monde »	**O ((hauteur*largeur) +n)**