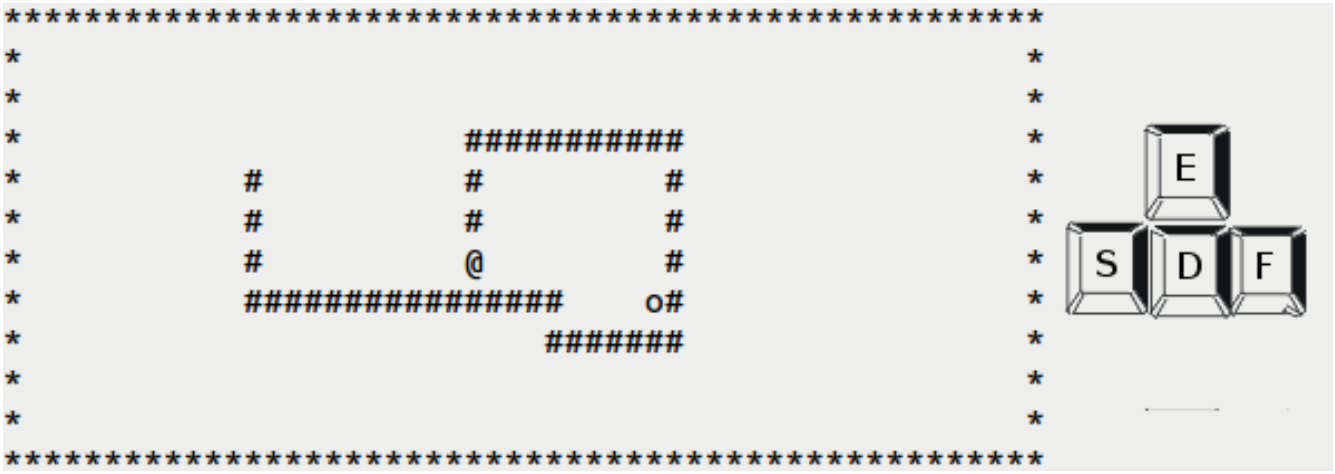
Maxime CHICOIS
14/11/14

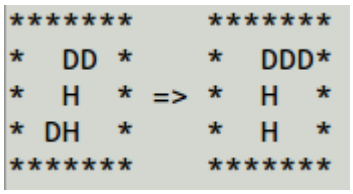
Projet Personnel en C
Snake



I) Première idée : construire le snake depuis une matrice de mouvement

Cette première idée consistait à enregistrer dans une seconde matrice les mouvements du snake afin de le reconstruire à chaque étape graphiquement ensuite (en remplissant la matrice par des caractères H,B,D et G en fonction de la direction). La lettre d'une case correspond à la position du prochain élément du Snake.
La matrice du Snake aurait alors comme information cette matrice de mouvement, ainsi que la position de la queue (l'hérédité de la queue étant conservée, sachant que la matrice de mouvement permet de déduire (grâce à la lettre de la queue) quelle case adjacente devient la nouvelle queue).

On représente dans l'exemple ci-dessous l'affichage de la matrice de mouvements avant et après que l'on appuie sur une touche de mouvement (ici vers la droite).



À chaque mouvement, on recalcule la matrice de mouvements, puis on trace le Snake avec une fonction graphique (# pour le corps, @ pour la tête).

Cependant, cette première idée semblait trop compliquée par rapport à ce qui pouvait se faire, j'ai donc décidé de revoir le programme intégralement.
Dans un premier temps j'ai créé une structure **coordonnee** qui consiste à enregistrer l'abscisse et l'ordonnée d'un élément de la matrice, il s'agit donc d'un record de deux entiers.

```
typedef struct {
     int x;
     int y;
} coordonnee;
```

On peut ainsi stocker la position du Snake dans un tableau de coordonnées **position**. Cependant il faut prendre en compte que la taille du Snake varie, il faut donc allouer assez d'espace pour éviter tout problème (à savoir en fonction de taille de la matrice, Hauteur*Largeur).

Une autre solution vue plus tard en cours consiste à utiliser des listes chaînées afin de stocker la position du Snake. cependant ayant déjà commencé à travailler sur un tableau fixe, j'ai préféré continuer là-dessus.À partir de ce tableau, la première étape consistait à faire bouger un snake d'une taille fixe en gérant le déplacement de chaque élément du Snake.
Ceci est géré par la fonction CoordonneeSnake qui affecte à chaque morceau de snake sa nouvelle position :

```
int i;
for (i=0;i<taille-1;i++)
{
     position[i]=position[i+1];
}
```

La position de la tête dépend du mouvement : on calcule sa valeur à l'aide du case of fourni dans le fichier sample.c. Un Snake de taille fixe peut alors bouger sur l'écran à l'aide du clavier. Cependant il ne gère pas encore les collisions et peut se rentrer dans lui-même (par exemple si on appuie sur la touche de direction opposée).

Par exemple, en appuyant deux fois sur la touche droite dans cette situation :

```
###@ => ##@ => #@#
```

La gestion de cette collision sera gérée une fois que le Snake bougera de manière automatique (tant que l'on n'appuie pas sur une touche pour le faire changer de direction) en gêrant les exceptions.

II) L'arrivée des fruits

À partir de la fonction **random** vue en cours (modulo la taille de la matrice) on affiche un fruit (noté "o") sur une case vide. Plus précisément, tant que la case n'était pas vide on recalcule ses coordonnées. Techniquement cela peut engendrer une boucle infinie si jamais la matrice est pleine, cependant cela n'arrive que si le Snake remplit déjà tout l'écran.

Lorsque le Snake mange un fruit, celui-ci s'agrandit (c'est-à-dire qu'il conserve sa queue et le reste de son corps lors du prochain mouvement). Dans un même temps, on calcule la nouvelle position du fruit. L'affichage du Snake fonctionne comme lorsqu'il ne mange pas de fruit, il s'agit juste de modifier différemment le **tableau** position contenant les coordonnées (on augmente la variable **taille** de 1, ce qui a pour effet de décaler la tête d'un cran dans le tableau).

III) Gérer les collisions, le déplacement automatique du Snake et les exceptions

Jusque là, le Snake pouvait se traverser. On crée alors une fonction **Collision** qui permet de sortir de la boucle while (la période de « jeu ») lorsque l'on se mord la queue.
Pratiquement, elle consiste icisimplement à modifier la valeur du paramètre du while **go** lorsque l'on tente d'écrire la tête (signe @) sur un signe # déjà existant.

On modifie ensuite le readkey initialement donné en rajoutant ces lignes :

```
     newattr.c_cc[VMIN] = 0 ;
     newattr.c_cc[VTIME] = (taille < speed) ? 2 : 1;
```

Ainsi, tant que l'on n'appuie pas, le readkey renvoie -1 (en continu) et continue d'aller dans la dernière direction donnée (il faut au préalable avoir modifié le main), plutôt que d'attendre que l'on appuie sur une touche afin qu'il bouge d'une case.
Initialement, la seconde ligne (correspondant au temps d'attente, 1 étant la vitesse maximale du Snake) vallait 1. On rajoute ici une condition (si le Snake est suffisamment grand, on augmente la difficulté en accélérant le jeu).

Une fois ceci mis en place, on gère un certain nombre d'exception : lorsque l'on est dans une direction donnée on ne doit pas pouvoir se retourner, sinon on rentre dans le Snake immédiatement et cela crée une collision.
De plus, si on appuie sur une autre touche que celles choisies (q pour quitter, ainsi que les 4 de mouvements) on ne doit rien faire.

```
switch(d)
{
case 's':
case 'f':
     if(c == 'd' || c == 'e')
          c=d;
     break;
case 'd':
case 'e':
     if(c == 's' || c == 'f')
          c=d;
     break;
}
```

Si on se déplace initialement horizontalement (respectivement verticalement) et que la prochaine touche pressée est verticale (respectivement horizontale) alors on enregistre le mouvement.

IV) Conclusion et remarques

Un certain nombre de points au niveau de la complexité peuvent être amélioré, en particulier lorsque l'on gère le cas où un fruit est mangé (on déplace inutilement toutes les cases pour les remettre dans leur état d'origine lorsque l'on remarque que le fruit a été mangé).
De plus l'utilisation de listes permettrait de gagner une place considérable sachant que l'on travaille ici avec un tableau fixe de taille correspondant au pire des cas. De plus il peut paraître plus naturel de travailler ici avec des listes (sachant qu'il suffit de rajouter une tête et de retirer la dernière queue).

Ce projet aura duré environ 10 heures en comptant la rédaction du compte rendu, la réflexion initiale afin de trouver une méthode pratique pour résoudre le problème ainsi que la compréhension de Doxygen.

Des bugs persistent, en particulier le fait qu'en restant appuyé sur une touche directionnelle, le Snake se met à bouger assez vite (cela est lié au Readkey). Cela se voit surtout lorsque le Snake n'a pas encore accéléré (avant qu'il n'atteigne une taille de 15).

De plus, une des premières choses qui a été faite consistait à jouer avec les flèches directionnelles (haut, bas, gauche droite) sur le clavier en obtenant dans un premier temps leur valeur. Cependant ces caractères spéciaux ne correspondent pas exactement à un caractère (↑ correspondant à [A par exemple).
Lorsque le snake ne bougeait pas tout seul (une pression de touche = un mouvement) il avait tendance à faire plusieurs mouvements en une seule pression, voire même déformer le Snake (les coordonnées du snake se superposent, ce qui fait qu'une partie de son corps se trouve plusieurs fois sur une même case, rapetissant ainsi l'ensemble). Ces touches ont donc été ignorées par la suite.

Enfin, le programme a été initialement codé sur un seul fichier, en avançant petit à petit en rajoutant des morceaux du Snake. Des constantes (H et L représentant les dimensions du Snake) se retrouvaient un peu partout dans les fichiers, après avoir segmenté ce dernier : il a fallu créer un fichier constantes.h contenant ces valeurs. Cependant si jamais on désire éditer ce dernier sans toucher aux différents fichiers .c, il faut faire la commande **rm *.o** avant de lancer le makefile afin de forcer l'édition des différents fichiers (constantes.h n'ayant pas une extension .c) afin que la modification soit effective.