Ce document est à l'usage des développeurs

Table des matières

1.Grille

Le fichier grille.h contient tous les éléments relatifs à la construction d'une grille. Ils seront appelés dans jeu.c.

Ok ok ! Vous avez certainement dû apercevoir une variable comme celle-ci ! AFFICHE_GRILLE_H_

a) Une drôle de variable

```
#ifndef AFFICHE_GRILLE_H_
#define AFFICHE_GRILLE_H_
```

C'est alors que Sherlock se pose la question mais à quoi cette variable peut bien servir, C'est élémentaire Watson elle permet d'être sûr que chacune des fonctions est appelée qu'une seule fois pour éviter tout problème car en effet la variable est définie qu'une seule fois donc il n'y a pas de raison pour que celle-ci soit appelé quelque part dans le code.

b) Les couleurs

On y définit les constantes de couleurs comme la couleur noire (qui est en réalité dorée vu la tête du code hexadécimal) et blanc par exemple les couleurs sont définies en hexadécimal.

```
#define NOIR 0xffd700
#define BLANC 0xFFFFFF
```

c) Une structure pour de l'allocation de mémoire dynamique

Tout est dans le titre, on désire que les tailles de lignes et de colonnes soient traitées de manière automatique par le code, pour cela on passe par de l'allocation dynamique, cette allocation se concrétise par une simple structure prenant une grille et deux constantes, nombre de lignes et nombre de colonnes.

```
typedef struct
{
        char** grille;
        int nombredeligne;
        int nombredecolonne;
}Grille;
```

Pour l'originalité je l'ai appelé Grille car cette structure contiendra un tableau double de caractères que j'ai nommé grille qui sera plus tard affichée.

d) La fonction alloue_grille

La fonction alloue_grille prend en paramètre d'une part le nombre de ligne et d'autre part le nombre de colonne, elle se contente de retourner un pointeur de grille qui a alloué les ressources pour la grille demandée.
Cette fonction repose sur les mallocs, ou allocation dynamique de tableaux, la fonction est plutôt simple.
Il s'agit dans un premier temps de déclarer une variable de type Grille* et oui un pointeur de structure.
On alloue à l'adresse de ce pointeur une structure grille, cette structure grille est alors dans le tas, il ne reste plus qu'à l'alimenter !

```
variable = (Grille*)malloc(sizeof(Grille));
```

Pour l'alimentation, il suffit tout simplement d'allouer en mémoire les lignes et les colonnes de la grille pour cela rien de plus simple, on va d'abord allouer toutes les lignes à l'adresse de la structure pointant vers la grille (le tableau double de char).

```
variable->grille=(char**)malloc(sizeof(char*)*n);
```

Maintenant que les lignes sont en place il suffit d'y allouer pour chaque ligne le nombre de colonne, j'ai opté pour une boucle for

```
for (i=0;i<n;i++){
        variable-> grille[i]=(char*)malloc(m*sizeof(char));
```

Désormais notre tableau double de char est alloué il suffit de compléter le reste de notre structure c'est à dire son nombre de ligne et son nombre de colonne, il suffit alors de faire pointer à l'adresse de « nombredeligne » et « nombredecolonne » et d'y associer les valeurs de la fonction prise en paramètre, comme suit :

```
variable->nombredeligne = n;
variable->nombredecolonne = m;
```

e) La fonction libere_grille

C’est intéressant de pouvoir créer des grilles dans le tas, je suis certain que vous serez d’accord avec moi, mais le gros défaut des mallocs c’est que la mémoire ne se libère pas automatiquement, et oui ! Il faut la libérer soi-même, vous serez alors le héros de votre mémoire vive qui vous remerciera du mieux qu’elle le peut.

Pour accomplir cet exploit, on va procéder ligne par ligne en les libérant une à une.

```
for(i=0;i!=(variable->nombredeligne);i++){
        free(variable -> grille[i]);
```

Maintenant que les lignes et les colonnes de grille (le tableau double de caractères est libéré) il suffit simplement de libérer le « nombredeligne » et « nombredecolonne » ce qui libérera la structure du tas. Pour faire d’une pierre deux coups on peut faire ainsi :

```
free(variable);
```

f) La fonction debug

Les gens disent généralement qu’ils croient ce qu’ils voient, cette fonction sert exactement à ça, à montrer afficher dans le terminal la grille (le tableau double de caractères).
Cette fonction prend en paramètre un pointeur de structure Grille et affiche la grille pointée à l’adresse de ce pointeur.
Pour faire cela pour chaque ligne on affiche les colonnes en affichant un espace à chaque fin de ligne.

```
for (i = 0; i != variable->nombredeligne; i++)
{

        printf ("\n");
        for (j = 0; j != variable->nombredecolonne; j++)
        {
                printf ("%c",variable->grille[i][j]);
        }
}
```

g) La fonction sauver

La fonction sauver prend en paramètre un pointeur de structure Grille, et un nom de fichier.
La fonction sauver permet en autre de sauver la grille pointée à l’adresse de la structure Grille passée en paramètre dans un fichier lui aussi un fichier passé en paramètre et de la stocker.
On procède tout d’abord par donner les droits au fichier comme ceci :

```
FILE * fichier;
fichier = fopen(file,"w+");
```

Puis pour chaque ligne lue par le pointeur à l’adresse de la structure vers grille, puis la sauver dans le fichier de destination passé en paramètre.

```
for(;i<g->nombredeligne;i++){
        fputs(g->grille[i],fichier);
        fputs("\n",fichier);
}
```

On utilise la fonction fputs qui permet de stocker une ligne entière dans un fichier, à chaque fin de ligne on y stock aussi un saut de ligne pour éviter de tout stocker sur la même ligne.

Puis on pense à fermer le fichier, pour éviter tout problème.

```
fclose(fichier);
```

h) La fonction redefine_fenetre

La fonction redefine_fenetre donne à l'utilisateur de changer la taille de la fenêtre proposée par le Caml,

Il demande à l'utilisateur d'entrer les valeurs des fenêtres puis les stocks dans un tableau qui est passé en paramètre, j'ai utilisé la commande scanf, bien utile pour demander à l'utilisateur d'entrer des valeurs d'int.

```
printf ("Veuillez entrer la largeur : ");
scanf ("%d", &LARGEUR);
printf ("Veuillez entrer la hauteur : ");
scanf ("%d", &HAUTEUR);
*fenetre = LARGEUR;
*(fenetre + 1) = HAUTEUR;
```

i) La fonction lecteur

La fonction lecteur prend en paramètre un pointeur de structure Grille et un nom de fichier et retourne un nouveau pointeur de structure Grille.
Elle permet de lire une grille que j'aurais créé en fichier par exemple la default_grille, ou bien celle avec mon prénom dans le fichier test.

Cette fonction appelle deux autres fonctions, qui permettent de récupérer le nombre de ligne et de colonne du fichier.
Pour compter le nombre de ligne on appelle la fonction count_nb_line,

```
while(!feof(stream))
{
        ch = fgetc(stream);
        if(ch == '\n')
        {
                nbline++;
        }

}

rewind(stream);
```

Elle incrémente un compteur à chaque fois qu'on est en fin de ligne dans le fichier, on le vérifie à l'aide du caractère de fin de chaîne de caractères soit '\n'. La fonction rewind permet de remettre le pointeur de fichier qui s'est déplacé à la fin au début du fichier, ça serait bête de ne pas pouvoir compter car on est à la fin du fichier à chaque qu'on désire rouvrir le fichier.

Puis ensuite, nous allons cette nouvelle structure dans le tas avec ces éléments, puis enfin nous copions élément par élément du fichier vers la grille (tableau double de caractères).
Pour copier élément par élément on vérifie simple qu'on est pas à un saut de ligne sinon on va copier le caractère '\n' ce qu'on ne veut pas.

```
void copy(char* line_tmp,char *grille){
        int i=0;
        while(*(line_tmp+i)!='\n'){
                grille[i]=line_tmp[i];
                i++;
        }
}
```

j) La fonction affiche_grille

La fonction affiche_grille prend en paramètre un pointeur de structure Grille, la hauteur de la fenêtre et la taille d'un côté d'un carré.
Ce qu'on va faire c'est qu'on va calculer à la fois la position sur la fenêtre et la position du caractère qu'on veut afficher dans la grille (le tableau double de caractères) pointée par le pointeur de structure Grille.
La variable j va calculer la position verticale sur la fenêtre graphique, et la variable i va calculer la position horizontale sur la fenêtre graphique. Alors que la variable l va regarder la position sur la ligne de la grille pointée par le pointeur de structure Grille, et la variable k va regarder la position de la colonne de la grille pointée par le pointeur de structure Grille.

```
for (j = h - c, l = 0; l != (variable->nombredeligne); j -= c, l++)
{
        for (i = 0, k = 0; k != (variable->nombredecolonne); i += c, k++)
        {
```

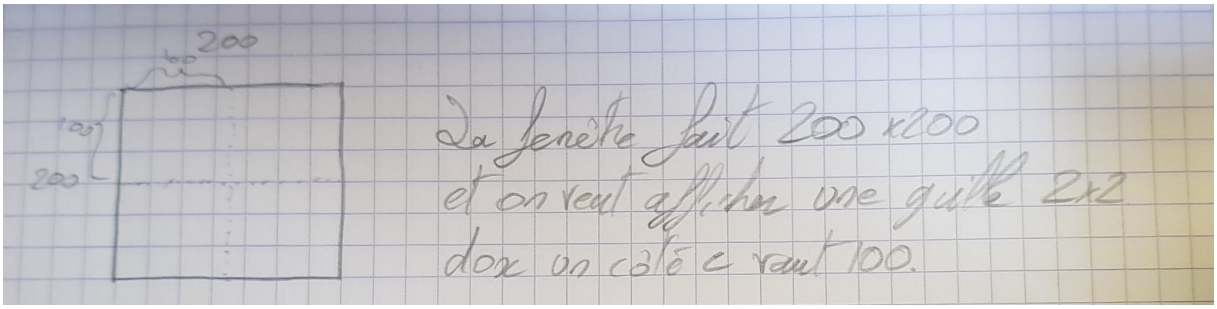
On vérifie alors si dans la grille pointée par le pointeur de structure Grille trouve un élément « n » puis on l'affiche de la couleur passée en paramètre dans le .h à la position pointée par i, j et puisque c'est un carré que l'on veut afficher la longueur et la largeur sont de c.
On décrémente à chaque tour de boucle j et on incrément i de c car, on a segmenté la fenêtre graphique en c carrés horizontaux et c carrés verticaux, on commence en haut à gauche de la fenêtre.

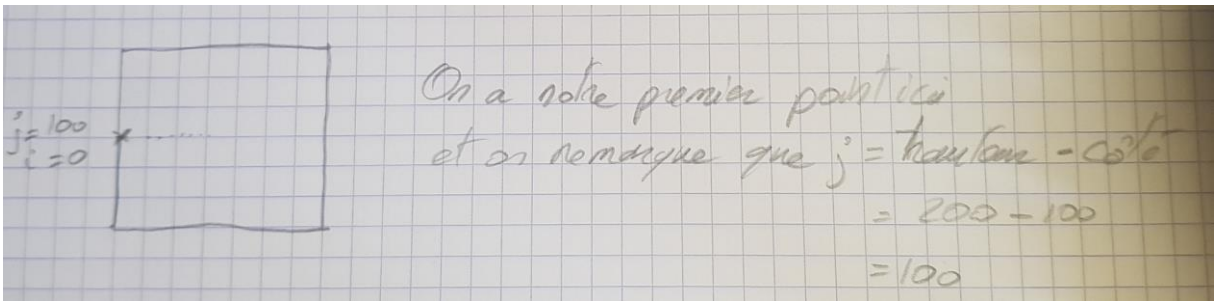
```
a = variable->grille[l][k];
if (a == 'n')
{
        gr_set_color (NOIR);
        gr_fill_rect (i, j, c, c);
}
```

Sinon affiche un carré blanc au même endroit.

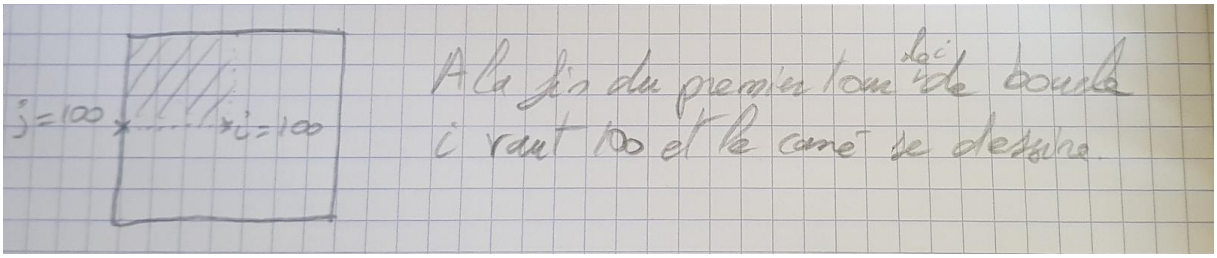
```
else
{

        gr_set_color (BLANC);
        gr_fill_rect (i, j, c, c);

}
```

Exemple sur papier :

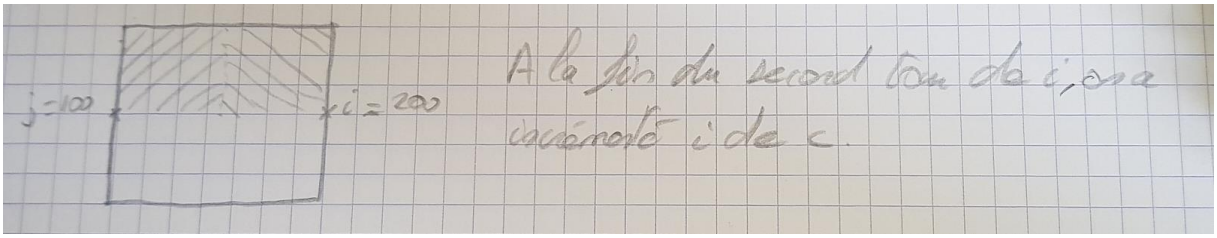




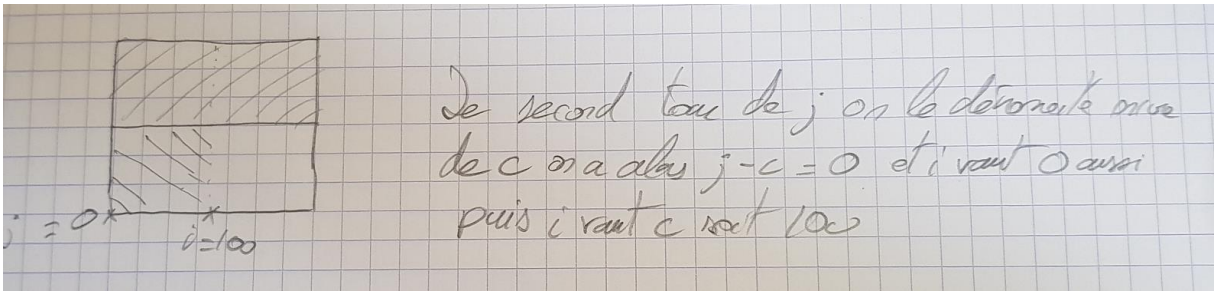
On a noté premier point ici et on remarque que j = hauteur - côté
= 200 - 100
= 100



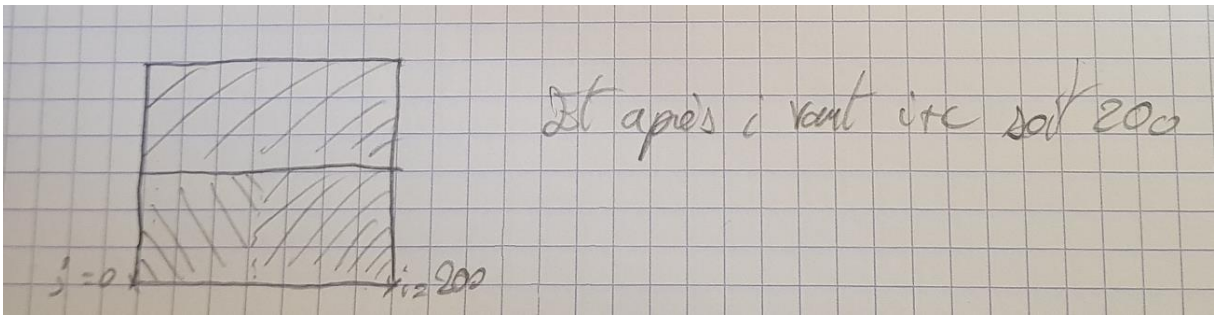
A la fin du premier tour de boucle i vaut 100 et le carré se dessine.



A la fin du second tour de i, on a incrémenté i de c.



Le second tour de j on le détermine encore de c on a alors j-c=0 et i vaut 0 aussi puis i vaut c vaut 100



Et après i vaut i+c soit 200

k) La fonction copie_grille

La fonction copie_grille permet de copier une structure Grille vers une autre en s'aidant des pointeurs de structure Grille.
Pour chaque ligne, on va copier les éléments de chaque colonne élément par élément.

```
for (l = 0; l != (variable->nombredeligne); l++)
{
        for (c = 0; c != (variable->nombredecolonne); c++)
        {
```

Pour cela on va faire pointer nos pointeurs de structures à l'adresse de la grille (tableau double de caractères).

```
variable1->grille[l][c] = variable->grille[l][c];
```

l) La fonction compte_voisin

La fonction compte_voisin prend en paramètre un pointeur de structure Grille, et une position i (les lignes), et j (les colonnes).

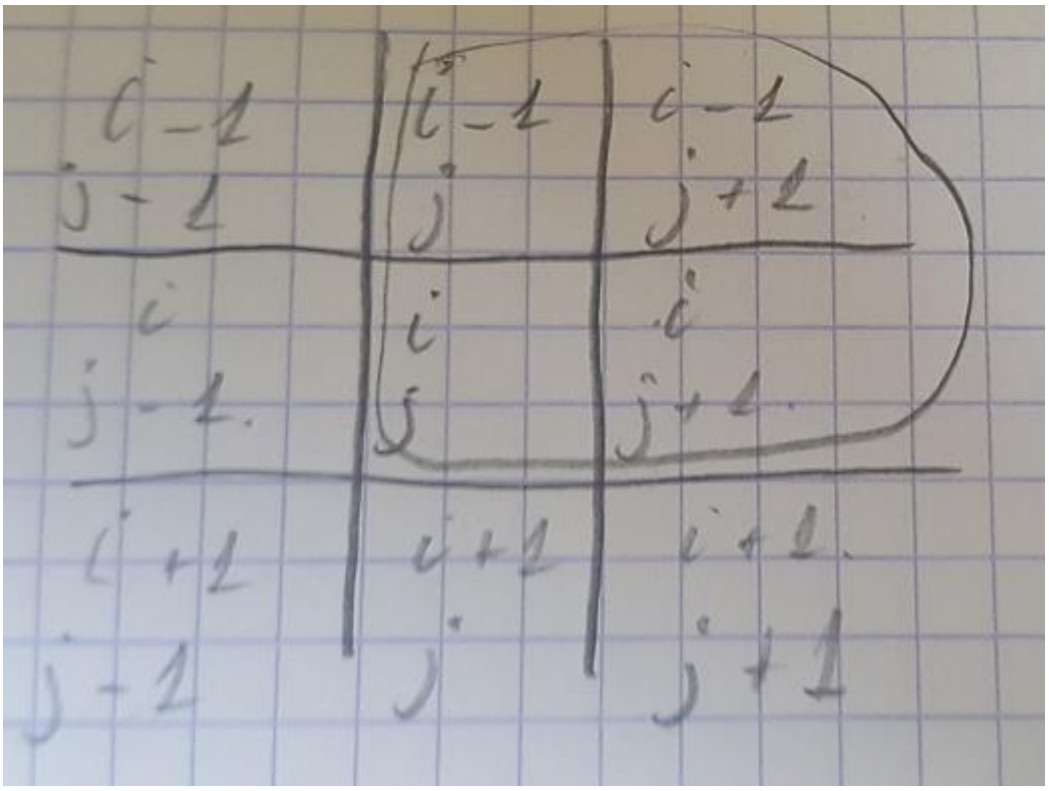
Il ne faut jamais faire confiance à ses voisins, on ne sait jamais qui ils sont ! il s'agit de l'essence même de l'émission petit secret entre voisins. On va donc vérifier pour chaque cas critiques genre les coins, les bords, etc., la présence des 'n'

Je vais prendre seulement un exemple pour expliquer parce que c'est très redondant.

```
if( i == 0 && j ==0){
        if (variable->grille[i - 0][j + 1] == 'n')
                compteurnoir += 1;
        if (variable->grille[i + 1][j + 1] == 'n')
                compteurnoir += 1;
        if (variable->grille[i + 1][j - 0] == 'n')
                compteurnoir += 1;
        return compteurnoir;
}
```

On s'intéresse ici au coin en haut à gauche,

On peut alors s'aider de ce schéma



On vérifie que pour les trois côtés que l'on peut regarder (pas plus puisque la grille n'est pas allouée ailleurs donc une erreur de segmentation potentielle) la présence de n et s'il est bien présent on incrémente notre petit compteur.

m) La fonction applique

La fonction applique prend en paramètre un pointeur de structure Grille.
Il s'agit là de la fonction clé qui permet au jeu de la vie d'exister, elle suit des règles précises, donnés en TP :

- une case naît (devient noire) si elle a exactement 3 voisines vivantes (noires)
- une case ne change pas (reste blanche si elle est blanche, noire si elle est noire) lorsqu'elle a exactement 2 voisines vivantes (noires)
- sinon meurt (devient blanche)

Puisque l'état d'une case à l'étape n+1 dépend de l'état de son voisinage complet à l'étape n on a besoin d'une grille temporaire pour sauvegarder la grille de l'étape n avant de commencer à appliquer les changements.

```
copie = alloue_grille(variable->nombredeligne,variable->nombredecolonne);
copie_grille(variable, copie);
```

Puis on vérifie pour les conditions sur chacun des éléments de la grille (tableau double de caractères).
Pour faire cela on compte tout d'abord le nombre de voisin d'élément qu'on est entrain de regarder.

```
        for (l = 0; l != variable->nombredeligne; l++)
        {
                for (c = 0; c   != variable->nombredecolonne;
c++)
                {
                        casenoir = compte_voisin (copie, l,
c);
```

Puis on vérifie les conditions exposées ci-dessus

```
                        switch (casenoir)
                        {
                                case 3:
                                        variable->grille[l]
[c] = 'n';
                                        break;
                                case 2:
                                        break;
                                default:
                                        variable->grille[l]
[c]= ' ';
                                        break;
                        }
```

n) La fonction applique_n_fois

La fonction applique_n_fois prend en plus d'un pointeur de structure Grille un nombre n en paramètre.

En-fait l'idée est simple, on va faire un for n fois de la fonction applique.

```
for(i=0;i<n;i++){
        applique (variable);
}
```

o) La fonction calcule_cote

Cette fonction permet de calculer la valeur d'un côté en fonction du nombre d'élément présent dans la grille (tableau double de caractères) et de la largeur et de la longueur de la fenêtre.

L'idée est la suivante : Quelle serait la taille d'un côté si j'ai par exemple 2x2 éléments à afficher sur une fenêtre de 200x200 alors, on peut affirmer que chaque côté vaudra 100 de taille.

```
while (w > (variable->nombredecolonne)*c && h > (variable->nombredeligne) * c)
        c += 1;
```

On peut faire en sorte d'incrémenter un compteur c qui représente la taille du côté d'un carré tant qu'on n'a pas dépassé la largeur et la longueur.

2.Pile

La pile permet entre autres de revenir en arrière le fichier pile.h décrit la signature des fonctions et le pile.c décrit toutes les fonctions pour la créer qui seront appelés dans le jeu.c

a) La même drôle variable

Explication dans le 1) Une drôle de variable

```
#ifndef PILE_H
#define PILE_H
```

b) La structure Cell

Il s'agit de la structure même qui va permettre de créer une cellule dans la liste chaînée, en gros créer un élément dans la liste chaînée, il contient une grille, et un pointeur de cellule qui va pointer vers la cellule suivante.

```
typedef struct _cell
{
    Grille * g;
    struct _cell *nxt;
}Cell;
```

c) La structure Tampon

La structure Tampon me permet de renvoyer deux valeurs, notamment quand je dépile, car j'avais un problème lorsque je retirais des éléments de la pile, en effet la grille n'était pas mise à jour, alors pour remédier à ce problème je suis passé par une structure tampon qui me permettra de mettre à jour la valeur de la grille.
Cette structure contient un pointeur de structure Cell que j'ai nommé liste (position de la cellule dans la liste), et un pointeur de structure Grille que j'ai nommé grille.

```
typedef struct _result
{
        Cell* liste;
        Grille* grille;
}Tampon;
```

d) La fonction add_first

La fonction add_first prend en paramètre un pointeur de structure Cell, et un pointeur de structure Grille.

Il faut noter qu'on empile en bas de la pile et non vers le haut ! (Plus facile pour lisibilité quand on veut faire débogage).
On va d'abord allouer une nouvelle cellule tout comme les grilles

```
Cell* nouvelCell = malloc(sizeof(Cell));
Grille* newGrille = alloue_grille(g->nombredeligne,g->nombredecolonne);
```

Puis on met dans la cellule au niveau de la grille là ou pointe l'adresse du pointeur (tableau double de caractères) la grille courante.
Puis on va au dernier élément de la pile (celui qui est marqué par NULL)

```
while(temp->nxt != NULL)
```

Puis on ajoute notre élément

```
temp->nxt = nouvelCell;
```

e) La fonction free_list

La fonction free_list prend en paramètre une List.
Elle effectue simplement un free de la liste.

```
free(liste);
```

f) La fonction remove_first

La fonction remove_first permet de supprimer le dernier élément de la pile. Elle prend en paramètre une liste et renvoie une structure tampon.
On va déclarer un tmp qui va pointer vers le dernier élément de la liste et un ptmp qui va pointer vers l'avant dernier élément de la ligne plus tard.

```
Cell* tmp = liste;
Cell* ptmp = liste;
```

Et un tampon pour retourner à la fois la grille (le tableau double de caractères) et la position de la cellule sur la liste.

S'il n'y a pas d'élément dans la liste

```
if(liste == NULL){
    t.liste = NULL;
    t.grille=liste->g;
    return t;
    }
```

Alors on retourne NULL.

S'il y a un élément dans la liste

```
if(liste->nxt == NULL)
{
    t.liste = NULL;
    t.grille = tmp->g;
    free(t.liste);
    return t;
}
```

Alors on retourne celui-ci.

Et sinon on va faire pointer tmp vers le dernier élément de la liste et ptmp vers l'avant dernier élément de la pile à la fin du while.

```
while(tmp->nxt != NULL)
{
    ptmp = tmp;
    tmp = tmp->nxt;
}
```

Puis on fait pointer l'avant dernier élément vers NULL puisque c'est le dernier élément.

```
ptmp->nxt = NULL;
```

Et on libère l'espace mémoire ce qui va supprimer le dernier élément.

```
free(tmp);
```

Et enfin on met dans le tampon ce qui nous intéresse c'est à dire la grille qui est présente dans l'avant dernier élément de la liste et la grille présente à l'avant dernier élément de la liste.

```
t.liste= liste;
t.grille=ptmp->g;
```

g) La fonction view

Elle s’appuie principalement sur la fonction debug de grille.c, elle prend en paramètre une liste, et affiche tous les éléments de celle-ci.

```
while(tmp != NULL)
{
    debug(tmp->g);
    tmp = tmp->nxt;
}
```

3.jeu.c

Jeu.c appelle toutes les fonctions nécessaires à l'exécution du programme c'est l'exécutable et il est présent dans le dossier /jdv_sinvani_ben/bin/

a) int fenetre[2]

Elle sera définie pour changer la taille de la fenêtre pour l'utilisateur toujours le confort en propriété !

```
int fenetre[2];
```

b) Tampon t

Pour renvoyer la grille et la position dans liste de l'élément précédent dans la liste.

c) Grille* grille

Va stocker la future grille soit dans le fichier default_grille ou d'un autre fichier comme vous désirez après tout.

d) int check_i

Même propriété qu'un boolean si c'est c'est vrai ça vaut quelque chose différent de 0

```
        if(check_i == 0){
        grille=lecteur(grille,"default_grille");
}
```

Si i n'est pas entrée on récupère le fichier default_grille.

e) Les options

```
case 'h':
/* -h or --help */
/* L'utilisateur a demandé l'aide-mémoire. L'affiche sur la sortie
standard et quitte avec le code de sortie 0 (fin normale). */
print_usage (stdout, 0);
break;
```

Si on tape -h on affiche la liste des options disponibles.

```
                    case 'n' :
                check_i = next_option;
                    /* -n or --nultiplier */
                    /* On appelle n fois la fonction applique_n_fois */
        grille=lecteur(grille,"default_grille");                    applique_n_fois(grille,atoi
(optarg));
                    List = add_first(List,grille);
```

Si on tape l'option -n on lit la grille par défaut et on applique n fois puis on commence à empiler dans la pile.

```
case 'o':
/* -o or --output */
/* On appelle la fonction sauver sur le fichier passé en argument*/
sauver(grille,optarg);
break;
```

Si on tape l'option -o on sauve dans le fichier mit en paramètre.

```
case '?':               /* L'utilisateur a saisi une option invalide. */
/* Affiche l'aide-mémoire sur le flux d'erreur et sort avec le code
de sortie un (indiquant une fin anormale). */
print_usage (stderr, 1);
break;
```

Si c'est un autre paramètre on affiche aussi comme -h pour permettre à l'utilisateur de saisir une bonne option.

f) Lancement du code

```
redefine_fenetre (fenetre);
gr_open_graph (" fenetre[0]xfenetre[1]");
```

On redéfinit la fenêtre pour le plaisir de l'utilisateur.

```
gr_clear_graph ();
affiche_grille (grille, gr_size_y (),
calcule_cote (grille,gr_size_x (), gr_size_y ()));
```

A chaque fois qu'on appuiera sur une touche on va nettoyer l'ancienne fenêtre afficher la nouvelle fenêtre en calculant le côté de la fenêtre courante.

```
case 'r':
affiche_grille (List->g, gr_size_y (),
calcule_cote (List->g,gr_size_x (), gr_size_y ()));
break;
```

L'option r permet de réajuster la fenêtre si on décide de l'agrandir ou de la rétrécir.

```
case 'n':
List = add_first(List,grille);
applique (grille);
break;
```

L'option n permet d'empiler dans la pile la cellule courante et ensuite on utilise la fonction applique pour continuer le jeu de la vie.

```
case 's':
sauver(grille,"save");
break;
```

L'option s permet de sauver la grille courante dans le fichier de destination nommé save

```
case 'b':
if(List != NULL){
t = remove_first(List);
List=t.liste;
grille= t.grille;
}
break;
```

L'option b permet de revenir en arrière elle retire de la pile la dernière cellule.

```
while (gr_read_key () != 'q');
sauver(grille,"save");
libere_grille(grille);
free_list(List);
```

Si l'utilisateur appuie sur q alors on sauve la grille dans save et on libère l'espace mémoire de la grille et de la liste.

g) Les bugs et remarques

Il existe un bug quand nous cherchons à inverser l'ordre des arguments, par exemple si

```
kali@kali:~/Téléchargements/jdlv_sinvani_ben$ ./bin/jeu -n 456 -i test
```

Le bug réside dans le fait que je n'ai pas su gérer le fait de stocker l'argument de i dans une variable quel que soit sa place. Du coup il faut impérativement taper -i « nom de fichier » et ensuite -n pour enchainer les options.

Je n'ai pas compris l'intérêt de la consigne suivante :

Ajoutez une option -o à votre porgramme prenant en paramètre un nom de fichier. Le programme écrira alors la grille courante à la fin du fichier lorsque l'utilisateur appuyera sur la touche s du clavier, ou lorsqu'il appuyera sur q (fin du programme).

Pourquoi vérifier -o ? puisque quand nous quittons le programme il le sauvegarde automatiquement dans le fichier de sauvegarde save. J'ai alors décidé de ne pas traiter la question mais je vais expliquer ici comment faire.
Tout se passe dans le fichier jeu.c.
Il faut d'abord déclarer la variable suivante

```
        int check_o=0; /*Permet de vérifier si l'option o est
là*/
```

Puis modifier la fonction comme cela

```
                        case 'n' :
                        /* -n or --nultiplier */
                        /* On appelle n fois la fonction
applique_n_fois */
if(check_o == 0){fprintf(stderr,"l'option -o n'est psa
présente\n");exit(1);}
if(check_i == 0) grille=lecteur(grille,"default_grille");
else{
                        check_i = next_option;
                        grille=lecteur(grille,argv[2]);
}
                        applique_n_fois(grille,atoi(optarg));
                        List = add_first(List,grille);
                        break;
```