TP du mercredi à 18h00
Groupe B
Félix Houdebert - TC02

Mini-projet : « Le jeu de la vie »

Création d'un algorithme et transcription en langage C

Présentation

Objectif

L'objectif était de réfléchir à un algorithme permettant de mettre au point le « *Jeu de la vie* », la simulation mathématique conçue par Conway. Il fallait ensuite traduire cet algorithme en un programme développé en **langage C**.
Le programme devait permettre à l'utilisateur de **placer des individus** (représentés par un 'x') sur les cases d'une **surface quadrillée de dimensions 30x30**, puis de lancer une simulation faisant **évoluer au fil des générations l'état de chaque case en fonction du nombre de cases voisines occupées par un individu.**

Principe

Le programme propose à l'utilisateur de lancer une simulation.
Si l'utilisateur accepte, il peut ensuite **entrer manuellement les coordonnées de chaque individu** présent à la génération de départ, ou **générer ces individus de manière pseudo-aléatoire** en indiquant seulement **leur nombre**.
Une fois la simulation initialisée, et tant qu'il reste des individus « en vie », l'utilisateur peut la faire avancer en **indiquant un certain nombre de générations**, ou y mettre fin.
Une fois une simulation terminée, l'utilisateur est invité à en lancer une nouvelle.

Mode d'emploi

1. Lancer une simulation

```
Voulez-vous lancer une simulation ? (o/n) :
O
```

Saisir 'O' permet de lancer une simulation. Saisir 'N' permet de quitter le programme.

2. Placer des individus

```
Voulez-vous que les individus soient generes automatiquement ? (o/n) :
```

Saisir 'O' permet de procéder à une génération pseudo-aléatoire des individus :

```
O
Saisir le nombre d'individus a generer (entier positif compris entre 0 et 900) : 200
```

L'utilisateur n'a qu'à entrer le nombre d'individus à générer. Ce nombre doit être compris entre 0 et 900 inclus.

Saisir 'N' permet de placer manuellement les individus :

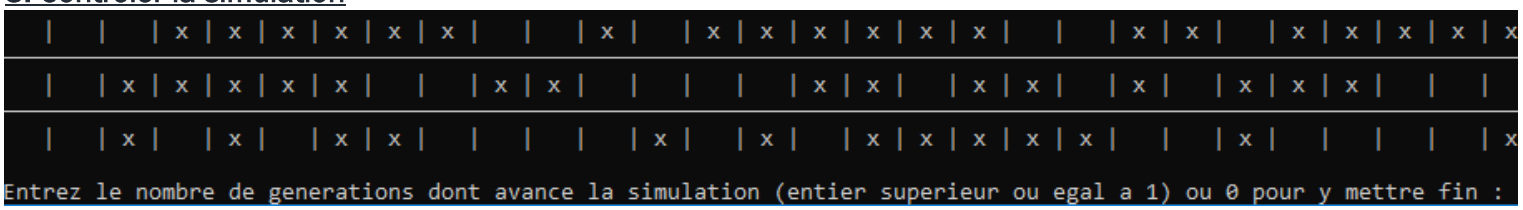
```
N

Saisir a la suite les coordonnees des individus presents initialement au format :
        "x , y" (avec x et y entiers naturels compris entre 1 et 30 inclus).
        Entrer les coordonnees d'une case contenant deja un individu le supprimera.
(Entrer 's' pour mettre fin a la saisie.)

2 : 3
Saisie incorrecte, veuillez reessayer :
2 , 3
s
```

L'utilisateur doit alors saisir les coordonnées de chacun des individus au format «x , y» (où x et y sont des entiers compris entre 1 et 30 inclus – car *on considèrere que les couples représentant les cases vont de (1 , 1) à (30 , 30).*
Saisir les coordonnées d'une case déjà remplie la videra.
Saisir 's' permettre de passer à l'étape suivante.

3. Contrôler la simulation



Tant qu'il reste des individus en vie, l'utilisateur est invité à faire avancer manuellement la simulation en saisissant un nombre de générations (qui doit être un entier supérieur ou égal à 1).
Saisir '0' mettra fin à la simulation.

Mise au point de l'algorithme

Spécifications

Données : A plusieurs reprises, l'utilisateur saisit (**au clavier**) les informations déterminant le déroulement de l'algorithme.
- *Au premier lancement et après chaque simulation,* **saisie d'un caractère** ('o' ou 'n').

- *A chaque simulation,* le mode de placement des individus : **saisie d'un caractère** ('o' ou 'n').
  - *En mode automatique*, le nombre d'individus générés : **saisie d'un entier** (compris entre 0 et 900).
  - *En mode manuel, la première fois et chaque fois qu'un individu est placé :* l'existence d'un individu et ses coordonnées : **saisie d'une chaîne de caractères** de la forme 's' ou «x , y » avec x et y des entiers compris entre 1 et 30.

- *A chaque simulation, a chaque fois que l'utilisateur demande une actualisation,* nombre de générations à simuler : **saisie d'un entier** (positif).

Résultats : Les résultats sont tous **affichés à l'écran**. Nous ignorerons ici les messages guidant l'utilisateur dans les simulations.
- *A chaque simulation, la première fois et après chaque actualisation* : le numéro de génération : **affichage d'un entier** (strictement positif).

- *A chaque simulation, la première fois et après chaque actualisation* : l'évolution de la population : **affichage d'un ensemble de caractères représentant une grille de dimensions 30x30.** (les cases qui ne sont pas situées en bordure de grille sont délimitées horizontalement par des '|' et verticalement par des '_'. Les '|' sont espacés de trois caractères. Pour les cases vides, il s'agit de trois caractères ' '. Pour les cases pleines, il s'agit de la succession des caractères ' ', 'x', et ' '. Après l'affichage de la 30e case de chaque ligne, affichage du caractère '\n'.

- *A chaque simulation et apres chaque actualisation* :
  - *S'il reste des individus en vie*, le nombre d'individus que compte la génération : **affichage d'un entier** (strictement positif).
  - *Si tous les individus sont morts,* numéro de la dernière génération : **affichage d'un entier** (strictement positif).

Analyse *(nous décrirons en détaill l'ensemble des fonctionnalités du programme afin de fournir à l'utilisateur son commentaire complet.*
En revanche, seuls les algorithmes correspondant aux fonctionnalités principales seront développés dans la partie Algorithmes).
Boucle principale : Nous aurons besoin d'un booléen : *lancer*. On lui affecte *vrai* ou *faux* selon le caractère saisi par l'utilisateur. Les instructions liées aux simulations se trouveront à l'intérieur d'une **boucle indéterminée** qui s'exécutera Tant Que *lancer* sera vrai.

Saisies : On aura recours à plusieurs types de lectures :
- Les saisies de caractères correspondant à des « yes no questions » (l'utilisateur saisit 'o' ou 'n') consistent en la **lecture d'un seul caractère.** On aura donc besoin d'un caractère : *c.*

Afin de **sécuriser la saisie** : le caractère entré sera converti en minuscules (en ajoutant 32 si il est compris entre 'A' et 'Z'), les caractères en surplus seront « nettoyés » de la console et si l'utilisateur saisit un caractère inattendu, on lui demande à nouveau. (la saisie sera donc effectuée à l'intérieur d'une **boucle indéterminée** qui se poursuivra tant qu'elle est incorrecte). Ces saisies seront effectuées depuis le SA « *yesno_questions* »

- La lecture des entiers (nombre d'individus à générer/nombre de générations à simuler) sera sécurisée par une vérification que le nombre saisi respecte les contraintes imposées par le concepteur : là encore, on l'effectue dans une boucle indéterminée.
De plus, les caractères excédentaires seront « nettoyés » afin de ne pas corrompre les prochaines saisies.
La saisie sera enregistrée directement dans la variable correspondante.

- Les coordonnées des individus placés manuellement seront lues sous la forme d'une chaîne de caractères. La saisie s'effectuera dans une boucle indéterminée afin d'éviter de demander le nombre d'individus à placer à l'utilisateur. On créera un tableau[1..15] de caractères, *saisie*, qui stockera la chaîne entrée.
Le processus de lecture suivra le schéma habituel : lecture du premier caractère, puis entrée dans une boucle indéterminée qui remplira le tableau de caractères tant que l'indice ne dépasse pas 15 et tant que le caractère '\n' ne se trouve pas dans la chaîne. Puis on affectera '\0' à la place du dernier caractère affecté à la chaîne.
Les caractères excédentaires seront là aussi « nettoyés » de la console.
La chaîne sera ensuite parcourue une première fois : si le caractère 's' est trouvé, on quittera la boucle et la saisie des individus prendra fin.
On parcourt une deuxième fois la chaîne : si on y trouve deux entiers compris entre 1 et 30, et séparés par une virgule, on affecte ces entiers dans des variables x et y qui serviront à modifier une case de la grille. Sinon, on demande à l'utilisateur de saisir de nouveau les coordonnées de son individu.
Cette saisie sera effectuée depuis un sous-algorithme : « *saisir_coord* ».

Surface : Pour la grille, nous utiliserons un tableau[1..30,1..30] de caractères : *surf*. La première dimension identifie la ligne à laquelle se trouve une case, la seconde désigne la colonne.
En pratique, il faudra convertir les coordonnées entrées par l'utilisateur, car en langage C, les tableaux commencent à l'indice 0.

Une deuxième grille similaire devra être utilisée pour l'actualisation de *surf* : en effet, les traitements doivent être effectués sur une copie de la surface car les cases encore non actualisées seraient affectées par les modifications des autres.

Cases : Les cases de *surf* seront remplies soient par le caractère '\0', qui indiquera que la case est vide, soit par un autre caractère pour indiquer qu'un individu y est présent.

Affichage de la surface : L'affichage de la surface se fait case par case, dans un sous-algorithme : « *afficherSurface* ». On parcourra l'ensemble de « *surf* » en utilisant deux boucles déterminées imbriquées, qui parcourront pour chaque ligne, chaque colonne du tableau à deux dimensions. Nous aurons donc besoin de deux itérateurs entiers : *i,j*. Ils iront de 0 à 30 non-inclus.
Pour chaque ligne de *surf* :
    Pour chaque colonne de *surf* :
        - Si la case est vide : affichage de trois espaces à l'écran.
        - Sinon : affichage d'un espace, suivi du caractère 'x', suivi d'un espace.
        - Si la case n'est pas située sur la dernière colonne : affichage du caractère '|'.
    Afficher '\n' pour faire un retour à la ligne.
    Si la ligne d'indice i n'est pas la dernière (càd la 29) : entrée dans une autre boucle déterminée. Pour j allant de 1 à 118 (nombre de caractères affichés avant), afficher le caractère 196 (ligne horizontale située à mi-hauteur).

Ainsi, nous affichons à l'écran une grille représentant le contenu de chaque case de notre surface, de largeur et de hauteur proches, et n'ayant pas de délimitations en bordure.

La grille sera affichée une première fois après le placement des individus, puis à chaque actualisation de la part de l'utilisateur.

Comptage du nombre de voisins : Tâche répétée un très grand nombre de fois donc effectuée à partir d'une fonction : *getNbVoisins*. Entrée : une surface 30x30, et deux caractères *xini* et *yini* correspondant aux coordonnées d'une case. Sortie : un caractère de valeur comprise entre 0 et 8. Le déroulement du sous-algorithme nécessite deux itérateurs entiers *i* et *j*, un caractère *nbv* (initialisée à 0), et deux caractères *x* et *y* correspondant aux coordonnées d'une case voisine.

Pour compter les individus voisins de la case (*xini*,*yini*), on se place d'abord à la case située en haut à gauche de la case initiale : (*xini*-1,*yini*-1).
A l'aide de deux boucles déterminées imbriquées, nous parcourons depuis cette case 3 lignes et 3 colonnes à chaque ligne : de cette façons nous avons accès aux 8 cases avoisinant notre case d'intérêt, ainsi que cette case elle-même.
Il faut ensuite vérifier si la case existe dans la surface : en effet, (*xini*, *yini*) est située en bordure, certaines cases n'existeront pas. De plus, nous devons ignorer (*xini*,*yini*) qui ne compte bien sûr pas parmi ses voisins.
Nous poserons donc la condition que x et y soient compris entre 1 et 30 (0 et 29, en pratique) et que *x* soit différent de *xini*, ou *y* soit différent de *yini*. Si cette condition est vérifiée, nous passons à l'étape suivante.
Enfin, il faut vérifier que la case (*x*,*y*) contienne un individu. Si c'est le cas, *nbv* est incrémenté.

Une fois sorti des deux boucles, on retourne *nbv*, qui aura pris la valeur du nombre de cases de *surf* occupées parmi le voisinage de (*xini,*yini).
Ce comptage est effectué pour les 900 cases à chaque fois qu'une génération est simulée ; elle est appelée depuis un autre sous-algorithme.

Actualisation de la surface : Tâche effectuée depuis un sous-algorithme : *actualiser*. Entrée : une surface 30x30, *plat*. Sortie : un booléen : *ilenreste* indiquant directement s'il reste des individus en vie (initialisé à *faux*), et un entier : *nb*, le nombre d'individus en vie (initialisé à 0). L'éxécution du sous-algorithme modifiera également la surface principale. Le sous-algorithme nécessite une deuxième surface 30x30, *plat_temp*, un caractère *nbv*, ainsi que deux itérateurs entiers, *x* et *y*.

Avant toute chose, *plat* est copiée dans *plat_temp* à l'aide de deux boucles déterminées imbriquées : en effet, les cases étant actualisées les unes après les autres, on risque d'affecter les cases non traitées si on actualise directement ses voisines. On travaillera donc sur une copie de notre surface principale.

Nous parcourons ensuite chacune des cases de *plat* à l'aide de deux autres boucles déterminées imbriquées. Pour chaque case, on affecte à *nbv* le caractère de sortie de la fonction *getNbvoisins* (en comptant bien les voisins dans *plat* et non sa copie). Nous traitons ensuite la case correspondante dans *plat_temp* selon la valeur de *nbv*, suivant les règles du jeu de la vie :

- Si *nbv* vaut 3, un individu survit ou naît : on assigne le caractère 'x' à la case de *plat_temp*. *nb* est incrémenté.
- Si *nbv* vaut 2 : si la case est non vide : un individu survit. On affecte quand même de nouveau 'x' à la case de *plat_temp*. *nb* est incrémenté.
- Sinon : la case a moins de 2 voisins, ou plus de 3 voisins : pas de naissance, et mort de l'éventuel individu occupant la case : on y affecte le caractère 0.

Enfin, nous copions chaque case de *plat_temp* dans *plat* à l'aide de deux nouvelles boucles déterminées pour mettre à jour notre surface principale une fois tous les traitements effectués. Si *nb* est strictement positif, *ilenreste* prend la valeur *vrai*. Ces deux variables sont retournées en paramètres de sortie.

Ce sous-algorithme est appelé pour chaque génération, excepté la première.

Remplissage pseudo-aléatoire de la surface : Cette fonctionnalité n'était pas demandée, elle sera donc survolée. Elle est remplie depuis un sous-algorithme *remplissage_auto* n'ayant qu'un paramètre d'entrée, un tableau[1..30,1..30] de caractères.
Après avoir lu un nombre d'individus *nb_indiv* correspondant aux attentes du concepteur, le programme répétera

nb_indiv fois les opérations suivantes :
génération pseudo-aléatoire (basée sur l'heure de lancement du programme) de deux entiers x et *y* compris entre 0 et 29.
Vérification que ces coordonnées n'ont pas déjà été choisies en parcourant un tableau *tab_dejapris* contenant, pour chaque individu déjà généré, les coordonnées où il a été placé.

- Si la case est disponible : placement de l'individu à la case (*x*,*y*) de la grille.
  Enregistrement des coordonnées dans *tab_dejapris*.
- Sinon : génération d'un nombre pseudo-aléatoire d'un booléen.
  Selon sa valeur, on affectera à *x* : (x+1)%30 , ou à y : (y+1)%30. Répétition de la vérification jusqu'à avoir les coordonnées d'une case libre.

Remplissage manuel de la surface : Cette fonctionnalité est effectuée depuis le sous-algorithme s*aisir_coord* prenant en entrée *plat* : un tableau de caractère à deux dimensions.
Le mécanisme de saisie des coordonnées a déjà été décrit précédemment.

On enregistre les coordonnées lues, si elles sont correctes, dans les entiers *x* et *y*.

- Si *plat*[*x*][*y*] est vide, on y place un individu en lui affectant le caractère 'x'.
- Sinon, on supprime l'individu présent en affectant à la case le caractère 0.

L'opération est répétée à chaque nouvelle simulation et tant que l'utilisateur n'a pas saisi, au lieu des coordonnées d'une case, le caractère 's'.

Algorithme principal : *On considèrera ici une version simplifiée de l'algorithme permettant d'assurer uniquement les fonctions essentielles indiquées dans le sujet.*
L'algorithme principal utilisera notamment un tableau[1..30,1..30] de caractères, *surf*, de deux itérateurs entiers *i* et *j*, des entiers *g,*et *dg* pour stocker la génération actuelle et le nombre de générations à faire passer, ainsi qu'un booléen *lancer* .

On initialisera un booléen *lancer* en lisant un caractère 'o' (correspondant à *vrai*) ou 'n' (correspondant à *faux*)
Il y aura ensuite une boucle indéterminée : tant que *lancer* est vrai, nous procédons à une simulation.
On commencera par initialiser/réinitialiser les variables utilisées dans une simulation : il faut remplir *surf* de caractères '\0' à l'aide de deux boucles déterminées imbriquées.
On initialise également *g* à 1 et *dg* à 0.
On affichera ensuite à l'écran les conditions à respecter pour la saisie des coordonnées, puis on appellera *saisir_coord*.
Une fois la grille remplie par l'utilisateur, on affichera la génération 1, en affichant g puis en appelant *afficherSurface* avec comme paramètre *surf*.

On entre ensuite dans une boucle indéterminée qui s'éxécute tant que dg est différent de 0.
On demande d'abord à l'utilisateur de saisir le nombre de générations dont avance la simulation, un entier strictement positif, ou 0 s'il veut terminer la simulation.
On lira sa saisie dans *dg*.
Si *dg* > 0 :
On entre dans une boucle déterminée, dont l'itérateur va de 1 à *dg* par pas de 1.
A chaque itération, on appelle *actualiser* sur *surf* afin de « faire passer une génération ».
Puis on incrémentera *g*.
Enfin, on procèdera à l'affichage de la génération *g* en affichant *g* lui-même puis en appelant *afficherSurface* sur *surf*.
On réaffecte enfin *lancer* avec un nouveau message et une nouvelle lecture.
Une fois sorti de cette boucle, nous saluerons l'utilisateurs et l'algorithme prendra fin.

Algorithmes *(adaptés aux conventions de NF04)*

ALGORITHME **algoprincipal**
VARIABLES :
- *surf* : tableau[1..30,1..30] de caractères
- *lancer* : booléen
- *i,j,g,dg* : entiers
- *c* : caractère

INSTRUCTIONS :
```
 1 Ecrire(« Souhaitez-vous commencer une simulation ? (‘o’\’n’) » !)
 2 Lire(clavier ! c)
 3 Si c = ‘o’ Alors :
 4       lancer <- vrai
 5 Sinon si c = ‘n’ Alors :
 6       lancer <- faux
 7 FinSi
 8 TantQue lancer Faire :
 9       g <- 1
10       dg <- 0
11       Pour i allant de 1 à 30 Faire : //remplissage de la grille par des cases vides
12               Pour j allant de 1 à 30 Faire :
13                       surf[i][j] <- ‘\0’
14               FinPour
15       FinPour
16       Ecrire(« Saisir les coordonnees au format ‘x , y’ avec x et y des entiers compris entre 1 et 30, ou ‘s’ pour arrêter » !)
17       saisir_coord(surf !)
18       Ecrire(« Generation »,1 , « : » !)
19       afficherSurface(surf !)
20       TantQue dg>0 Faire :
21               Ecrire(« Saisir le nombre de générations à simuler (un entier positif) ou 0 pour terminer la simulation » !)
22               Lire(clavier ! dg)
23               Si dg > 0 :
24                       Pour i allant de 1 à dg Faire :
25                               actualiser(surf!)
26                               g <- g+1
27                       FinPour
28               FinSi
29               Ecrire(« Generation »,g, « : » !)
30               afficherSurface(surf !)
31       FinTantQue
32       Ecrire(« Souhaitez-vous lancer une nouvelle simulation ? (‘o’\’n’) » !)
33       Lire(clavier ! c)
34       Si c = ‘o’ Alors :
35               lancer <- vrai
36       Sinon si c = ‘n’ Alors :
37               lancer <- faux
38       FinSi
39 FinTantQue
40 Ecrire(« Au revoir ! » !)
41 Fin algoprincipal
```

SOUS-ALGORITHME **saisir_coord**
//Sert à saisir manuellement les coordonnées d’un individu
PE : *plat* : tableau[1..][1..] de caractères.
VARIABLES :
- *saisie* : tableau[1..15] de caractères.
- *c* : caractère
- *i,x,y* : entiers
- *continuer* : booleen

```
INSTRUCTIONS :
 1 i <- 1
 2 Lire(clavier ! saisie[1])
 3 TantQue saisie[i] != '\0' ET i<15 Faire :
 4       i <- i+1
 5       Lire(clavier ! saisie[i])
 6 FinTantQue
 7 saisie[i] <- '\0'
 8 continuer <- vrai
 9 TantQue continuer Faire :
10       continuer <- vrai
11       //
12       i <- 1
13       TantQue saisie[i] != '\0' ET continuer Faire:
14                Si saisie[i] = 's' Alors :
15                         continuer <- faux
16                FinSi
17                i <- i +1
18       FinTantQue
19       Si continuer Alors : //si s n'a pas été entré, affectation d'une case de la grille
20                Lire(saisie ! x,y)
21                Si plat[x][y] = '\0' Alors :
22                         plat[x][y] <- 'x' //si la case est vide, placement d'un individu
23                Sinon :
24                         plat[x][y] <- '\0' //si la case est pleine, suppression de son individu
25                FinSi
26       FinSi
27 FinTantQue
28 Fin saisir_coord

SOUS-ALGORITHME getNbVoisins
//Sert à compter le nombre de voisins d'une case de coordonnées (xini,yini)
PE : plat : tableau[1..][1..] de caractères ; xini,yini : caractères
PS : nbv : caractère
VARIABLES : i,j,x,y : caractères
INSTRUCTIONS :
 1 nbv <- 0
 2 x <- xini – 1 // on se place à la case en haut à gauche de (xini,yini)
 3 Pour i allant de 1 à 3 Faire : //on va lire 3 cases sur chaque ligne, sur 3 lignes.
 4       y <- yini – 1
 5       Pour j allant de 1 à 3 Faire :
 6                //On doit vérifier que la case existe et que ce n'est pas la case dont on compte les voisins
 7                Si( ((x >= 1) ET (x <= 30) ET (y >= 1) ET (y <= 30)) ET (x != xini OU y != yini) ) Alors :
 8                         Si plat[x][y] != '\0' Alors :       //S' il y a un individu sur (x,y)
 9                                  nbv <- nbv+1 //on incrémente le nombre de voisins
10                         FinSi
11                FinSi
12                y <- y+1
13       FinPour
14       x <- x+1
15 FinPour
16 Fin getNbVoisins

SOUS-ALGORITHME actualiser
//Sert à actualiser une case en fonction de son nombre de voisins
PE : plat : tableau[1..][1..] de caractères
VARIABLES :
          - plat_temp : tableau[1..30][1..30]de caractères
```

```
        - x,y,nbv : caractères
 1 Pour x allant de 1 à 30 Faire : //on commence par copier plat dans plat_temp
 2      Pour y allant de 1 à 30 Faire :
 3            plat_temp[x][y] <- plat[x][y]
 4      FinPour
 5  FinPour
 6 FinTantQue
 7 //on balaye ensuite plat pour compter les voisins de chacune de ses cases
 8 Pour x allant de 1 à 30 Faire :
 9      Pour y allant de 1 à 30 Faire :
10            getNbVoisins(plat,x,y ! nbv) //on compte les voisins de la case (x,y) et on enregistre leur nombre dans nbv
11            Si nbv = 3 Alors : //survie ou naissance d'un individu sur la case
12                  plat_temp[x][y] <- 'x'
13            Sinon Si nbv = 2 :
14                  Si plat[x][y] != '\0' Alors : //survie d'un individu déjà présent sur la case. Aucun traitement requis
15                  FinSi
16            Sinon : //mort d'un individu ou absence de naissance.
17                  plat_temp[x][y] <- '\0'
18            FinSi
19      FinPour
20  FinPour
21 //On recopie enfin plat_temp dans plat pour « publier » les modifications :
22 Pour x allant de 1 à 30 Faire :
23      Pour y allant de 1 à 30 Faire :
24            plat[x][y] <- plat_temp[x][y]
25      FinPour
26 FinPour
27 Fin actualiser

SOUS-ALGORITHME afficherSurface
//Sert à afficher une grille représentant notre population
PE : plat : tableau[1..][1..] de caractères ;
VARIABLES : i,j : caractères
INSTRUCTIONS :
 1 Pour i allant de 1 à 30 Faire :
 2      Pour j allant de 1 à 30 Faire :
 3            Si plat[i][j] = '\0' Alors :
 4                  Ecrire(«   » !) //affichage d'un espace vide
 5            Sinon :
 6                  Ecrire(« x » !) //affichage d'un individu représenté par un 'x'
 7            FinSi
 8            Si j != 30 Alors : //on n'affiche pas de séparateur aux bordures horizontales de la grille
 9                  Ecrire(«|» !)
10            FinSi
11       FinPour
12       Ecrire(«\n» !) //Passage à la ligne
13       Si i != 30 Alors : //on n'affiche pas de séparateur aux bordures verticales de la grille
14            Pour j allant de 1 à 118 Faire : //Il faut afficher le séparateur 118 fois pour couvrir la largeur d'une ligne
15                  Ecrire(«—»!)
16            FinPour
17      FinSi
18 FinPour
19 Ecrire(«\n»!)
20 Fin afficherSurface
```

Observations

Correspondance des langages : Le langage C, bien que bas niveau, est plus permissif que le langage utilisé en NF04 : il permet par exemple l'utilisation de boucles do{*instructions*}while(*instructions*) ; exécutant une première fois les instructions avant de rentrer dans la boucle indéterminée.
Penser à ces différences et annoter des raccourcis en commentaires lors de la conception de l'algorithme m'a permis de gagner beaucoup de temps pour sa transcription en C.

Temps passé : J'ai passé environ une heure à créer un brouillon des algorithmes permettant d'assurer les fonctions essentielles du projet, et quarante minutes à les mettre en pratique sous la forme d'un programme C.

Il m'a ensuite fallu plus d'une heure pour complexifier la structure du programme en y ajoutant des saisies sécurisées et certaines fonctionnalités comme l'affichage du nombre d'individus en vie à chaque génération, la génération aléatoire d'individus et l'arrêt automatique de la simulation après la mort de tous les individus.

Afin d'avoir un programme avec une structure plus « propre », j'essayerai la prochaine fois de penser à ces ajouts avant l'écriture du programme. Ce sera pour NF05 !

Améliorations : Force est de constater qu'apporter des améliorations au projet, en terme d'affichage, en terme de fonctionnalités ou en terme de sécurisation, complexifie grandement l'écriture et la lecture du rapport …

Je n'améliorerai donc pas le programme que j'ai mis au point – mais voici les différentes fonctionnalités qui pourraient y être ajoutées :

- Un mode « défilement automatique », avec une vitesse choisie par l'utilisateur, qui ferait passer les générations une par une automatiquement pour créer une animation fluide.

- La possibilité pour l'utilisateur d'afficher le caractère de son choix à la place de 'x' pour représenter les individus.

- La création d'une version avec une surface de dimensions choisies par l'utilisateur – mais cela obligerait à repenser toute la structure du programme pour utiliser des allocations dynamiques de la mémoire et à entrer la taille de la grilles dan les fonctions de traitement.

- La possibilité de signaler à l'utilisateur qu'une génération est pérenne et que son nombre d'individus restera définitivement le même au fil des générations.