Chapitre 8
Concevoir et réaliser un programme

1. Déterminer le sujet et définir des objectifs

Avec les tableaux, les structures et les fonctions, il devient possible de faire des programmes assez importants éventuellement répartis sur plusieurs fichiers sources. Par exemple des versions simples de beaucoup de jeux classiques, Pacman, Casse-brique, Space Invaders, Tétris, etc. De tels programmes supposent une réflexion préalable : avec quelles variables et quelles fonctions et selon quel algorithme général je fais mon programme ? Mais dans le cas de programme que l'on imagine sur des sujets que l'on découvre, avant de pouvoir répondre définitivement à ces questions il est nécessaire de poser une hypothèse de départ et de se donner des objectifs à atteindre. Ces objectifs pourront évoluer au fur et à mesure du développement du projet et l'hypothèse de départ sera ou non corrigée en fonction des résultats obtenus.

L'idée ici est de réaliser un automate cellulaire 2D en mode console. Qu'est-ce qu'un automate cellulaire ? Quel fonctionnement mettre en œuvre ? Avec quelle structure de données ? Le premier point est de créer un espace de connaissance propre au projet sur lequel s'appuyer ensuite pour le réaliser.

1.1 Principe de l'automate cellulaire

Un automate cellulaire traduit le comportement d'un ensemble d'individus, de cellules, de points, etc. Il repose sur des règles simples exercées localement pour chaque individu en fonction de ses voisins immédiats. C'est le mouvement d'ensemble produit par toute la population soumise aux mêmes lois qui intéresse. Cette figure articule l'espace et le temps dans une évolution et elle n'est pas mathématisable parce que trop complexe. Le seul moyen de l'observer ou de l'utiliser est de la programmer pour un ordinateur. Pour cette raison c'est une figure emblématique de l'informatique. L'automate cellulaire se retrouve dans des domaines différents comme la modélisation de feux de forêt, l'évolution de population dans des villes, mais aussi dans le domaine graphique pour la richesse étonnante des figures qu'il peut générer.

Un chercheur de l'université de Paris 8, Pierre Audibert, le décrit ainsi : "À chaque étape de temps, tous ces individus évoluent en même temps, en fonction de leur situation locale. Le processus est typiquement parallèle (...) soumis à des règles simples dictées par leur voisinage immédiat, ils donnent lieu à des phénomènes complexes et contrastés. Apparitions localisées de formes organisées à partir d'un contexte aléatoire, ou inversement, développement de mouvements désordonnés et complexes qui cependant obéissent à des lois d'ensemble, partout règne l'harmonie des contraires. Cela concerne aussi bien des populations que l'évolution de formes dans le monde naturel, et même les incendies de forêt".

Comment, à partir de ce texte, concevoir un ou plusieurs programmes graphiques, juste pour voir ?

1.2 Fonctionnement envisagé

Nous allons nous appuyer sur un moteur d'organisation qui s'inscrit dans la tradition initiée dans les années soixante par John Horton Conway et nommée "Jeu de la vie".

Le principe est le suivant :

Chaque position dans le plan est dotée d'une valeur soit de zéro soit de un.

Au départ toutes les positions du plan sont mises à zéro et une fonction d'initialisation permet d'en mettre quelques-unes à un.

Ensuite toutes les positions du plan sont passées en revue. Pour chacune d'entre elles, les positions voisines sont regardées et les valeurs trouvées comptabilisées, ce qui donne par exemple une situation comme :

(x-1, y-1) avec valeur 0	(x, y-1) avec valeur 1	(x+1, y-1) avec valeur 1
(x-1, y) avec valeur 1	(x, y) avec valeur 0	(x+1, y) avec valeur 0
(x-1, y+1) avec valeur 0	(x, y+1) avec valeur 0	(x+1, y+1) avec valeur 0

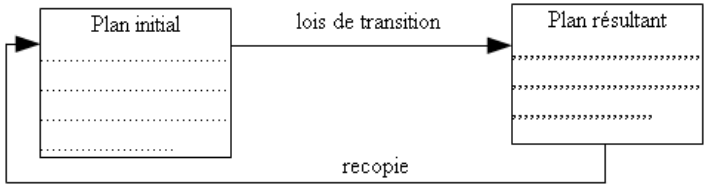
(x, y) au centre est la position courante : x est pour la position horizontale et y pour la position verticale. Il y a 3 positions adjacentes qui valent 1.

La valeur de la position courante va évoluer ou rester identique en fonction d'une règle de transition. La règle de transition repose sur les valeurs trouvées dans les positions voisines. Dans l'exemple ci-dessus, la position courante comptabilise trois 1 trouvés dans les cellules adjacentes. En quelque sorte il y a trois voisins. La règle de transition repose sur ce nombre de voisins.

Par exemple on aura une règle du type : si le nombre de voisins est inférieur à deux ou si le nombre de voisins est supérieur à trois, c'est-à-dire si le nombre de voisins est différent de 2 ou de 3, alors la position courante prend une valeur de un, sinon elle prend une valeur de zéro.

On peut imaginer d'autres règles, d'une façon générale le principe est : pour la position courante, si le nombre de voisins est inférieur à "un plancher" ou si le nombre de voisins est supérieur à "un plafond", alors la position courante prend la valeur booléenne "v1", sinon elle prend la valeur booléenne complémentaire "v2".

Il convient de préciser que le calcul se fait selon les positions du plan mais que les résultats, pour chaque position, sont stockés au fur et à mesure dans un plan équivalent en miroir du premier. Une fois que les valeurs de chaque position ont été recalculées et stockées dans le plan réservé aux résultats, ce dernier est recopié dans le plan initial et l'opération recommence :



2. Trouver une structure de données valable

Voilà pour le principe de l'automate mais comment en faire un programme ? Comment traduire ce processus en une suite d'instructions dans un langage pour la machine ?

Une fois l'idée sur laquelle partir est posée, le premier point est de réfléchir à une possibilité de structure de données. Sur quoi va s'appuyer le moteur de l'automate pour fonctionner ? C'est-à-dire quel type de données peut-on choisir pour coder l'automate ? Variables simples ? Structure ? Tableaux ?

En l'occurrence cela saute aux yeux ici et ce n'est pas très compliqué. Essentiellement il y a deux surfaces 2D de booléens, des matrices d'entiers feront très bien l'affaire, une pour le plan initial et une autre pour le plan résultant. La taille sera définie par deux macros constantes, ce qui donne :

```
#define TX  80
#define TY  50

int MAT[TY][TX];
int SAV[TY][TX];
```

3. Identifier les fonctions principales

Maintenant que nous savons à partir de quoi travailler, nos deux matrices, quelles sont les opérations à réaliser afin de mettre en œuvre le principe de fonctionnement que nous avons retenu ? C'est-à-dire quelles sont les opérations à faire et dans quel ordre ?

Il s'agit ici en s'appuyant sur nos deux matrices de repérer et d'imaginer un algorithme efficace pour le projet.

1) A priori la première étape est une étape d'initialisation :

– Au début les deux matrices sont initialisées à 0. Quelques positions du plan initial, la matrice MAT sont mises à 1. Écrire une fonction d'initialisation.

2) Ensuite le moteur tourne en boucle et à chaque cycle il doit :

– Afficher le plan initial.

– Calculer la transition pour chaque position et sauver le résultat dans SAV.

– Recopier la matrice SAV dans le plan initial MAT.

Il se dégage déjà quatre fonctions à écrire :

– une fonction d'initialisation,

– une fonction d'affichage du plan,

– une fonction de calcul,

– une fonction de recopie.

Initialisation, affichage et recopie sont simples. La fonction de calcul nécessite un petit zoom :

Calculer c'est passer en revue (boucle) toutes les positions du plan initial et :

1 - pour chaque position compter le nombre des positions voisines à 1,

2 - appliquer la loi de transition en fonction de ce nombre,

3 - stocker le résultat dans la matrice SAV.

Sur ces trois étapes dans l'algorithme, la première, le comptage des positions voisines peut faire l'objet d'une fonction à part. Cette fonction pourrait prendre en paramètre la position courante dans la matrice initiale et retourner le nombre de positions voisines à 1 trouvées tout autour.

La seconde étape pourrait aussi faire l'objet d'une fonction à part. Par exemple, si le moteur était destiné à fonctionner selon différents paramétrages des lois de transition. Ces paramétrages pourraient alors être passés en paramètre.

Mais pour l'heure le moteur n'aura qu'un seul type de loi de transition. Le traitement dans la fonction calcul n'aura donc besoin que du comptage des positions voisines à 1 : une cinquième fonction, la fonction compte voisins.

4. Choisir le niveau des variables fondamentales

Maintenant nous avons une hypothèse de structure de données et une hypothèse des fonctions à écrire, il reste juste un point à décider : est-ce que les matrices sont des variables globales visibles de toutes les fonctions, ou sont-elles encapsulées localement dans le main(), ce qui nécessite une transmission via les paramètres ?

Quelle est la différence du point de vue du développement du projet ? Quels sont les avantages ou inconvénients de l'une et de l'autre ?

Si nos matrices sont globales, il n'y a pas besoin de les passer en paramètre et cela allège un peu l'écriture des fonctions. C'est un petit programme qui tient sur un seul fichier, aucune confusion n'est possible avec d'autres processus à l'œuvre dans le même programme. Un problème pourrait éventuellement survenir ultérieurement si le développement du projet portait à faire tourner simultanément plusieurs automates et gérer plusieurs plans initiaux. Dans ce cas, il serait préférable de pouvoir passer au moins le plan initial en paramètre.

Cependant, il y aurait aussi la possibilité de partir sur un tableau de matrices déclaré en global et de modifier un peu chaque fonction en ajoutant une boucle pour NB matrices :

```
#define NB            10
int NBMAT[NB][TY][TX];   // 10 matrices de TY par TX
```

Mais a priori, pour ce que nous voulons faire maintenant, il n'y a pas d'obstacle à utiliser deux matrices en global.

Les deux matrices et les macros sont déclarées en début de fichier après les inclusions nécessaires avant le main() et initialisées à 0 :

```
#include <stdio.h>
#include <stdlib.h>
#include <conio.h>
#include <windows.h>

#define TX  80
#define TY  50

int MAT[TY][TX] = {0};
int SAV[TY][TX] = {0};
```

Remarque

En général, toutes les variables globales sont spécifiées par des noms en majuscules, c'est une convention.

5. Écrire les fonctions

5.1 Fonction d'initialisation

La fonction d'initialisation est simple : mettre à 1 quelques positions choisies.

```
void init_matrice(void)
{
   // quatre positions au centre qui sont mises à 1 (le reste est à 0)
   MAT[TY/2][TX/2]=1;
   MAT[TY/2+1][TX/2]=1;
   MAT[TY/2][TX/2+1]=1;
   MAT[TY/2+1][TX/2+1]=1;
}
```

Voici la même fonction avec possibilité de transmission de différentes matrices de même taille (TY et TX) en paramètre :

```
void init_matrice(int M[][TX])
{
   M[TY/2][TX/2]=1;
   M[TY/2+1][TX/2]=1;
   M[TY/2][TX/2+1]=1;
   M[TY/2+1][TX/2+1]=1;
}
```

5.2 Fonction d'affichage

La fonction d'affichage utilise les fonctions gotoxy() et textcolor(). Le principe est simple : parcourir la matrice et pour chaque position, si la valeur est 1 colorer le fond en rouge, si la valeur est 0 colorer en bleu et dans les deux cas afficher un espace.

```
void affiche(void)
{
int x,y ;

   for (y=0; y<TY; y++){     // pour chaque position
      for (x=0 ;x<TX ;x++){
         gotoxy(x,y);        // déplacer le curseur à la position
         if(MAT[y][x]==1)    // si valeur 1
           textcolor(192);   // couleur rouge en fond
         else
           textcolor(16);    // sinon couleur bleu en fond
         putchar(' ');       // affiche un espace
      }
   }
```

```
}
/*****************************************************************
*****************************************************************/
void gotoxy(int x, int y)
{
HANDLE h=GetStdHandle(STD_OUTPUT_HANDLE);
COORD c;
   c.X=x;
   c.Y=y;
   SetConsoleCursorPosition(h,c);
}
/*****************************************************************
*****************************************************************/
void textcolor(int color)
{
HANDLE h=GetStdHandle(STD_OUTPUT_HANDLE);
   SetConsoleTextAttribute(h,color);
}
/*****************************************************************
*****************************************************************/
```

5.3 Fonction de calcul

La fonction du calcul fait appel à la fonction de comptage des voisins qui a besoin de la position courante en paramètre et retourne le nombre des voisins trouvés (les positions à 1). Pour chaque position, la loi de transition est appliquée et donne 0 si le nombre de voisins est inférieur à 2 ou si le nombre de voisins est supérieur à 3 (si le nombre de voisins est différent de 2 et 3) et donne 1 dans tous les autres cas.

```
void calcul(void)
{
int x,y,nb_voisins;

   // pour chaque position dans la matrice (sans compter le
   // pourtour à cause de la recherche des voisins : de 1 à TY-1
   // et non de 0 à TY, idem pour x)
   for (y=1; y<TY-1; y++){
      for (x=1 ;x<TX-1 ;x++){

         // récupération du nombre de voisins à 1
         nb_voisins = compte_voisins(y,x);

         // application de la loi de transition basique et
         // stockage du résultat dans la matrice SAV
         if (nb_voisins <2 || nb_voisins>3)
            SAV[y][x]=0;
```