TP1: Tableaux statiques et automates cellulaires

Vladislav Tempez - MP2I - Lycée Descartes

14 septembre 2022

1 Consignes préalables

Pour ce TP et pour le reste de l'année, il vous est demandé d'utiliser des noms de variables et de fonctions qui sont pertinents et suffisamment explicites. Le choix d'un nom pour une fonction ou une variable n'est pas une chose aisée, et dépend fortement du contexte. Il est ainsi tout à fait acceptable de nommer une variable de boucle `i`, alors que nommer une fonction ainsi est bien plus problématique. Par cohérence avec les mots clé du langage et pour s'habituer à une bonne pratique il vous suggéré de nommer les variables et fonction en anglais, et de choisir comme convention séparer les mots dans un nom de variable par des `_` . Les noms de fonction et de variables proposés dans ce TP et dans le code source proposé comme correction vont dans ce sens.

Par ailleurs, si en `C` la structure du code n'a aucun impact à la compilation, pour des raisons de lisibilité pour vous-même et vos éventuels correcteurs, il vous est demandé de faire apparaître la structure du code via des espaces, des sauts de ligne, etc, c'est-à-dire d'indenter celui-ci. Essayer d'utiliser les mêmes conventions dans l'ensemble de vos codes sources.

L'usage des options de compilation n'est pas à voir comme une contrainte dans l'écriture de vos programmes mais comme une aide à l'écriture de programme corrects. Les messages d'erreurs et d'avertissement (warnings) sont des outils important pour prévenir et corriger des bugs. **Il ne faut pas ignorer ces messages d'erreurs ni les avertissements.** Il est donc attendu de vous que vous compiliez vos programmes avec les options :

1. `-Wall`
2. `-Wextra` qui ajoutent toutes deux des avertissements concernant des morceaux de code qui contiennent probablement une erreur
3. `-o <nom de l'exécutable>` qui permet de nommer l'exécutable produit
4. `-fsanitize=address,undefined` qui permet d'obtenir des messages d'erreurs plus complets lors d'une erreur d'exécution.

2 Tableaux en C

2.1 Pourquoi utiliser des tableaux

Il est parfois intéressant d'effectuer un traitement algorithmique identique sur un nombre important de variables. Afin d'automatiser ce traitement, il est nécessaire de pouvoir accéder à ces variables de manière automatique, par exemple via une boucle. Des variables déclarées de manière individuelle ne permettent pas ceci.

L'usage de tableau permet d'obtenir une collection de variables indexées par un entier auxquelles il est possible d'accéder et de traiter via une boucle. Dans ce TP, nous allons manipuler des tableaux dont la taille est **statique**, c'est-à-dire **fixée au moment de la compilation**, par opposition à des tableaux dynamiques dont la taille est fixée à l'exécution.

Nous verrons plus tard dans l'année comment manipuler des tableaux dont la taille est dynamique.

2.2 Syntaxe des tableaux statiques en C

En C les tableaux sont fortement typés, comme le sont les variables. Cela signifie qu'ils ne peuvent contenir qu'un seul type de valeur. La déclaration d'un tableau se fait avec la syntaxe suivante :

```
<type> <nom_du_tableau>[<taille_du_tableau>];
```

Attention, la taille du tableau doit sous la forme d'une expression qui ne contient pas de variable, `3*5` convient, mais pas `2*n`, même si `n` est une variable déjà déclarée et définie dont la valeur ne change pas.

Un exemple de déclaration de tableau où on déclare un tableau de 5 `double` nommé `some_double_array` :

```
double some_double_array[5*5];
```

L'initialisation des valeurs d'un tableau peut être faite de deux manière. La première manière ne peut être réalisée que lors de la déclaration du tableau et consiste à fournir les éléments dans l'ordre, séparés par des virgules :

```
int some_int_array[3] = {1, 2, 3};
```

ou bien

```
double some_double_array[5*5] = { 0 };
```

pour initialiser tous les éléments à 0.

Pour accéder aux éléments d'un tableau on peut utiliser la syntaxe

```
<nom_du_tableau>[<indice>]
```

par exemple on accède au 3e élément du tableau `some_int_array` avec la syntaxe :

```
some_int_array[2]
```

Ici, le troisième élément est indicé par 2 car les indices comment à 0.

On peut respectivement récupérer cette valeur dans une variable ou la modifier :

```
int a = some_int_array[2];
some_int_array[2] = 2*a;
```

Les tableaux se comportent différemment des variables classiques : il n'est pas possible de le renvoyer en tant que valeur, mais les modifications appliquées lors de l'exécution d'une fonction persistent à la sortie de la fonction.

On peut illustrer ceci avec un exemple

```c
#include <stdio.h>

void incr(int x){
  x = x+1;
}

void incr_array(int array[], int array_size){
  if (0 <taille_t){
    t[0] = t[0]+1;
  }
}
int main(){
  int x = 0;
  int t[3] = { 0 };
  printf("x before incr %d\n",x);
  incr(x);
  printf("x after incr %d\n",x);
  printf("a_0 before incr %d\n",t[0]);
  incr_tab(t);
  printf("a_0 after incr %d\n",t[0]);
}
```

qui donne :

```
x avant 0
x après 0
t_0 avant 0
t_0 après 1
```

On dit que `x` est passé par **valeur** alors que `t` est passé par **paramètre**. Le détail de ce mécanisme sera expliqué quand nous aborderons les pointeurs. Une conséquence de ceci (dont l'explication précise n'est pas donnée ici) est qu'on ne peut pas utiliser un tableau déclaré dans une fonction une fois sorti de la fonction. En particulier, **une fonction ne peut pas renvoyer de tableau** pour le moment.

On peut noter que pour déclarer que l'argument d'une fonction est de type `int` on indique `int <nom_du_tableau>[]` dans les arguments de la fonction. Il n'est pas nécessaire d'indiquer entre [] la taille du tableau.

Cependant, il n'est pas directement possible de connaître la taille du tableau à partir de celui-ci. Il est donc **impératif de passer en argument de toute fonction qui manipule un tableau la taille de ce tableau**.

2.3 Quelques exercices de syntaxe sur les tableaux

1. Dans la fonction main, déclarez un tableau d'entier de taille 10 et initialisez-le de manière à ce que la case `i` contienne `i` (à l'aide d'un `for`).
2. Écrivez une fonction `void print_int_array(int t[], int array_size)` qui affiche les éléments d'un tableau d'entier sur une même ligne et un retour à la ligne en fin de tableau.
3. Écrivez une fonction `void init_array(int array[], int array_size)` qui initialise un tableau de manière à ce que la case `i` contienne `f(i)` (à l'aide d'un `for`) où `int f(int i)`

renvoie la plus grande puissance de 2 inférieure à `i`, c'est-à-dire `k` tel que $2^k \leq i \leq 2^{k+1}$ ou bien 0 si `k` est une puissance de 2.

4. Écrivez une fonction `int int_array_sum(int array[], int array_size)` qui renvoie la somme des éléments d'un tableau, la tester sur les deux tableaux précédents
5. Écrivez une fonction `int int_array_min(int array[], int array_size)` qui renvoie le minimum des éléments d'un tableau, la tester sur les deux tableaux
6. Écrivez une fonction `int int_array_max(int array[], int array_size)` qui renvoie le maximum des éléments d'un tableau, la tester sur les deux tableaux précédents
7. Reprenez ces questions pour des tableaux contenant des `double`.

2.4 Présentation des automates cellulaires

Dans le reste de ce TP on va s'intéresser à certains automates cellulaires. Les automates cellulaires sont des sortes de machines, dont l'instance la plus célèbre est le **Jeu de la vie** inventé par *John Conway*.

Plus formellement, un automate cellulaire est constitué d'une **grille** dont les **cases** (ou **cellules**) peuvent être dans un nombre fini d'états. L'état dans lequel sont ces cases évolue en fonction de l'état courant et de l'état des cases voisines. L'ensemble des états de la grille à une étape donnée est appelé une **configuration**. La plupart du temps la grille est un quadrillage de dimension finie ou infinie et les cases voisines sont les cases qui bordent une case donnée.

Ici nous nous intéresserons à des automates cellulaires plus modeste pour lesquels la grille est en une dimension et d'une taille finie. On représentera bien entendu ces grilles par des tableaux. On peut illustrer ceci avec l'exemple d'automate cellulaire suivant :

— La grille est un tableau de 5 cases adjacentes, numérotées de 0 à 4 ;
— Chaque case de la grille peut être dans un des états suivants : {a, b, c, d}.
— Les cases voisine d'une case donnée sont les cases situées à sa droite et à sa gauche.
— L'état d'une case évolue de manière à prendre l'état majoritaire parmi les cases voisine, case courante comprise. C'est l'ordre alphabétique qui départage les égalités.

On peut voir ici un exemple d'évolution des états d'un tel automate à partir d'un état initial :

t	case 0	case 1	case 2	case 3	case 4
0	b	a	b	c	c
1	a	b	a	c	c
2	a	a	a	c	c
3	a	a	a	c	c

Pour les cases du bord, on a utilisé le seul voisin disponible dans le décompte de l'état majoritaire.

2.5 Implémentation d'un premier automate cellulaire

Pour implémenter un premier automate cellulaire, nous allons nous fixer des règles et états plus simple encore :

— La grille est un tableau de 64 cases.
— Chaque case peut prendre l'état 0 ou 1.
— L'évolution des états est donnée par le tableau suivant :

voisine de gauche	case courante	voisine de droite	nouvel état
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	0
1	0	0	1
1	0	1	0
1	1	0	0
1	1	1	0

Ou plus graphiquement, avec en noire les états 1 et en blanc les états 0.

état courant (voisines comprises)	□□□	□□■	□■□	□■■	■□□	■□■	■■□	■■■
nouvel état de la case courante	□	■	□	□	■	□	□	□

1. Déroulez sur un papier 3 étapes de l'évolution de cet automate sur un papier pour une grille réduite à 5 cases. Quel problème se pose ? Comment proposeriez-vous de le régler ?
2. Dans la suite de ce TP, on fixera les états des cases voisines à gauche de 0 et à droite de la case la plus à droite (63) comme constants à 0.
3. (Bonus) Imaginez une autre manière de procéder que de fixer un état pour les cases adjacentes au bord. Pendant ce TP implémentez ce qui est nécessaire pour procéder de cette manière.
4. Écrivez une fonction `int automata_rule(int current_cell, int left_cell, int right_cell)` qui prend en entrée l'état d'une case et de ses deux voisines et renvoie la sortie selon la règle détaillée plus haut.
5. Écrivez une fonction `void update_state(int grid_now[], int updated_grid[], int grid_size)` qui calcule et stocke un nouvel état de grille dans le tableau `updated_grid` passé en argument à partir d'un état courant contenu dans `grid_now`. Cette fonction sera capable de calculer un nouvel état pour une taille de grille arbitraire mais passée en argument sous le nom `grid_size`.
6. (Bonus) Écrivez une fonction qui met à jour l'état d'une grille **en place**, c'est-à-dire sans avoir besoin d'un tableau supplémentaire pour stocker le nouvel état en modifiant directement le tableau contenant l'état courant. Attention à bien vérifier que cette fonction a un comportement identique à la fonction précédente.
7. Écrivez une fonction `void run_automata(int init_grid[], int previous_step_grid[], int next_step_grid[], int grid_size, int nb_steps)` qui affiche dans le terminal les états successifs de l'automate à partir d'une grille initiale `init_grid` de taille `grid_size` pendant `nb_steps` étapes en se servant des tableaux `previous_step_grid` et `next_step_grid` pour stocker les résultats du calcul d'un changement d'état.
8. Essayez cette fonction sur une grille initiale de taille 64 dont toutes les cases sont dans l'état 0 sauf la 31e pour un nombre d'étapes de votre choix.
9. Pour obtenir un meilleur affichage, écrivez une fonction `void print_grid_state(int grid[], int grid_size)` qui affiche des `*` pour les cases dans l'état `1` et des espaces pour les cases dans l'état `0`.
10. Appliquez cette fonction à partir d'autres états initiaux plus diversifiés, qu'observez-vous ?

2.6 Un affichage encore amélioré

Le terminal ne permet pas d'afficher correctement des grilles dont la taille est supérieure à 80 cases. Pour résoudre ce problème nous allons créer une image dans le format basique `.pbm` pour *portable bitmap*. Un fichier `StatePrint.c` est fourni pour ce TP dans le répertoire partagé. Ce fichier contient les fonctions

1. `FILE* init_picture(char *name, int grid_size, int nb_steps)` qui permet de créer un fichier image aux dimensions de la grille et du nombre d'étapes à afficher et renvoie ce fichier. L'argument `name` correspond au nom du fichier et devra terminer par `.pbm`. Ce fichier sera créé à l'endroit où est exécuté le programme C. Le type de ce fichier est `FILE*`. Nous n'aborderons pas dans le TP l'écriture et la lecture de fichiers en C.
2. `print_state_to_file(FILE *f, int grid_state[], int grid_size)` qui permet d'écrire dans le fichier image les informations nécessaires à l'affichage de l'état de la grille contenu dans `grid_state`.
3. `void save_picture(FILE *f)` qui permet de réaliser les opérations nécessaires à la sauvegarde du fichier.

Pour intégrer utiliser le code d'un fichier externe, vous pouvez ajouter la ligne `#include "<nom_du_fichier>.c"` en haut de votre code source si ce fichier est placé dans le même répertoire que le code source dans lequel il est inclus.

1. Écrivez une fonction `run_automata_with_picture(int init_grid[], int previous_step_grid[], int next_step_grid[], int grid_size, int nb_steps)` qui crée une image des états successifs de l'automate plutôt que de les afficher dans le terminal.
2. Générez une image pour une grille de taille 256 et 1000 états successifs.

2.7 Pour aller plus loin (Bonus)

— Remplacez la règle décrite précédemment par la règle suivante et observez les motifs obtenus :

état courant (voisines comprises)	□□□	□□■	□■□	□■■	■□□	■□■	■■□	■■■
nouvel état de la case courante	■	□	□	■	□	□	■	□

— À partir de combien d'étape a-t-on vu tout ce qu'il y avait à voir pour un état initial donné ? Comment pourrait-on détecter cet état de fait ou calculer ce nombre d'étapes ?

2.8 Génération de toutes les règles d'évolution pour les automates de ce type (Bonus)

On a pour le moment vu deux règles différente d'évolution de l'état des cases. Dans cette partie nous allons tenter d'énumérer toutes les règles possibles et d'implémenter une fonction qui applique une règle étant donné son numéro.

Pour énumérer ces règles, la première étape consiste à s'assurer que celles-ci sont en nombre fini.

1. Comment pourrait-on formaliser une règle sous la forme d'une fonction ? Précisez les ensembles de départ et d'arrivée de cette fonction.
2. Calculez combien il existe de règles différentes d'après cette formulation.
3. Pour caractériser une règle, on choisit de fixer un ordre sur les différentes configurations possible de voisinage pour une case. On peut remarquer qu'on a déjà procédé de la sorte dans la description des règles précédentes. Proposez une numérotation pour les différentes configurations possible, en n'oubliant pas que l'état de chacune des cases est un 0 ou un 1.

4. Une fois une numérotation fixée sur les différentes configurations de voisinage, comment décrire de manière minimaliste une règle à l'aide de 0 et de 1.
5. Déduisez de cette représentation une numérotation des règles.
6. Écrivez une fonction qui étant donné un tableau de 0 et de 1 calcule l'entier dont c'est la représentation en base 2. Le nombre dans la case 0 correspondra au coefficient associé à la puissance de 2 la plus petite.
7. Écrivez la fonction réciproque qui stocke dans un tableau, dont la taille est soigneusement choisie, la décomposition en base 2 d'un entier passé en entrée.
8. Écrivez une fonction qui étant donné l'état d'une case et de ses deux voisines applique la règle dont le numéro est donné comme argument supplémentaire et calcule le nouvel état de la case courante. On pourra transformer cet état du voisinage en un entier, le numéro de ce voisinage et obtenir l'image de ce voisinage à partir de la décomposition du numéro de la règle.
9. Écrivez une variante des fonctions `update_grid` et `run` pour appliquer une règle dont le numéro est donné en argument.
10. Observez les motifs produits par plusieurs règles de votre choix.