### Test de l'environnement de développement

Pour tester votre installation, vous allez commencer par créer un petit programme "Hello World". Pour cela, il faut :

1. Créer un répertoire (bonne pratique : un répertoire = un programme) sur votre disque C (p.ex. C :/Programmation_Avancee/TP/Test/). ATTENTION : ce répertoire ne doit pas être dans "Documents".
2. Ouvrir ce répertoire dans VSCode en allant dans le menu "File" et en cliquant sur "Open Folder" (raccourci ctrl+k ctrl+o)
3. Valider votre confiance ("trust") dans les données de ce répertoire
4. Créer un nouveau fichier en allant sur le menu "File", puis "New File" (raccourci ctrl+n)
5. Enregistrer ce fichier dans votre répertoire en cliquant sur le menu "File" > "Save" (raccourci ctrl+s), puis saisir le nom du fichier (main.c) et valider (le fichier doit maintenant apparaître dans l'arborescence de votre répertoire à gauche dans VSCode)
6. Saisir dans ce fichier main.c le code présenté dans le transparent 9 du cours. A ce stade, on doit avoir la coloration syntaxique du code.

VSCode fait de la coloration syntaxique et affiche des conseils relatifs au langage C, mais il ne permet pas de compiler le code (VSCode est à la base un éditeur de texte et non environnement de développement intégré, ou IDE). Pour cela, il faut utiliser un compilateur externe tels que gcc ou clang. Ces compilateurs ne sont pas installés par défaut sous Windows mais ils le sont dans Linux. Nous allons donc utiliser le terminal Linux Ubuntu installé dans WSL, compiler le code dans ce terminal Linux et l'exécuter. Pour cela, il suffit de :

1. Ouvrir un terminal en allant dans le menu "Terminal", et en sélectionnant "New Terminal"
2. Remplacer le terminal Powershell (par défaut) par un terminal Ubuntu en cliquant sur la fleche descendante à côté du "+" en bas à droite de la fenêtre (à côté du mot "powershell"), puis en sélectionnant "Ubuntu 20.04 (WSL)"
3. Taper la commande "gcc -o main main.c" dans ce terminal pour compiler le code
4. Exécuter une première fois le programme (fichier main généré à l'étape précédente) en tapant dans le terminal "./main"
5. Exécuter une seconde fois le programme en tapant dans le terminal "./main hello world"

ATTENTION : le répertoire dans lequel vous avez créé ce projet est temporaire. Il sera supprimé automatiquement à la fermeture de votre session. Il est donc TRES important de penser à le sauvegarder sur votre disque personnel (vous pouvez utiliser pour cela l'explorateur Windows).

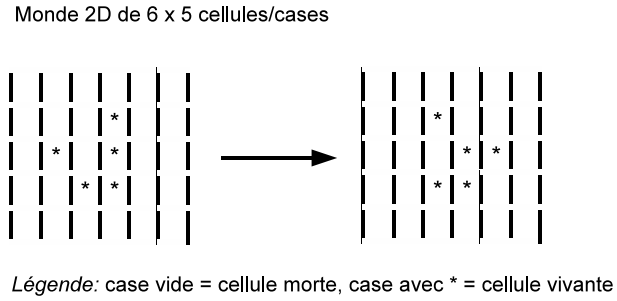
## Description du problème

Le Jeu de la Vie, inventé par John Conway en 1970, est un exemple de jeu, de modèle (un automate cellulaire), simulant l'évolution d'une colonie de cellules en fonction de règles simples. Ce jeu sans joueur est constitué d'un monde en deux dimensions divisé en "cellules". Chaque cellule est soit "vivante", soit "morte" à une "génération" donnée. Le jeu est composé d'un ensemble de règles décrivant comment les cellules évoluent d'une génération à une autre. Ces règles calculent l'état d'une cellule à la génération suivante en fonction des états des ses cellules voisines dans la génération "courante". Dans un monde en 2D, le voisinage d'une cellule est l'ensemble des 8 cellules adjacentes (verticalement, horizontalement et en diagonal). Les règles de Conway peuvent être résumées de la manière suivante :

— si une cellule a strictement moins de 2 voisines, elle meure (isolement),
— si une cellule a strictement plus de 3 voisines, elle meure (surpopulation),
— si une cellule a 2 ou 3 voisines, elle vit (se régénère),
— si une case vide (i.e. une cellule morte) a exactement 3 voisines, une nouvelle cellule apparaît (reproduction).

Dans ce TP, nous supposerons que le monde 2D étudié est fini. Les cellules en dehors des limites du monde sont supposées mortes.

Exemple d'évolution :




Vous pouvez trouver d'autres informations sur ce jeu sur Wikipedia :
`http://fr.wikipedia.org/wiki/Jeu_de_la_vie`

# Partie 1 : Implémenter l'évolution

Le "monde" à une génération donnée peut être représenté par un tableau à deux dimensions (i.e. une matrice). Le type composé suivant peut par exemple être utilisé pour représenter ce type d'information :

```
typedef struct {
int lin, col;
char cellules[50][50];
} Monde ;
```

Le champ `cellules` est un tableau à deux dimensions représentant toutes les cellules du monde 2D. Les champs `lin` et `col` représentent le nombre de lignes et de colonnes du tableau. L'état d'une cellule située à la position (l,c) est indiqué dans la case `cellules[l][c]`. Une valeur `" "` dans cette case signifie que la case est libre, autrement dit que la cellule est morte. Une valeur `"*"` signifie que la cellule est vivante.

Vous écrirez tout d'abord une première version du programme basée sur des allocations statiques de la mémoire. Le monde aura donc une taille maximale, fixée par le développeur dans le programme principal (50x50 dans l'exemple précédent). Une fois que le simulateur fonctionnera, vous améliorerez votre programme de façon à mieux gérer la mémoire, i.e. en faisant des allocations dynamiques (en fonction des choix des utilisateurs).

Les étapes du développement du simulateur sont détaillées dans les sections ci-dessous.

### Initialiser le monde virtuel

Tout d'abord, vous devrez écrire une procédure `void initEmpty( monde * m )` qui initialisera à vide le monde (toutes les cellules sont mortes). Vous écrirez aussi une procédure d'affichage `void print(monde * m)` afin d'afficher à l'écran l'"état" du monde `m`. Vous écrirez un programme principal

pour tester ces procédures et l'éxécuterez. Attention, l'allocation de la mémoire devra être faite dans le programme principal.

### Accéder/modifier une case

Ensuite, vous écrirez deux fonctions `char getCell(monde * m, int l, int c)` et `void setCell( monde * m, int l, int c, char state)`. La première fonction retourne l'état de la cellule située à la position (l,c) dans le monde. Elle retournera donc "*" ou " ". La deuxième fonction permet de modifier l'état de la cellule située à la position (l,c) dans le monde. La cellule modifiée prendra alors pour valeur `state` (qui devra être "*" ou " "). Vous modifierez votre programme principal afin de tester les procédures créées.

### Coder les règles d'évolution

Dans un second temps, vous devrez écrire la procédure `void evolve( monde * avant, monde * apres)` qui permet de simuler le passage d'une génération à une autre. Cette procédure appliquera les règles de Conway sur chaque cellule du monde `avant` et enregistrera le résultat dans le monde `apres`. Dans votre programme principal, testez votre procédure avec en entrée des mondes dans des états différents. Vous reproduirez notamment l'exemple d'évolution donné précédemment.

### Finaliser le simulateur

Ensuite, vous écrirez une procédure `void runSimu( monde * avant, monde * apres, int nbGen)` qui permettra de lancer la simulation sur `nbGen` générations. Vous modifierez votre programme principal pour lancer une simulation complète. Dans cet objectif, vous demanderez à l'utilisateur de saisir jusqu'à combien de générations la simulation doit se projeter.

## Partie 2 : Le monde dans un fichier et le monde aléatoire

Dans la première partie du TP, l'état initial du monde devait être codé "en dur" dans le programme principal ou saisi (case après case) par l'utilisateur à chaque exécution. De plus, l'état final du monde était uniquement affiché dans le terminal. Il n'était pas conservé après la fin du programme. Dans cette partie, nous allons ajouter des procédures afin de lire/écrire ces informations dans un fichier, ainsi que des procédures permettant d'initialiser aléatoirement l'état d'un monde.

### Le monde dans un fichier

Vous écrirez deux méthodes `int initialize_world_from_file(char * filename, monde * m)` et `int save_world_to_file(char * filename, monde * m)`.

La première lit les informations contenues dans le fichier `filename` et les enregistre dans la structure de données `m`. Chaque ligne du fichier correspond à une succession de "*" et de " ". Le i-ème caractère de la j-ème ligne du fichier correspond à l'état de la cellule située à la position (i,j) du monde. Si la ligne ne contient pas assez de caractères ou si le fichier ne contient pas assez de lignes, les cellules non spécifiées sont supposées mortes. Vous mettrez en place un système de gestions des erreurs (p.ex. si le fichier n'existe pas), et retournerez -1 si la lecture ne s'est pas faite correctement.

La deuxième méthode enregistre le monde `m` dans le fichier `filename` en suivant le format décrit précédemment. Cette fonction retournera -1 si la sauvegarde n'a pu être menée à bien.

Le programme principal sera modifié afin de permettre à l'utilisateur de saisir le nom du fichier à lire. Avant le début de la simulation, l'utilisateur pourra également choisir entre deux options d'un menu textuel :

— "Affichage étape après étape" : cette option indique que l'utilisateur souhaite voir à l'écran l'évolution des cellules génération après génération.

— "Enregistrement dans un fichier" : cette option indique que l'utilisateur souhaite enregistrer le résultat final dans un fichier dont il précisera le nom.

### Le monde aléatoire

Dans cette sous-section, l'objectif sera d'implémenter une méthode `void initAleatLive(monde * m, int p)` qui initialisera à "vivant" `p` pour cent des cellules du monde `m`. La position des cellules vivantes sera tirée aléatoirement en fonction de la taille du monde.

Vous modifierez votre menu textuel afin de permettre à l'utilisateur d'utiliser cette fonctionnalité (et vous testerez, bien entendu).

## Indications concernant l'implémentation du programme et les tests

Chaque fonction/procédure devra être commentée et documentée dans le code source (texte décrivant la fonction, ses entrées et ses sorties). De plus, chaque fonction/procédure devra être aussi testée au fur et à mesure sur plusieurs exemples de données (différents). Pour cela, vous devrez écrire une fonction de test pour chaque fonction/procédure implémentée. Les captures d'écran des tests réalisés devront être intégrées au rapport. **Toute fonction non testée sera considérée comme ne fonctionnant pas.**

A titre d'exemple, supposons que nous ayons écrit une fonction permettant de concaténer deux chaînes de caractères : `char * concatene(char * a, char *b){ ... }`. Il faudra donc écrire une fonction de test (et l'exécuter dans un main) du type :

```
int concateneTest() {
        char * a = "je suis en ";
        char * b = "TP de programmation avancee";
        char * resultatAttendu = "je suis en TP de programmation avancee";

        char * resultatObtenu = concatene(a, b);

        if ( strcmp(resultatAttendu, resultatObtenu) == 0)
        {
             return 1;
        }
        else
        {
             return 0;
        }
}
```

## Quelques commandes C utiles

### Créer des nombres pseudo-aléatoires

Pour créer un nombre pseudo-aléatoire (entre 0 et un entier $x$, $x$ non inclus), il faut :

1. importer la librairie contenant les primitives pour les nombres aléatoires : *#include <stdlib.h>* et *#include <time.h>*

2. Initialiser le générateur :
   *srand( time(NULL) ) ;*
3. exécuter le générateur de nombres entiers pseudo-aléatoires et stocker la valeur générée dans une variable de type entier :
   *int aleat = rand()% x ;*

### Manipuler des fichiers

Les étapes à suivre pour lire/écrire dans un fichier sont :

1. importer les librairies stdlib et stdio :
   `#include <stdlib.h>` et `#include <stdio.h>`
2. ouvrir le fichier et renvoyer un pointeur vers celui-ci :
   `FILE* fopen(const char* nomDuFichier, const char* modeOuverture);`
3. pour chaque ligne du fichier, au choix
   — lire la ligne courante :
   `char* fgets(char* chaine, int nbreDeCaracteresALire, FILE* pointeurSurFichier);`
   (ou utiliser `fscanf` pour un fichier formatté)
   — écrire une ligne (après la ligne courante) :
   `char* fputs(const char* chaine, FILE* pointeurSurFichier);`
   (ou utiliser `fprintf` pour un fichier formatté)
4. fermer le fichier :
   `int fclose(FILE* pteurFichier);`

Exemple : lire le contenu d'un fichier ("test.txt")

```
FILE * fichier = NULL ;
char ligne[500] ="" ;
char * testLigneVide = NULL ;
int cpt = 0 ;
fichier = fopen("test.txt","r") ;
if( fichier != NULL ) {
  testLigneVide = fgets( ligne, 500, fichier ) ;
  while( testLigneVide != NULL )
  {
    cpt++ ;
    printf("Ligne %d : %s \n", cpt, ligne ) ;
    testLigneVide = fgets( ligne, 500, fichier ) ;
  }
  fclose( fichier ) ;
}
```

### Gérer des tableaux multidimensionnels

— Déclaration d'un tableau à N dimensions avec allocation de la mémoire :
  — *type* `NomTableau [ tailleD1 ][ tailleD2 ][ tailleD3 ] .. [ tailleDN ]`
  — ex. : `int myTab [3][4];` (permet de créer une matrice avec 3 lignes et 4 colonnes)

— Accès à une case du tableau

  - `NomTableau [`$i_1$`] [`$i_2$`] [`$i_3$`] ... [`$i_n$`]`
  - ex. : `int val = myTab [0][2] ;` (récupère la valeur située dans la case à l'intersection de la première ligne et troisième colonne)

- Affecter une valeur à une case du tableau
  - `NomTableau [`$i_1$`] [`$i_2$`] [`$i_3$`] ... [`$i_n$`] = val`
  - ex. : `myTab [0][2] = val;` (initialise à `val` la case à l'intersection de la première ligne et troisième colonne)

- Les tableaux à deux dimensions dynamiques
  - Déclarer :
    - `type ** NomTableau`
    - ex. `int ** matrice;` (déclare une matrice d'entiers)
  - Allouer la mémoire :
    - Sachant qu'un tableau à deux dimensions est un tableau de tableaux, il faut d'abord allouer le tableau principal, puis faire une boucle pour allouer les tableaux associés à chaque case.
    - ex. permettant d'allouer dynamiquement une matrice avec 2 lignes et 3 colonnes

```
int taille1 = 2 , taille2 = 3;
int ** matrix;
matrix = malloc(taille1 * sizeof( int* ));
for(int i=0 ; i < taille1 ; i++)
    matrix[i] = malloc(taille2 * sizeof( int ) );
```

  - Libérer la mémoire :
    - Il faut faire une boucle pour libérer les tableaux contenus dans chaque case du tableau principal, puis libérer le tableau principal.
    - ex. permettant de libérer la mémoire d'une matrice avec 2 lignes et 3 colonnes

```
for(int i=0 ; i < taille1 ; i++)
    free( matrix[i] );
 free( matrix );
```