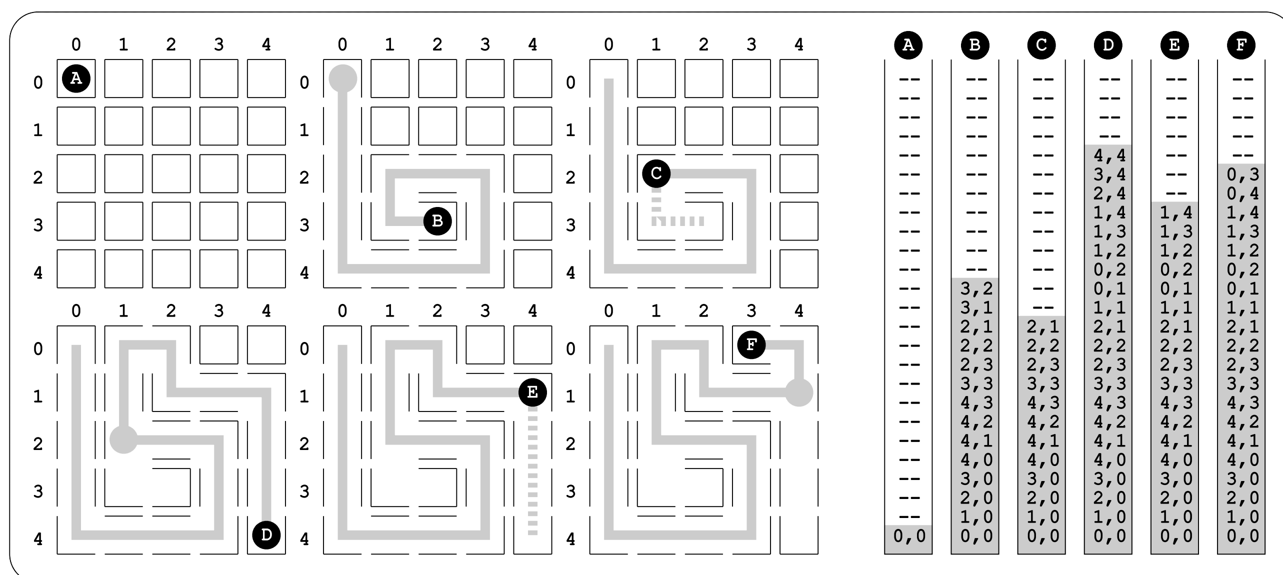
TD/TP de Programmation – Planche 5
Génération aléatoire de labyrinthes

On souhaite construire des labyrinthes aléatoirement. Un labyrinthe est vu comme un ensemble de cellules disposées dans une grille (une cellule par case). Chaque cellule peut posséder jusqu'à 4 cloisons (une par direction Nord, Est, Sud, Ouest) la séparant des cellules voisines. Les murs entre les cellules sont toujours constitués de doubles cloisons : par exemple, si deux cellules occupent deux cases contiguës verticalement, et que la cellule Nord possède une cloison Sud, alors la cellule Sud possède toujours la cloison jumelle Nord.

Le labyrinthe est initialement constitué de cellules toutes isolées (cloisonnées dans les 4 directions). Le labyrinthe est construit grâce à la marche aléatoire d'un lutin qui abat les murs au fur et à mesure de sa progression, jusqu'à ce qu'aucune cellule ne soit plus isolée.



Au départ, le lutin est enfermé dans une cellule choisie aléatoirement (schéma A). À chaque étape de sa progression, il choisit aléatoirement une cellule isolée et voisine de sa cellule actuelle. Il abat les deux cloisons séparant ces deux cellules, et il se déplace dans la cellule voisine. Chaque fois qu'il rentre dans une pièce, le lutin y dépose un caillou à la manière du Petit Poucet, de telle sorte que les cailloux forment un chemin linéaire de sa cellule actuelle à sa cellule de départ. Lorsque le lutin ne peut choisir aucune cellule voisine (parce qu'elles sont toutes non-isolées, schéma B), il doit revenir sur son chemin : il reprend le caillou déposé dans la cellule actuelle et revient dans la cellule précédente (backtrack). Il devra éventuellement backtracker plusieurs fois (schéma C) avant de pouvoir continuer à progresser (schéma D).

L'algorithme se termine lorsque le lutin bracktracke jusqu'à son point de départ (toutes les cases sont alors non-isolées, schéma F). Le labyrinthe a alors une forme arborescente (tous les chemins finissent en cul-de-sac, on ne peut pas donc pas "tourner en rond"). Ceci peut se corriger en abattant aléatoirement un certain nombre de murs supplémentaires après la marche du lutin.

Le système des cailloux s'implémente avec une structure de pile. La pile contient toutes les coordonnées des cellules contenant un caillou, dans l'ordre du chemin, de la cellule de départ (en bas de pile) à la cellule actuelle (en haut de pile). Visiter une cellule consiste à empiler ses coordonnées (opération *push*), c'est-à-dire poser un caillou. Backtracker consiste dépiler une coordonnée (opération *pop*), c'est-à-dire reprendre le caillou posé dans la cellule actuelle. on connait toujours la cellule actuelle en consultant le haut de pile (opération *peek*). L'algorithme se termine lorsque l'on tente de backtracker depuis la cellule de départ, c'est à dire lorsque la pile est vide.

Structures de Données

`Dir` est un type énumératif permettant d'avoir des constantes symboliques pour les entiers 0,1,2,3 représentant le Nord, l'Est, le Sud et l'Ouest. Le type `Pos` permet de désigner la position d'une cellule dans la grille ou la dimension de la grille par un couple d'entiers (numéro de ligne, numéro de colonne). Le type `Cell` représente l'état d'une cellule par 4 booléens indiquant la présence ou l'absence de cloison dans les 4 directions. Le type `Maze` représente un labyrinthe par sa dimension et un tableau plat de cellules. Enfin le type `Stack` représente une pile de positions implémentée par tableau.

```
typedef enum Dir {
  DIR_NONE= -1,
  DIR_NORTH, DIR_EAST, DIR_SOUTH, DIR_WEST,
  NB_DIRS
} Dir;

typedef struct Pos {
  int row, col;
} Pos;

typedef struct Cell {
  bool walls [NB_DIRS];
} Cell;

typedef struct Maze {
  Pos dim;
  Cell * cells;
} Maze;

typedef struct Stack {
  Pos * elements;
  int nb_elements, capacity;
} Stack;
```

On réutilisera également le type `Coord` des TP précédents pour réprésenter les points du plan.

1 Directions : `enum Dir`

Implémenter les fonctions suivantes relatives aux directions : `Dir_Name()` qui retourne le nom d'une direction ; `Dir_Opposite()` qui retourne la direction opposée d'une direction ; `Dir_Random()` qui retourne une direction choisie aléatoirement.

```
char const * Dir_Name (Dir dir);

Dir Dir_Opposite (Dir dir);
Dir Dir_Random   (void);

void MainTest_Dir (void);
```

Écrire un petit programme de test pour ces fonctions : par exemple, tirer une trentaine de directions au hasard, et afficher leur nom ainsi que le nom de la direction opposée.

```
W -> E
W -> E
E -> W
N -> S
S -> N
E -> W
...
```

2 Positions et Dimensions : `struct Pos`

Implémenter les fonctions suivantes relatives aux positions : `Pos_Make()` qui fabrique et retourne une position ; `Pos_Neighbor()` qui retourne la position voisine d'une position dans une direction donnée ; `Pos_Random()` qui retourne une position aléatoire dans une grille de dimension donnée.

```
Pos Pos_Make     (int row, int col);
Pos Pos_Neighbor (Pos pos, Dir dir);
Pos Pos_Random   (Pos dim);

void MainTest_Pos (void);
```

Écrire un petit programme de test pour ces fonctions : par exemple, tirer une trentaine de positions et directions au hasard, et afficher à chaque fois la position, le nom de la direction, ainsi que la position voisine selon cette direction.

```
(0,6) -> N -> (-1,6)
(3,4) -> S -> (4,4)
(2,5) -> E -> (2,6)
(0,2) -> N -> (-1,2)
(1,5) -> N -> (0,5)
(3,1) -> W -> (3,0)
...
```

3 Cellules : `struct Cell`

Implémenter les fonctions suivantes relatives aux cellules : `Cell_Init()` qui initialise une cellule isolée par 4 murs ; `Cell_EraseWall()` qui casse une cloison de cellule selon une direction donnée ; `Cell_HasWall()` qui teste la présence d'une cloison de cellule selon une direction donnée ; `Cell_IsIsolated()` qui teste si une cellule est isolée ; `Cell_ToString()` qui retourne une représentation-texte de l'état d'une cellule (par exemple `"I:NESW"` si elle est isolée, et `"O:N-S-"` si elle est ouverte par l'Est et l'Ouest).

```
void   Cell_Init       (Cell * cell);
void   Cell_EraseWall  (Cell * cell, Dir dir);

bool   Cell_HasWall    (Cell const * cell, Dir dir);
bool   Cell_IsIsolated (Cell const * cell);
char * Cell_ToString   (Cell const * cell);

void MainTest_Cell (void);
```

Écrire un petit programme de test pour ces fonctions : par exemple, créer une cellule, puis tirer une dizaine de directions au hasard, casser la cloison selon cette direction, et afficher l'état des murs avant et après l'opération.

```
(I:NESW) - N = (O:-ESW)
(O:-ESW) - N = (O:-ESW)
(O:-ESW) - S = (O:-E-W)
(O:-E-W) - E = (O:---W)
(O:---W) - S = (O:---W)
(O:---W) - N = (O:---W)
(O:---W) - E = (O:---W)
(O:---W) - N = (O:---W)
(O:---W) - W = (O:----)
(O:----) - S = (O:----)
```

4 Labyrinthe : `struct Maze`

Implémenter les fonctions suivantes relatives au labyrinthe : `Maze_NbCells()` qui retourne le nombre de cellules d'un labyrinthe ; `Maze_FlatIndex()` qui retourne l'index plat correspondant à une position donnée dans la grille ; `Maze_CellAt()` qui retourne la cellule à une position donnée ; `Maze_Init()` qui initialise un labyrinthe de cellules isolées pour une dimension donnée ; `Maze_Clean()` qui libère les ressources mémoire allouées par un labyrinthe.

```
int Maze_NbCells   (Maze const * maze);
int Maze_FlatIndex (Maze const * maze, Pos pos);
Cell * Maze_CellAt (Maze const * maze, Pos pos);

void Maze_Init  (Maze * maze, Pos dim);
void Maze_Clean (Maze * maze);

void MainTest_Maze (void);
```

Écrire un petit programme de test pour ces fonctions : par exemple, initialiser un labyrinthe de dimension 3×5, afficher les cellules pour toutes les positions, et détruire le labyrinthe.

```
[(0,0) I:NESW] [(0,1) I:NESW] [(0,2) I:NESW]
[(1,0) I:NESW] [(1,1) I:NESW] [(1,2) I:NESW]
[(2,0) I:NESW] [(2,1) I:NESW] [(2,2) I:NESW]
[(3,0) I:NESW] [(3,1) I:NESW] [(3,2) I:NESW]
[(4,0) I:NESW] [(4,1) I:NESW] [(4,2) I:NESW]
```

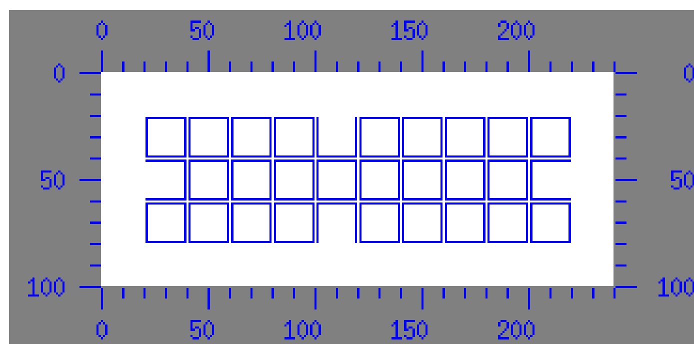
5 Affichage graphique

Écrire `Cell_Draw()` qui affiche dans une fenêtre une cellule carrée de `side-2` pixels de côté, aux coordonées `top_left` (coin supérieur-gauche). Écrire `Maze_CellCoord()` qui retourne les coordonées de la cellule d'un labyrinthe en position `Pos`. Écrire `Maze_Draw()` qui affiche dans une fenêtre un labyrinthe complet aux coordonées `top_left` (coin supérieur-gauche) avec des cellules de `cell_side` pixels de côté.

```
void Cell_Draw (Cell const * cell, bx_window win, Coord top_left, int side);
Coord Maze_CellCoord (Pos pos, Coord maze_coord, int cell_side);
void Maze_Draw (Maze const * maze, bx_window win, Coord top_left, int cell_side);

void MainTest_Drawing (void);
```

Tester ces fonctions : initialiser un labyrinthe, casser quelques cloisons, et afficher.

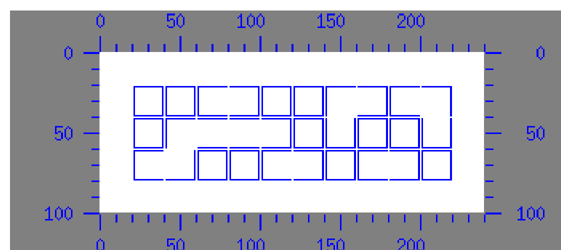


6 Destruction aléatoire de murs

Implémenter : `Maze_RowIsValid()`, `Maze_ColIsValid()`, `Maze_PosIsValid()` qui testent respectivement si une ligne, une colonne, et une position se situent dans les limites de la grille ; `Maze_RandomPos()` qui tire une position aléatoire à l'intérieur de la grille ; `Maze_EraseCellWall()` qui abat la double cloison séparant la cellule en une position donnée de sa voisine selon une direction donnée ; `Maze_EraseRandomWalls()` qui abat aléatoirement des doubles cloisons (en tirant au hasard une cellule et une direction `nb_times` fois). Tester les versions itérative et récursive terminale de cette dernière fonction : initialiser un laryrinthe, abattre aléatoirement des murs, et afficher.

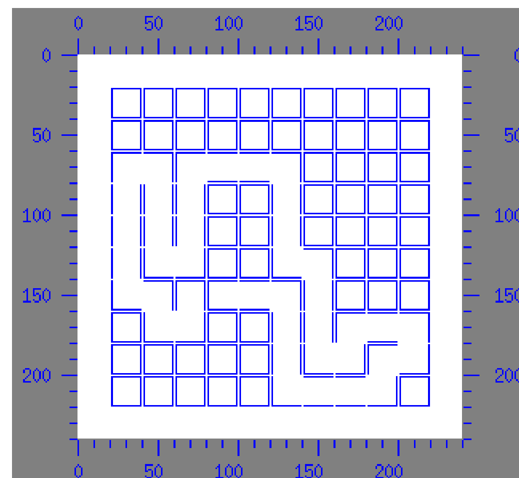
```
bool Maze_RowIsValid (Maze const * maze, int row);
bool Maze_ColIsValid (Maze const * maze, int col);
bool Maze_PosIsValid (Maze const * maze, Pos pos);
Pos  Maze_RandomPos      (Maze const * maze);
void Maze_EraseCellWall (Maze * maze, Pos pos, Dir dir);

void Maze_EraseRandomWalls_Iter (Maze * maze, int nb_times);
void Maze_EraseRandomWalls_Rec  (Maze * maze, int nb_times);
void MainTest_EraseRandomWalls  (void);
```



7 Marche aléatoire

Implémenter : `Maze_CountIsolatedNeighbors()` qui compte le nombre de cellules isolées voisines d'une cellule donnée ; `Maze_IsolatedDir()` qui retourne la `n`$^{\text{ième}}$ voisine isolée d'une cellule ; `Maze_RandomIsolatedDir()` qui retourne aléatoirement une voisine isolée parmi `n` voisines isolées ; `Maze_DoRandomWalk()` qui creuse un couloir dans un labyrinthe par une marche aléatoire. La marche se termine lorsqu'on recontre un cul-de-sac, c'est-à-dire lorsque l'on atteint une cellule sans voisine isolée. La progression de la marche est affichée dans une fenêtre avec une attente de `delay` millisecondes entre chaque étape. Tester les versions itérative et récursive terminale de cette dernière fonction.




```
int Maze_CountIsolatedNeighbors (Maze const * maze, Pos pos);
Dir Maze_IsolatedDir            (Maze const * maze, Pos pos, int n);
Dir Maze_RandomIsolatedDir      (Maze const * maze, Pos pos, int n);

void Maze_DoRandomWalk_Iter (Maze * maze, Pos start, bx_window win,
                             Coord maze_coord, int cell_side, int delay);

void Maze_DoRandomWalk_Rec  (Maze * maze, Pos start, bx_window win,
                             Coord maze_coord, int cell_side, int delay);

void MainTest_DoRandomWalk  (void);
```

8 Pile de positions : struct Stack

Implémenter les fonctions relatives à une pile de positions : `Stack_Init()` qui initialise une pile vide d'une capacité donnée ; `Stack_Clean()` qui libère les ressources d'une pile après utilisation ; `Stack_IsEmpty()` qui teste si une pile est vide ; `Stack_IsFull()` qui teste si une pile a atteint sa capacité ; `Stack_Push()` qui empile une position ; `Stack_Pop()` qui dépile une position ; `Stack_Peek()` qui consulte la position en haut de pile sans la dépiler ; `Stack_Print()` qui affiche une pile.

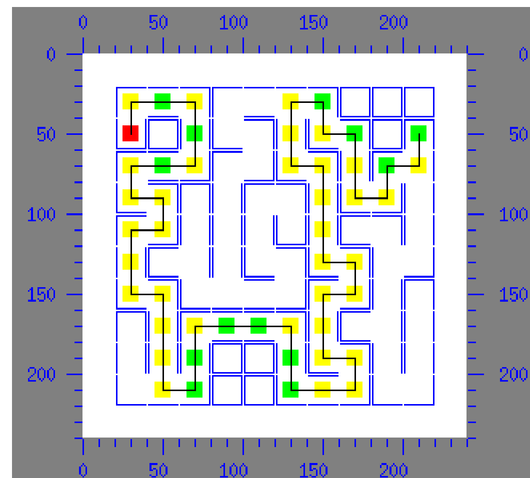
```
void Stack_Init    (Stack * s, int capacity);
void Stack_Clean   (Stack * s);
bool Stack_IsEmpty (Stack const * s)
bool Stack_IsFull  (Stack const * s);
void Stack_Push    (Stack * s, Pos pushed);
Pos  Stack_Pop     (Stack * s);
Pos  Stack_Peek    (Stack const * s);
void Stack_Print   (Stack const * s, FILE * file);
void MainTest_Stack (void);
```

Écrire un petit programme de test pour ces fonctions : créer une pile de capacité 4, et balayer la chaîne `"++++--++----"` où `'+'` représente l'empilement d'une position aléatoire, et `'-'` représente un dépilement ; effectuer l'opération associée au caractère balayé et afficher la pile après chaque étape.

```
[ ] FILL-RATIO 0/4 (EMPTY)
[ (6,3) ]  FILL-RATIO 1/4 AFTER +(6,3)
[ (6,3) (15,17) ] FILL-RATIO 2/4 AFTER +(15,17)
[ (6,3) (15,17) (15,13) ] FILL-RATIO 3/4 AFTER +(15,13)
[ (6,3) (15,17) (15,13) (12,6) ] FILL-RATIO 4/4 (FULL) AFTER +(12,6)
[ (6,3) (15,17) (15,13) ] FILL-RATIO 3/4 AFTER -(12,6)
...
```

9 Génération aléatoire

Écrire `Maze_DoRandomGen()`, une modification de `Maze_DoRandomWalk()` qui génère aléatoirement un labyrinthe complet. La version itérative utilise une pile de positions pour mémoriser le couloir creusé. Lorsqu'on tombe sur un cul-de-sac, plutôt que de s'arrêter, on revient en arrière (*backtrack*) en dépilant des positions jusqu'à trouver une cellule possédant des voisines isolées et continuer le couloir depuis ce point d'embranchement. La version récursive backtracke naturellement lors des remontées des appels récursifs. Implémenter également `Maze_DrawPath()` afin d'afficher à chaque étape le couloir représenté par la pile de positions dans la version itérative.




```
void Maze_DrawPath (Maze const * maze, Stack const * pending,
                    bx_window win, Coord maze_coord, int cell_side);

void Maze_DoRandomGen_Iter (Maze * maze, Pos start, bx_window win,
                            Coord maze_coord, int cell_side, int delay);

void Maze_DoRandomGen_Rec (Maze * maze, Pos current, bx_window win,
                           Coord maze_coord, int cell_side, int delay);

void MainTest_DoRandomGen (void);
```