# TD/TP de Programmation – Correction de la Planche 5
# Génération aléatoire de labyrinthes

## 1 Directions : `enum Dir`

```
int // il y un un biais si nb_wanted_values ne divise pas 1+RAND_MAX
Int_Random (int nb_wanted_values) {
  return rand () % nb_wanted_values;
}
```

```
int // solution avec retirage, suppose que UINT_MAX > RAND_MAX
Int_Random (int nb_wanted_values)
{
  unsigned nb_total_values    = 1U + (unsigned) RAND_MAX;
  unsigned nb_values_per_block= nb_total_values / nb_wanted_values;
  unsigned nb_rejected_values = nb_total_values % nb_wanted_values;
  unsigned nb_accepted_values = nb_total_values - nb_rejected_values;

  unsigned value, block_num;
  do { value= rand (); } while (value >= nb_accepted_values);
  block_num= value / nb_values_per_block;
  return block_num;
}
```

```
Dir
Dir_Random (void) {
  return Int_Random (NB_DIRS);
}
```

```
Dir
Dir_Opposite (Dir dir) {
  return ( dir + NB_DIRS/2 ) % NB_DIRS;
}
```

```
char const *
Dir_Name (Dir dir)
{
  switch (dir) {
  case DIR_NORTH: return "N";
  case DIR_EAST : return "E";
  case DIR_SOUTH: return "S";
  case DIR_WEST : return "W";
  default       : return "?";
  }
}
```

```
void
MainTest_Dir (void) {
  srand (time (NULL));
  int nb_draws= 30;
  for (int k= 0; k < nb_draws; k++) {
    Dir dir= Dir_Random ();
    Dir opp= Dir_Opposite (dir);
    printf ("%s␣->␣%s\n", Dir_Name (dir), Dir_Name (opp));
  }
}
```

## 2 Position et dimensions : `struct Pos`

```
Pos // before C99
Pos_Make (int row, int col)
{
  Pos pos;
  pos.row= row;
  pos.col= col;
  return pos;
}
```

```
Pos // using compound-literal since C99
Pos_Make (int row, int col)
{
  return (Pos) { .row= row, .col= col };
}
```

```
Pos
Pos_Invalid (void)
{
  return Pos_Make (-1, -1);
}
```

```
Pos
Pos_Neighbor (Pos pos, Dir dir)
{
  switch (dir){
  case DIR_NORTH: return Pos_Make (pos.row-1, pos.col  );
  case DIR_EAST : return Pos_Make (pos.row  , pos.col+1);
  case DIR_SOUTH: return Pos_Make (pos.row+1, pos.col  );
  case DIR_WEST : return Pos_Make (pos.row  , pos.col-1);
  default       : return Pos_Invalid ();
  }
}
```

```
Pos
Pos_Random (Pos dim)
{
  return Pos_Make (Int_Random (dim.row),
                   Int_Random (dim.col));
}
```

```
void
MainTest_Pos (void)
{
  srand (time (NULL));
  int nb_draws= 30;
  Pos dim= Pos_Make (5, 7);

  for (int k= 0; k < nb_draws; k++) {
    Pos pos= Pos_Random (dim);
    Dir dir= Dir_Random ();
    Pos neighbor= Pos_Neighbor (pos, dir);
    printf ("(%d,%d) -> %s -> (%d,%d)\n",
            pos.row, pos.col,
            Dir_Name (dir),
            neighbor.row, neighbor.col);
  }
}
```

# 3 Cellules : `struct Cell`

```
void
Cell_Init (Cell * cell)
{
  for (Dir dir= 0; dir < NB_DIRS; dir++) {
    cell->walls [dir]= true;
  }
}
```

```
bool
Cell_HasWall (Cell const * cell, Dir dir)
{
  return cell->walls [dir];
}
```

```
bool
Cell_IsIsolated (Cell const * cell)
{
  for (Dir dir= 0; dir < NB_DIRS; dir++) {
    if (! Cell_HasWall (cell, dir)) return false;
  }
  return true;
}
```

```
void
Cell_EraseWall (Cell * cell, Dir dir)
{
  cell->walls [dir]= 0;
}
```

```
char *
Cell_ToString (Cell const * cell)
{
  static char result [1024];
  char const * status= Cell_IsIsolated (cell) ? "I" : "O";
  int start= 0;
  int length= sprintf (result+start, "%s:", status);
  for (Dir dir= 0; dir < NB_DIRS; dir++) {
    char const * dir_name= Cell_HasWall (cell, dir) ? Dir_Name (dir) : "-";
    length= sprintf (result+start, "%s", dir_name);
    start+= length;
  }
  return result;
}
```

```
void
MainTest_Cell (void)
{
  srand (time (NULL));
  Cell cell;
  Cell_Init (& cell);

  int nb_draws= 10;
  for (int k= 0; k < nb_draws; k++) {
    Dir dir= Dir_Random ();
    printf ("(%s) - %s", Cell_ToString (& cell), Dir_Name (dir));
    Cell_EraseWall (& cell, dir);
    printf ("= (%s)\n", Cell_ToString (& cell));
  }
}
```

# 4 Labyrinthe : `struct Maze`

```
int
Maze_NbCells (Maze const * maze)
{
  return maze->dim.col * maze->dim.row;
}
```

```
int
Maze_FlatIndex (Maze const * maze, Pos pos)
{
  return maze->dim.col * pos.row + pos.col;
}
```

```
Cell *
Maze_CellAt (Maze const * maze, Pos pos)
{
  int flat= Maze_FlatIndex (maze, pos);
  return & maze->cells [flat];
}
```

```
void
Maze_InitCell (Maze * maze, Pos pos)
{
  Cell_Init (Maze_CellAt (maze, pos));
}
```

```
void
Maze_InitAllCells (Maze * maze)
{
  for (int row= 0; row < maze->dim.row; row++)
    for (int col= 0; col < maze->dim.col; col++)
      Maze_InitCell (maze, Pos_Make (row, col));
}
```

```
void
Maze_Init (Maze * maze, Pos dim)
{
  maze->dim= dim;
  maze->cells= malloc (Maze_NbCells (maze) * sizeof * maze->cells);
  assert (maze->cells != NULL);
  Maze_InitAllCells (maze);
}
```

```
void
Maze_Clean (Maze * maze)
{
  free (maze->cells);
}
```

```
void
MainTest_Maze (void)
{
  Maze maze;
  Pos dim= Pos_Make (5, 3);
  Maze_Init (& maze, dim);
  for (int row= 0; row < dim.row; row++) {
    for (int col= 0; col < dim.col; col++) {
      Cell * cell= Maze_CellAt (& maze, Pos_Make (row, col));
      printf ("[(%d,%d)␣%s]␣", row, col, Cell_ToString (cell));
    }
    printf ("\n");
  }
  Maze_Clean (& maze);
}
```

# 5 Affichage graphique

```
void
Cell_Draw (Cell const * cell, bx_window win, Coord top_left, int side)
{
  Coord diagonal = Coord_FromXY (side-2, side-2);
  Coord bot_right= Coord_Sum (top_left, diagonal);
  Coord top_right= Coord_FromXY (bot_right.x, top_left .y);
  Coord bot_left = Coord_FromXY (top_left .x, bot_right.y);

  bx_set_color (bx_blue ());
  if (Cell_HasWall (cell, DIR_NORTH)) Util_DrawLine (win, top_left , top_right);
  if (Cell_HasWall (cell, DIR_EAST )) Util_DrawLine (win, top_right, bot_right);
  if (Cell_HasWall (cell, DIR_SOUTH)) Util_DrawLine (win, bot_left , bot_right);
  if (Cell_HasWall (cell, DIR_WEST )) Util_DrawLine (win, top_left , bot_left );
}
```

```
Coord
Maze_CellCoord (Pos pos, Coord maze_coord, int cell_side)
{
  Coord coord= Coord_FromXY (pos.col, pos.row);
  coord= Coord_Scale (coord, cell_side);
  coord= Coord_Sum (coord, maze_coord);
  return coord;
}
```

```
void
Maze_Draw (Maze const * maze, bx_window win, Coord maze_coord, int cell_side)
{
  for (int row= 0; row < maze->dim.row; row++) {
    for (int col= 0; col < maze->dim.col; col++) {
      Pos pos= Pos_Make (row, col);
      Coord cell_coord= Maze_CellCoord (pos, maze_coord, cell_side);
      Cell * cell= Maze_CellAt (maze, pos);
      Cell_Draw (cell, win, cell_coord, cell_side);
    }
  }
}
```

```
void
MainTest_Drawing (void)
{
  Maze maze;
  Pos maze_dim= Pos_Make (10, 15);
  Maze_Init (& maze, maze_dim);

  Cell_EraseWall (Maze_CellAt (& maze, Pos_Make ( 0,  7)), DIR_NORTH);
  Cell_EraseWall (Maze_CellAt (& maze, Pos_Make ( 9,  4)), DIR_SOUTH);
  Cell_EraseWall (Maze_CellAt (& maze, Pos_Make ( 3,  0)), DIR_WEST );
  Cell_EraseWall (Maze_CellAt (& maze, Pos_Make ( 7, 14)), DIR_EAST );

  bx_init ();
  bx_window win= bx_create_window ("Maze␣Drawing", 0, 0, 340, 240);
  Coord maze_coord= Coord_FromXY (20, 20);
  int cell_side= 20;
  Maze_Draw (& maze, win, maze_coord, cell_side);
  bx_show_canvas (win, 0);

  Maze_Clean (& maze);
  bx_loop ();
}
```

# 6 Destruction aléatoire de murs

```
bool
Maze_RowIsValid (Maze const * maze, int row)
{
  return 0 <= row && row < maze->dim.row;
}
```

```
bool
Maze_ColIsValid (Maze const * maze, int col)
{
  return 0 <= col && col < maze->dim.col;
}
```

```
bool
Maze_PosIsValid (Maze const * maze, Pos pos)
{
  return Maze_RowIsValid (maze, pos.row)
    &&   Maze_ColIsValid (maze, pos.col);
}
```

```
Pos
Maze_RandomPos (Maze const * maze)
{
  return Pos_Random (maze->dim);
}
```

```
void
Maze_EraseCellWall (Maze * maze, Pos pos, Dir dir)
{
  Cell_EraseWall (Maze_CellAt (maze, pos), dir);
}
```

```
void
Maze_EraseRandomWalls_Iter (Maze * maze, int nb_tries)
{
  for (int k= 0; k < nb_tries; k++) {
    Pos pos= Maze_RandomPos (maze);
    Dir dir= Dir_Random ();
    Pos neighbor= Pos_Neighbor (pos, dir);
    if ( ! Maze_PosIsValid (maze, neighbor)) continue;
    Maze_EraseCellWall (maze, pos, dir);
    Maze_EraseCellWall (maze, neighbor, Dir_Opposite (dir));
  }
}
```

```
void
Maze_EraseRandomWalls_Rec (Maze * maze, int nb_tries)
{
  if (nb_tries == 0) return;

  Pos pos= Maze_RandomPos (maze);
  Dir dir= Dir_Random ();
  Pos neighbor= Pos_Neighbor (pos, dir);
  if (Maze_PosIsValid (maze, neighbor)) {
    Maze_EraseCellWall (maze, pos, dir);
    Maze_EraseCellWall (maze, neighbor, Dir_Opposite (dir));
  }
  Maze_EraseRandomWalls_Rec (maze, nb_tries-1);
}
```

```
void
MainTest_EraseRandomWalls (void)
{
  bx_init ();
  bx_window win= bx_create_window ("Random␣Erasure", 0, 0, 340, 240);

  Maze maze;
  Pos maze_dim= Pos_Make (10, 15);
  Maze_Init (& maze, maze_dim);
  int nb_erasures= 15;
  Maze_EraseRandomWalls_Iter (& maze, nb_erasures);

  Coord maze_coord= Coord_FromXY (20, 20);
  int cell_side= 20;
  Maze_Draw (& maze, win, maze_coord, cell_side);
  bx_show_canvas (win, 0);

  Maze_Clean (& maze);
  bx_loop ();
}
```

# 7 Marche aléatoire

```
bool
Maze_CellHasWall (Maze const * maze, Pos pos, Dir dir)
{
  return Cell_HasWall (Maze_CellAt (maze, pos), dir);
}
```

```
bool
Maze_CellIsIsolated (Maze const * maze, Pos pos)
{
  return Cell_IsIsolated (Maze_CellAt (maze, pos));
}
```

```
Dir
Maze_CountIsolatedNeighbors (Maze const * maze, Pos pos)
{
  int count= 0;
  for (Dir dir= 0; dir < NB_DIRS; dir++) {
    Pos neighbor= Pos_Neighbor (pos, dir);
    if ( ! Maze_PosIsValid     (maze, neighbor)) continue;
    if ( ! Maze_CellIsIsolated (maze, neighbor)) continue;
    count++;
  }
  return count;
}
```

```
Dir
Maze_IsolatedDir (Maze const * maze, Pos pos, int n)
{
  int count= 0
  for (Dir dir= 0; dir < NB_DIRS; dir++) {
    Pos neighbor= Pos_Neighbor (pos, dir);
    if ( !  Maze_PosIsValid    (maze, neighbor)) continue;
    if ( ! Maze_CellIsIsolated (maze, neighbor)) continue;
    if (count == n) return dir;
    count++;
  }
  return DIR_NONE;
}
```

```
int
Maze_RandomIsolatedDir (Maze const * maze, Pos pos, int count)
{
  return Maze_IsolatedDir (maze, pos, Int_Random (count));
}
```

```
Pos
Maze_DoRandomWalkStep (Maze * maze, Pos current, int nb_isolated)
{
  Dir dir= Maze_RandomIsolatedDir (maze, current, nb_isolated);
  Maze_EraseCellWall (maze, current, dir);
  Pos next= Pos_Neighbor (current, dir);
  Maze_EraseCellWall (maze, next, Dir_Opposite (dir));
  return next;
}
```

```
void
Maze_DoRandomWalk_Iter (Maze * maze, Pos start, bx_window win,
                        Coord maze_coord, int cell_side, int delay)
{
  Pos pos= start;
  for (;;) {
    int nb_isolated= Maze_CountIsolatedNeighbors (maze, pos);
    if (nb_isolated == 0) break;
    pos= Maze_DoRandomWalkStep (maze, pos, nb_isolated);
    bx_clear_canvas (win, bx_white ());
    Maze_Draw (maze, win, maze_coord, cell_side);
    bx_show_canvas (win, delay);
  }
}
```

```
void
Maze_DoRandomWalk_Rec (Maze * maze, Pos start, bx_window win,
                       Coord maze_coord, int cell_side, int delay)
{
  Pos pos= start;
  int nb_isolated= Maze_CountIsolatedNeighbors (maze, pos);
  if (nb_isolated == 0) return;
  pos= Maze_DoRandomWalkStep (maze, pos, nb_isolated);
  bx_clear_canvas (win, bx_white ());
  Maze_Draw (maze, win, maze_coord, cell_side);
  bx_show_canvas (win, delay);
  Maze_DoRandomWalk_Rec (maze, pos, win, maze_coord, cell_side, delay);
}
```

```
void
MainTest_DoRandomWalk (void)
{
  bx_init ();
  bx_window win= bx_create_window ("Random␣Walk", 0, 0, 340, 240);

  Maze maze;
  Pos maze_dim= Pos_Make (10, 15);
  Maze_Init (& maze, maze_dim);
  Maze_Draw (& maze, win, maze_coord, cell_side);

  Pos start= Pos_Random (maze_dim);
  Coord maze_coord= Coord_FromXY (20, 20);
  int cell_side= 20, delay= 10;

  srand(time(NULL));
  Maze_DoRandomWalk_Rec (& maze, start, win, maze_coord, cell_side, delay);
  Maze_Clean (& maze);
  bx_loop ();
}
```

# 8 Pile de positions : `struct Stack`

```
void
Stack_Init (Stack * s, int capacity)
{
  s->capacity= capacity;
  s->elements= malloc (capacity * sizeof * s->elements);
  assert (s->elements != NULL);
  s->nb_elements= 0;
}
```

```
void
Stack_Clean (Stack * s)
{
  free (s->elements);
}
```

```
bool
Stack_IsEmpty (Stack const * s)
{
  return s->nb_elements == 0;
}
```

```
bool
Stack_IsFull (Stack const * s)
{
  return s->nb_elements == s->capacity;
}
```

```
void
Stack_Push (Stack * s, Pos pushed)
{
  assert ( ! Stack_IsFull (s));
  s->elements [s->nb_elements ++]= pushed;
}
```

```
Pos
Stack_Pop (Stack * s)
{
  assert ( ! Stack_IsEmpty (s));
  return s->elements [-- s->nb_elements];
}
```

```
Pos
Stack_Peek (Stack const * s)
{
  assert ( ! Stack_IsEmpty (s));
  return s->elements [s->nb_elements -1];
}
```

```
void
Stack_Print (Stack const * s, FILE * file)
{
  fprintf (file, "[");
  for (int k= 0; k < s->nb_elements; k++) {
    Pos pos= s->elements [k];
    fprintf (file, "␣(%d,%d)", pos.row, pos.col);
  }
  fprintf (file, "␣]␣FILL-RATIO␣%d/%d", s->nb_elements, s->capacity);
  if (Stack_IsEmpty (s)) fprintf (stdout, "␣(EMPTY)");
  if (Stack_IsFull  (s)) fprintf (stdout, "␣(FULL)");
}
```

```
void
MainTest_Stack (void)
{
  Stack s;
  int capacity= 4;
  Stack_Init  (& s, capacity);
  Stack_Print (& s, stdout);
  fprintf (stdout, "\n");

  srand (time (NULL));
  Pos dim= Pos_Make (20,20);
  char const ops []= "++++--++----";
  for (int k= 0; ops [k] != '\0'; k++) {
    Pos pos;
    if (ops [k] == '+') { pos= Pos_Random (dim); Stack_Push (& s, pos); }
    else                { pos= Stack_Pop (& s); }
    Stack_Print (& s, stdout);
    fprintf (stdout, "␣AFTER␣%c(%d,%d)\n", ops [k], pos.row, pos.col);
  }
  Stack_Clean (& s);
}
```

# 9 Génération aléatoire

```
Coord
Maze_CellCenter (Pos pos, Coord maze_coord, int cell_side)
{
  Coord size= Coord_FromXY (cell_side, cell_side);
  Coord shift= Coord_Scale (size, 0.5);
  Coord coord= Maze_CellCoord (pos, maze_coord, cell_side);
  return Coord_Sum (coord, shift);
}
```

```
void
Maze_DrawPath (Maze const * maze, Stack const * pending,
               bx_window win, Coord maze_coord, int cell_side)
{
  if (pending->nb_elements == 0) return;

  Pos pos0= pending->elements [0];
  Coord center0= Maze_CellCenter (pos0, maze_coord, cell_side);
  bx_set_color (bx_red ());
  Util_DrawCircle (win, center0, cell_side * 0.25, true);
  Coord prev_center= center0;
  for (int k= 1; k < pending->nb_elements; k++) {
    Pos pos= pending->elements [k];
    int count= Maze_CountIsolatedNeighbors (maze, pos);
    Coord center= Maze_CellCenter (pos, maze_coord, cell_side);
    bx_set_color (count ? bx_green () : bx_yellow ());
    Util_DrawCircle (win, center, cell_side * 0.25, true);
    bx_set_color (bx_black ());
    Util_DrawLine (win, prev_center, center);
    prev_center= center;
  }
}
```

```
void
Maze_DoRandomGenStep (Maze * maze, Stack * pending)
{
  Pos current= Stack_Peek (pending);
  int count= Maze_CountIsolatedNeighbors (maze, current);
  if (count == 0) {
    Stack_Pop (pending);
  } else {
    Dir dir= Maze_RandomIsolatedDir (maze, current, count);
    Pos neighbor= Pos_Neighbor (current, dir);
    Maze_EraseCellWall (maze, current, dir);
    Maze_EraseCellWall (maze, neighbor, Dir_Opposite (dir));
    Stack_Push (pending, neighbor);
  }
}
```

```
void
Maze_DoRandomGen_Iter (Maze * maze, Pos start, bx_window win,
                       Coord maze_coord, int cell_side, int delay)
{
  Stack pending;
  Stack_Init (& pending, Maze_NbCells (maze));
  Stack_Push (& pending, start);
  while ( ! Stack_IsEmpty (& pending)) {
    Maze_DoRandomGenStep (maze, & pending);
    bx_clear_canvas (win, bx_white ());
    Maze_Draw (maze, win, maze_coord, cell_side);
    Maze_DrawPath (maze, & pending, win, maze_coord, cell_side);
    bx_show_canvas (win, delay);
  }
  Stack_Clean (& pending);
}
```

```
void
Maze_DoRandomGen_Rec (Maze * maze, Pos current, bx_window win,
                      Coord maze_coord, int cell_side, int delay)
{
  for (;;) {
    int count= Maze_CountIsolatedNeighbors (maze, current);
    if (count == 0) return;
    Dir dir= Maze_RandomIsolatedDir (maze, current, count);
    Pos neighbor= Pos_Neighbor (current, dir);
    Maze_EraseCellWall (maze, current, dir);
    Maze_EraseCellWall (maze, neighbor, Dir_Opposite (dir));

    bx_clear_canvas (win, bx_white ());
    Maze_Draw (maze, win, maze_coord, cell_side);
    bx_show_canvas (win, delay);
    Maze_DoRandomGen_Rec (maze, neighbor, win, maze_coord, cell_side, delay);
  }
}
```

```
void
MainTest_DoRandomGen (void)
{
  bx_init ();
  bx_window win= bx_create_window ("Random␣Generation", 0, 0, 340, 240);

  Maze maze;
  Pos maze_dim= Pos_Make (10, 15);
  Maze_Init (& maze, maze_dim);

  Coord maze_coord= Coord_FromXY (20, 20);
  int cell_side= 20, delay= 100;
  Maze_Draw (& maze, win, maze_coord, cell_side);

  srand (time(NULL));
  Pos start= Pos_Random (maze_dim);
  Maze_DoRandomGen_Iter (& maze, start, win, maze_coord, cell_side, delay);

  Maze_Clean (& maze);
  bx_loop ();
}
```