# UNE STRUCTURE DE DONNÉES FASCINANTE : PROGRAMMER UN LABYRINTHE EN C++

**Rodrigue S. Mompelat, PhD**

Voici le code C++ de mon article.

### Listing1.hpp

```
/* C'est la fenêtre de base pour y dessiner  */

#ifndef GUI_H
#define GUI_H
#include<X11/Xlib.h>
#include<X11/Xutil.h>

const int Offset = 5;

class BaseWindow {
public:
   BaseWindow(int, int);
   inline unsigned int GetWhiteColor() { return WhiteColor; }
   inline unsigned int GetGreenColor() { return GreenColor; }
   inline int GetWidth() { return width; }
   inline int GetHeight() { return height; }
   inline void Show() { XMapWindow(XDisp, XWindow); }
   inline void Line(int a, int b, int c, int d)
               { XDrawLine(XDisp, XWindow, XGC, a, b, c, d); }
   inline void ColorCell(int a, int b, int width)
               { XFillRectangle(XDisp, XWindow, XGC, a+1, b+1, width-1, width-1); }
   void ChangePencil(unsigned int);
   int AskForEvents();
   void OnKeyPressed();
   void SaveImage();
   void ShowImage();
   virtual void Run()=0;
   virtual void OnExpose()=0;
   virtual ~BaseWindow();
private:
   Display* XDisp;
   Window XWindow;
   XEvent report;
   XGCValues GCValues;
   GC XGC;
   XImage* Canvas;
   int width;
   int height;
   unsigned int WhiteColor;
   unsigned int BlackColor;
   unsigned int RedColor;
   unsigned int GreenColor;
};

#endif
```

**Listing2.cpp**

```
#include"listing1.hpp"

BaseWindow::BaseWindow(int w, int h)
   : width(w), height(h), Canvas(0)
{
   XDisp = XOpenDisplay(NULL);  //Overture
   int sn = DefaultScreen(XDisp);
   unsigned long vm = CWBackPixel | CWBorderPixel;
   BlackColor = BlackPixel(XDisp, sn);
   WhiteColor =  WhitePixel(XDisp, sn);
   XSetWindowAttributes wa;
   wa.border_pixel = BlackColor;
   wa.background_pixel = WhiteColor;  // Couleur du fond
   XWindow = XCreateWindow(XDisp, RootWindow(XDisp, sn),
                                         0, 0,
                                         width, height, 4, DefaultDepth(XDisp, sn),
                                         InputOutput, NULL, vm, &wa);
   XGC = XCreateGC(XDisp, XWindow, 0, &GCValues);
   XColor MyColor; Colormap MyCM = DefaultColormap(XDisp, sn);
   MyColor.green = 255*255; MyColor.blue =  MyColor.red = 0;
   XAllocColor(XDisp, MyCM, &MyColor); GreenColor = MyColor.pixel;
}

// Pour colorer les cellules du chemin
void BaseWindow::ChangePencil(unsigned int Color)
{
   XSetForeground(XDisp, XGC, Color);
}
int BaseWindow::AskForEvents() {
   XSelectInput(XDisp, XWindow,
            ExposureMask | KeyPressMask |
            ButtonPressMask | ButtonReleaseMask);
   XNextEvent(XDisp, &report);
   return report.type;
}
```

```
// On garde l'oeuvre d'art
void BaseWindow::SaveImage() {
   Canvas = XGetImage(XDisp, XWindow,
                    0, 0, width, height,
                    AllPlanes, ZPixmap);
}
// Actualise l'image
void BaseWindow::ShowImage() {
   XPutImage(XDisp, XWindow,
             XGC, Canvas, 0, 0, 0, 0,
             width, height);

}

BaseWindow::~BaseWindow() {
   if (Canvas) XDestroyImage(Canvas);
   XFreeGC(XDisp, XGC);
   XDestroyWindow(XDisp, XWindow);
   XCloseDisplay(XDisp);
}
```

**Listing3.hpp**

```
#ifndef NODE_H
#define NODE_H

// Type du mouvement dans lab, D(own), R(ight), U(p), L(eft), C=Aucun
enum Move { D, R, U, L, C};
class Point {
public:
   Point();
   Point(int, int);
   int x, y;
   friend bool operator!=(Point& , Point& );
};

class Pile {
public:
   Pile();
   Pile(Point );
   bool Empty();
   void Push(Point& );
   Point Pop();
   //~Pile
private:
   Point Data;
   Pile* Head;
   Pile* next;
};

#endif
```

**Listing4.cpp**

```
#include"listing3.hpp"

Point::Point() : x(0), y(0) {}
Point::Point(int i, int j) : x(i), y(j) {}
bool operator!=(Point& P, Point& Q) {
   return (P.x != Q.x) || (P.y != Q.y);
}

Pile::Pile() : Head(0), next(0) {}
Pile::Pile(Point P) : next(0)  {
   Head = new Pile;
   Head→Data = P;
}
bool Pile::Empty() { return Head = = 0; }
void Pile::Push(Point& P)
{
   if (Head) {
                 Pile* p = new Pile;
                 p→Data = P;
                 p→next = Head;
                 Head = p;
   }
   else Head = new Pile(P);
}

Point Pile::Pop() {

   Pile* q=Head;
   Point P=q→Data;
   Head=Head→next;
   delete q;
   return P;
}
```

**Listing5.hpp**

```
/* Les structures concernant les chambres et le labyrinthe */

#ifndef CELL_H
#define CELL_H
// Les portes D= Down, C = Closed, R=Right, L=Locked
// En binaire
enum DoorStates {
   DCRC=0,
   DORC=1,
   DLRC=3,
   DCRO=4,
   DORO=5,
   DLRO=7,
   DCRL=12,
   DORL=13,
   DLRL=15
};


class Chambre {
public:
   Chambre();
   Chambre* suivant;         // Pour connecter la liste
   Chambre* precedent;
   inline unsigned int GetType() { return Type; }
   inline DoorStates GetState() { return State; }
   void SetStates(int );
   void SetType(unsigned int);
   void OuvreDroite();
   void LockRight();
   void OuvreDessous();
   void LockDown();
private:
   unsigned int Type;
   DoorStates State;
};


class Labyrinthe {
public:
   Labyrinthe(int, int);
   Chambre& operator()(int, int);
   ~Labyrinthe();
   inline int GetLines() { return lignes; }
   inline int GetColonnes() { return colonnes; }
private:
   void Glue(Chambre& , Chambre& );
   int lignes;
   int colonnes;
   Chambre* Puzzle;
};

#endif
```

**Listing6.cpp**

```
include<stdlib.h>
#include"listing5.hpp"

/* Initialement toutes les chambres ont leurs portes fermées, acune chambre
 * n'est liée et toutes ont donc un type différent
 */
Chambre::Chambre()
   : suivant(0), precedent(0), State(DCRC)
{
   static unsigned int counter = 1;
   Type = counter;
   counter++;
}

void Chambre::SetType(unsigned int t) { Type = t; }
void Chambre::LockDown()      { State = DoorStates(State | 3); }
void Chambre::OuvreDessous() { State = DoorStates(State | 1); }
void Chambre::OuvreDroite()  { State = DoorStates(State | 4); }
void Chambre::LockRight()     { State = DoorStates(State | 12);}


// Cette fonction colle les deux listes contenant les deux chambres
void Labyrinthe::Glue(Chambre& C1, Chambre& C2)
{
   Chambre* p=&C1;
   Chambre* q=&C2;
   while (q→precedent)
               q=q→precedent;
   while (p→suivant)
               p=p→suivant;
   unsigned int GlueType=p→GetType();
   p→suivant = q;
   q→precedent = p;
   while (q) {
               q→SetType(GlueType);
               q=q→suivant;
   }
}
```

```
// C'est ici que l'on construit le labyrinthe
Labyrinthe::Labyrinthe(int n, int m)
   : lignes(n), colonnes(m)
{
   const unsigned int total = lignes*colonnes;
   Puzzle = new Chambre[total];
   for(int i=1; i<=lignes; i++)
                Puzzle[i*colonnes - 1].LockRight();
   for(unsigned int i=(lignes -1)*colonnes; i<total; i++)
                Puzzle[i].LockDown();
   for (unsigned int i=0; i<50*total; i++) {
                //Un peu de desordre d'abord
                int seed = random() % total;
                switch (Puzzle[seed].GetState()) {
                   case DCRC:
                                if (random() % 2 ) {
                                   if ( Puzzle[seed].GetType() = =
                                                 Puzzle[seed+1].GetType() )
                                                Puzzle[seed].LockRight();
                                   else {
                                                Puzzle[seed].OuvreDroite();
                                                Glue(Puzzle[seed], Puzzle[seed+1]);
                                   }
                                } else {
                                   if ( Puzzle[seed].GetType() = =
                                                 Puzzle[seed+colonnes].GetType() )
                                                Puzzle[seed].LockDown();
                                   else {
                                                Puzzle[seed].OuvreDessous();
                                                Glue(Puzzle[seed], Puzzle[seed+colonnes]);
                                   }
                                }
                                break;
                   case DORC:
                   case DLRC:
                                if ( Puzzle[seed].GetType() = =
                                    Puzzle[seed+1].GetType() )
                                   Puzzle[seed].LockRight();
                                else {
                                  Puzzle[seed].OuvreDroite();
                                  Glue(Puzzle[seed], Puzzle[seed+1]);
                                }
                                break;
                   case DCRO:
                   case DCRL:
                                if ( Puzzle[seed].GetType() = =
                                    Puzzle[seed+colonnes].GetType() )
                                   Puzzle[seed].LockDown();
                                else {
                                   Puzzle[seed].OuvreDessous();
```

```
                    Glue(Puzzle[seed], Puzzle[seed+colonnes]);
                  }
                  break;
          default:                  // On peut rien faire avec le reste
                  break;
        }
}
// Maintenant il faut compléter tout le labyrinthe
for(unsigned int i=0; i<total; i++) {
        switch (Puzzle[i].GetState()) {
          case DCRC:
                  if (random() % 2 ) {
                    if ( Puzzle[i].GetType() = =
                             Puzzle[i+colonnes].GetType() )
                            Puzzle[i].LockDown();
                    else {
                            Puzzle[i].OuvreDessous();
                            Glue(Puzzle[i], Puzzle[i+colonnes]);
                    }
                  } else {
                    if ( Puzzle[i].GetType() = =
                             Puzzle[i+1].GetType() )
                            Puzzle[i].LockRight();
                    else {
                            Puzzle[i].OuvreDroite();
                            Glue(Puzzle[i], Puzzle[i+1]);
                    }
                  }
                  break;
          case DORC:
          case DLRC:
                  if ( Puzzle[i].GetType() = =
                     Puzzle[i+1].GetType() )
                    Puzzle[i].LockRight();
                  else {
                    Puzzle[i].OuvreDroite();
                    Glue(Puzzle[i], Puzzle[i+1]);
                  }
                  break;
          case DCRO:
          case DCRL:
                  if ( Puzzle[i].GetType() = =
                     Puzzle[i+colonnes].GetType() )
                    Puzzle[i].LockDown();
                  else {
                    Puzzle[i].OuvreDessous();
                    Glue(Puzzle[i], Puzzle[i+colonnes]);
                  }
                  break;
          default:
```

```
                                        break;
                    }
    }
}

Chambre& Labyrinthe::operator()(int i, int j)
{ return Puzzle[j + i*colonnes]; }

Labyrinthe::~Labyrinthe()
{ delete[] Puzzle; }
```

**Listing7.hpp**

```
/* Le « gros objet » */

#ifndef MAZE_H
#define MAZE_H
#include"listing1.hpp"
#include"listing5.hpp"
#include"listing3.hpp"

class MazeObject : public Labyrinthe, public BaseWindow {
public:
   MazeObject(int, int, int);
   void Run();
   void OnExpose();
   void PlotMaze();
   void SolveMaze();
private:
   bool Posible(Point, Move);
   bool Visite(Point, Move);
   int CellSize;
   Point Enter;
   Point Exit;
};
#endif
```

**Listing8.cpp**

```
/* L'objet principal, une fenêtre avec un labyrinthe */

#include<stdlib.h>
#include"listing7.hpp"

extern const int Offset;

MazeObject::MazeObject(int lin, int cols, int size)
   : Labyrinthe(lin, cols),
     BaseWindow( size*cols + 2*Offset, size*lin + 2*Offset) ,
     CellSize(size)
{
   Enter = Point(random() % lin, 0);
   Exit = Point(random() % lin, cols -1);
}

void MazeObject::OnExpose()
{
   static bool FirstTime = true;
   if (FirstTime) {
      PlotMaze();
                SolveMaze();
                SaveImage();
      FirstTime = false;
   }
   else
                ShowImage();
}

void MazeObject::Run()
{
   Show();
   for (;;) {
      switch (AskForEvents()) {
         case Expose:
            OnExpose();
            break;
         case KeyPress:
            exit(0);
            break;
         default:
            break;
      }
   }
}
```

```
// La routine qui réalise le graphique

void MazeObject::PlotMaze() {
   Line(Offset, Offset, GetWidth() - Offset, Offset);
   Line(Offset, Offset, Offset, GetHeight() - Offset);
   int x=0; int y=0;
   for (int i=0; i<GetLines()*GetColonnes(); i++) {
                switch ((*this)(y,x).GetState() ){
                   case DLRL:
         case DCRC:
         case DLRC:
         case DCRL:
                                Line(CellSize*x + Offset, CellSize*(y+1) + Offset,
                                    CellSize*(x+1) + Offset, CellSize*(y+1) + Offset);
                                Line(CellSize*(x + 1) + Offset, CellSize*y + Offset,
                                    CellSize*(x + 1) + Offset, CellSize*(y + 1) + Offset);
            break;
         case DORC:
         case DORL:
                                Line(CellSize*(x + 1) + Offset, CellSize*y + Offset,
                                    CellSize*(x + 1) + Offset, CellSize*(y + 1) + Offset);
            break;
         case DCRO:
         case DLRO:
                                Line(CellSize*x + Offset, CellSize*(y+1) + Offset,
                                    CellSize*(x+1) + Offset, CellSize*(y+1) + Offset);
            break;
         case DORO:
            break;
                }
                x++;
      if (x = = GetColonnes()) { x=0; y++; }
   }
   ChangePencil(GetWhiteColor());
   Line(Offset + Enter.y*CellSize, Offset + Enter.x*CellSize,
                 Offset + Enter.y*CellSize, Offset + (Enter.x+1)*CellSize);
   Line(Offset + (Exit.y+1)*CellSize, Offset + Exit.x*CellSize,
   Offset + (Exit.y+1)*CellSize, Offset + (Exit.x+1)*CellSize);
}

// Est-il possible d'aller vers la direction M?
bool MazeObject::Posible(Point P, Move M) {
   bool Answer=false;
   DoorStates DoorS = (*this)(P.x, P.y).GetState();
   switch (M) {
                case D:
                   Answer = (DoorS = = DORC || DoorS = = DORO || DoorS = = DORL);
                   break;
                case R:
                   Answer = (DoorS = = DCRO || DoorS = = DORO || DoorS = = DLRO);
```

```
                    break;
                case U:
                    if (P.x) {
                                DoorS = (*this)(P.x-1, P.y).GetState();
                                Answer = (DoorS = = DORC || DoorS = = DORO || DoorS = = DORL);
                    }
                    break;
                case L:
                    if (P.y) {
                                DoorS = (*this)(P.x, P.y-1).GetState();
                                Answer = (DoorS = = DCRO || DoorS = = DORO || DoorS = = DLRO);
                    }
                    break;
                default:
                    break;
    }
    return Answer;
}

// Est-on déjà passé par là ?
bool MazeObject::Visite(Point P, Move M) {
    bool Answer;
    switch (M) {
                case D:
                    if (P.x != GetLines()-1)
                                Answer =  (*this)(P.x+1, P.y).GetType() = = 1;
                    break;
                case R:
                    if (P.y != GetColonnes() - 1)
                                return (*this)(P.x, P.y+1).GetType() = = 1;
                    break;
                case U:
                    if (P.x)
                                return (*this)(P.x-1, P.y).GetType() = = 1;
                    break;
                case L:
                    if (P.y)
                                return (*this)(P.x, P.y-1).GetType() = = 1;
                default:
                    break;
    }
}
```

```
void MazeObject::SolveMaze()
{
   {  // On réinitialise les chambres
                int x=0; int y=0;
                for (int i=0; i<GetLines()*GetColonnes(); i++) {
                   (*this)(x, y).SetType(0);
                   y++;
                   if (y = =GetColonnes()) { y=0; x++; }
                }
   }
   Point Position=Enter;
   (*this)(Position.x, Position.y).SetType(1);

   Pile Path(Position);
   ChangePencil(GetGreenColor());
   ColorCell(Position.y*CellSize+Offset,
                     Position.x*CellSize+Offset, CellSize);
   while (Position != Exit) {
                ChangePencil(GetGreenColor());
                if (Posible(Position, D) && !Visite(Position, D) ) {
                   Position = Point(Position.x+1, Position.y);
                   (*this)(Position.x, Position.y).SetType(1);
                   Path.Push(Position);
                } else if (Posible(Position, R) && !Visite(Position, R)) {
                   Position = Point(Position.x, Position.y+1);
                   (*this)(Position.x, Position.y).SetType(1);
                   Path.Push(Position);
                } else if (Posible(Position, U) && !Visite(Position, U)) {
                   Position = Point(Position.x-1, Position.y);
                   (*this)(Position.x, Position.y).SetType(1);
                   Path.Push(Position);
                } else {
                   if (Posible(Position, L) && !Visite(Position, L)) {
                                Position = Point(Position.x, Position.y-1);
                                (*this)(Position.x, Position.y).SetType(1);
                                Path.Push(Position);
                   } else {
                                Position = Path.Pop();
                                Path.Push(Position = Path.Pop());
                                ChangePencil(GetWhiteColor());
                   }
                }
   }
   ChangePencil(GetGreenColor());
   while (!Path.Empty()) {
                   Position = Path.Pop();
                   ColorCell(Position.y*CellSize+Offset,
                                    Position.x*CellSize+Offset, CellSize);
   }
}
```

**Listing9.cpp**

```
#include<iostream>
#include<stdlib.h>
#include"listing7.hpp"

int main(int argc, char *argv[])
{
   if (argc != 4) {
      cout<<"Use: rodrimaze lines cols cellsize "<<endl;
      exit(-1);
   }
   else {
      int lines =atoi(argv[1]);
      int cols =atoi(argv[2]);
      int cellsize =atoi(argv[3]);
                MazeObject Arena(lines, cols, cellsize);
                Arena.Run();
   }
}
```

**Fonction ”Show”**

```
CC=g++

LDLIBS=-lm -lX11
CFLAGS=-Wall
XPATH=-L/usr/X11/lib

maze: listing2.o listing4.o listing6.o listing9.o
		$(CC) $(LDLIBS) $(XPATH) listing2.o listing4.o listing6.o listing9.o -o maze
listing2.o: listing2.cc
		$(CC) $(CFLAGS) -c listing2.cc
listing4.o: listing4.cc
		$(CC) $(CFLAGS) -c listing4.cc
listing6.o: listing6.cc
		$(CC) $(CFLAGS) -c listing6.cc
listing9.o: listing9.cc
		$(CC) $(CFLAGS) -c listing9.cc
```

**Makefile**

```
CC=g++

LDLIBS=-lm -lX11
CFLAGS=-Wall
XPATH=-L/usr/X11/lib

rodrimaze: listing2.o listing4.o listing6.o listing8.o listing9.o
	$(CC) $(LDLIBS) $(XPATH) listing2.o listing4.o listing6.o listing8.o listing9.o -o rodrimaze
listing2.o: listing2.cc
	$(CC) $(CFLAGS) -c listing2.cc
listing4.o: listing4.cc
	$(CC) $(CFLAGS) -c listing4.cc
listing6.o: listing6.cc
	$(CC) $(CFLAGS) -c listing6.cc
listing8.o: listing8.cc
	$(CC) $(CFLAGS) -c listing8.cc
listing9.o: listing9.cc
	$(CC) $(CFLAGS) -c listing9.cc
```