TP3 : La programmation orientée objet (1)

Préambule

Qu'est ce que la POO ?

Qu'est ce qu'un objet ? Il s'agit d'un mélange de plusieurs variables et fonctions. Imaginez que vous avez créé un programme qui permet de résoudre des équations différentielles ordinaires : vous pouvez afficher vos solutions, calculer des ordres de convergence, comparer deux méthodes ... Le code est complexe : il aura besoin de plusieurs fonctions qui s'appellent entre elles, ainsi que de variables pour mémoriser la solution au cours du temps, la méthode utilisée, le pas de temps choisi ... Au final, votre code est composé de plusieurs fonctions et variables. Votre code sera difficilement accessible par quelqu'un qui n'est pas un expert du sujet : Quelle fonction il faut appeler en premier ? Quelles valeurs doit-on envoyer à quelle fonction pour afficher la solution ? etc ... Votre solution est de concevoir votre code de *manière orientée objet*. Ce qui signifie que vous placerez tout votre code dans une grande boîte. Cette boîte c'est ce qu'on appelle un **objet**. L'objet contient toutes les fonctions et variables mais elles sont masquées pour l'utilisateur. Seulement quelques outils sont proposés à l'utilisateur comme par exemple : définir mon pas de temps, mon intervalle de calcul et ma méthode, calculer l'ordre de la méthode, afficher la solution ...

En quelques lignes :

- La programmation orientée objet est une façon de concevoir son code dans laquelle on manipule des objets.
- Les objets peuvent être complexes mais leur utilisation est simplifiée. C'est un des avantages de la programmation orientée objet.
- Un objet est constitué d'attributs et de méthodes, c'est-à-dire de variables et de fonctions membres.
- On appelle les méthodes de ces objets pour les modifier ou obtenir des informations.

Qu'est ce qu'une classe ? Pour créer un objet, il faut d'abord créer une classe. Créer une classe consiste à définir les plans de l'objet. Une fois que la classe est faite (le plan), il est possible de créer autant d'objets du même type. Vocabulaire : on dit qu'un objet est une **instance** d'une classe.

Objectifs

- Comprendre le principe de la programmation objet à l'aide de l'objet *string* (qui n'est pas un type comme les autres ...)
- Implémenter sa première classe : constructeur, attributs et méthodes et utilisation des pointeurs.

Exercices

Exercice 1 - Un premier exemple d'objet : le type string

1. Un ordinateur ne sait pas gérer du texte. Le type *char* stocke un nombre interprété comme une lettre grâce à la table ASCII (ex : le nombre 48 correspond au caractère *0*, 65 à *A*). Le type *char* ne correspond qu'à 1 seul caractère donc les textes sont des tableaux de *char*. Sans utiliser *string*, nous devrions donc utiliser ce type de code pour construire et afficher des mots :

```
#include<iostream>
#include<vector>
using namespace std;
int main()
{
  vector<char> mot(4); //Définit un mot comme un tableau de char
  mot[0]='C'; mot[1]='i'; mot[2]='a'; mot[3]='o'; //Mon mot lettre par lettre
  cout << mot[0] << mot[1] << mot[2] << mot[3] << endl; //Affichage du mot
  mot.resize(5); //Changement de mot = changement de taille
  mot[0]='S'; mot[1]='a'; mot[2]='l'; mot[3]='u'; mot[4]='t'; //Nouveau mot
  cout << mot[0] << mot[1] << mot[2] << mot[3] << mot[4] << endl; //Affichage
  return 0;
}
```

2. C'est là qu'intervient la programmation orientée objet : on place le tout dans une boîte facile à utiliser. Le même code avec l'objet *string* devient :

```
#include <iostream>
#include <string> // Obligatoire pour utiliser les objets string
using namespace std;
int main()
{
  string mot("Ciao"); //Création d'un objet 'mot' de type string
  cout << mot << endl; //Affichage
  mot = "Salut"; //Changement du contenu
  cout << mot << endl; //Affichage nouveau mot
  return 0;
}
```

Et nous pouvons aller encore plus loin :

```
string mot1("Ciao"), mot2("Salut"), mot3, mot4;
mot3 = "Vous préférez dire " + mot1 + " ou " + mot2 + " ?";
cout << mot3 << endl; //Concaténation

if (mot1 == mot2) //Comparaison de 2 mots
    cout << "Les deux mots sont identiques." << endl;
else
    cout << "Les deux mots sont differents." << endl;

mot4 = mot2.substr(2); //Extraction
cout << mot2 << " sans les 2 premières lettres devient " << mot4 << endl;
```

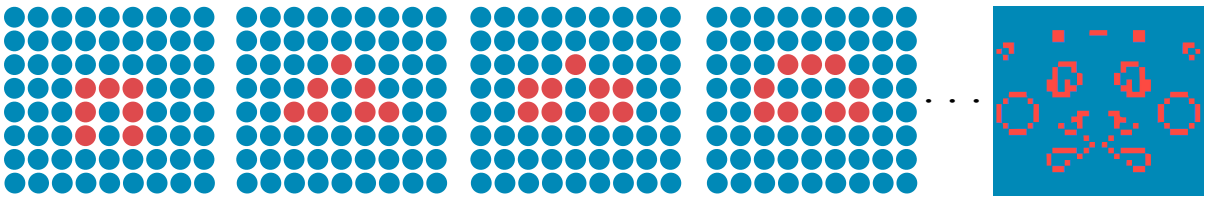
Réfléchir à comment sont implémentées (à partir de *char*) les méthodes : "+", "=" et "substr".

Exercice 2 - Ma première classe (Exercice noté)

Le code sera noté : ne pas oublier de le commenter pour faciliter sa notation. Un court rapport présentera les difficultés que vous avez rencontrées ; comment vous les avez surmontées ainsi que les résultats que vous avez obtenus.
C'est parti ! Nous allons créer une première classe. L'objectif est d'implémenter *le jeu de la vie* qui est un *automate cellulaire*. Chaque élément d'une grille (càd chaque pixel) évolue en fonction de ses voisins. Il peut-être vivant (allumé) ou mort (éteint) et sa situation peut évoluer au cours du temps. Des exemples sont proposés dans ces vidéos : lien 1 et lien 2 (> 1m08).
Le principe est vraiment simple. Le jeu se déroule sur une grille en 2D dont les cases, appelés des *cellules* par analogie avec les cellules vivantes, peuvent prendre deux états distincts : vivantes ou mortes. À chaque étape, l'évolution d'une cellule est entièrement déterminée par l'état de ses huit voisines :

- Une cellule morte possédant exactement trois voisines vivantes devient vivante.
- Une cellule vivante possédant deux ou trois voisines vivantes reste vivante, sinon elle meurt.

La figure, ci-dessous vous propose un exemple qui vous permet d'obtenir un "clown" à la 110ème itération. Vérifiez que vous comprenez le phénomène sur les premières itérations.



1. Nous voulons construire une classe *GameOfLife* qui permet de créer des objets *game* avec lesquels on peut jouer. L'idée est donc de pouvoir lancer le code suivant :

```
int main()
{
   int iterationNb = 110; // nombre d'itérations que l'on va faire
   GameOfLife game(50, 50); // définition d'un objet "game" de taille 50x50
   game.initialisation(); // on initialise (ex : figure ci-dessus)
   game.saveSolution(0); // on sauvegarde l'initialisation

   for(int i=1; i<=iterationNb; i++) // boucle sur les itérations
   {
      game.play(); // on joue (mise à jour du statut des cellules)
      game.saveSolution(i); // on sauvegarde les solutions obtenues
   }

   return 0;
}
```

Bien sur vous compilez ce code vous obtenez une erreur proche de celle-ci :

```
[gameoflife] $ g++ -std=c++11 -o run main.cc
main.cc:5:2: error: unknown type name 'GameOfLife'
       GameOfLife game(50, 50);
```

En effet la classe *GameOfLife* n'est pas encore définie !

2. Dans un fichier GameOfLife.h placer le prototype de la classe *GameOfLife* :

```
// Les includes dont vous avez besoin dans le fichier .h. Attention à ne pas
// mettre de using namespace dans le .h car c'est ce fichier qui va être inclus
// et il peut donc contaminer le reste du code ...
#include <vector>

class GameOfLife { //Définition de votre classe

  public: //Méthodes et opérateurs de la classe
     //Constructeur qui permet de construire un objet de la classe. Il peut y
     //avoir différents constructeurs (comme nous verrons dans la suite)
     //Il doit porter le même nom que la classe !
     GameOfLife(int numLines, int numCols);

     void initialisation(); //Initialisation du jeu
     void play(); //Mise à jour du statut des cellules lors d'une itération
     int nbAlivedNeighbours(int i, int j); //Nombre de voisins vivants pour (i,j)
     void saveSolution(int it); //Écrit un fichier solution au format Paraview

  private: //Les attributs de la classe
     int _numLines; //Nombre de lignes de la grille
     int _numCols; //Nombre de colonnes de la grille
     //La grille qui contient le statut des cellules 1 si cellule vivante,
     //0 si elle est morte. Elle est définie comme un vecteur de vecteur de int
     std::vector<std::vector<int> > _grid;
};
```

Dans un fichier GameOfLife.cpp il faut définir les fonctions. Pour savoir que les fonctions dépendent de la classe il faut rajouter le nom de la classe devant :

```
type NomClasse::nomFonction(arguments)
```

Ce qui devient dans notre cas :

```
#include "GameOfLife.h"  // Inclure le fichier .h

// Constructeur
GameOfLife::GameOfLife(int numLines, int numCols) {}

// Initialisation du jeu
void GameOfLife::initialisation() {}

// Mise à jour du statut des cellules lors d'une itération
void GameOfLife::play() {}

// Calcule le nombre de voisins à (i,j) qui sont vivants
int GameOfLife::nbAlivedNeighbours(int i, int j) { return 0; }

// Écrit un fichier avec la solution au format Paraview
void GameOfLife::saveSolution(int it) {}
```

Une fois que le fichier .h est inclus dans le main.cc (ne pas ajouter le .cpp), compiler le code :

```
g++ -std=c++11 -o run GameOfLife.cpp main.cc
```

et ensuite l'exécuter. Cependant pour le moment le code ne fait rien puisque les fonctions ne sont pas implémentées. Regarder la structure de la construction de votre première classe.

3. Implémenter les fonctions dans le .cpp. Ne pas oublier d'ajouter les *include* qui sont nécessaires.

Quelques conseils/astuces

a) *Constructeur :* Le constructeur a pour rôle d'initialiser les attributs de votre classe. Ici la classe a 3 attributs : *_numLines*, *_numCols* et *_grid* (pour ce dernier comme l'étape d'initialisation est séparée il s'agit de fixer les tailles du vecteur de vecteur de int). L'underscore devant les 3 attributs permet de ne pas oublier que ce ne sont pas des variables classiques. Il y a deux manières de faire qui sont données dans le code ci-dessous :

```
// Dans cet exemple :
// La valeur du 1er des attributs est donnée à travers le constructeur.
// Le 2ème est fixé directement (on voit qu'il s'agit d'un string)
// Le 3ème est un vecteur vector<type3> dont la taille est fixée à 10.
// 1ère stratégie
MaClasse::MaClasse(type1 val_attribut1)
{
   _attribut1 = val_attribut1;
   _attribut2 = "mon_resultat";
   _attribut3.resize(10);
}
// 2ème stratégie
MaClasse::MaClasse(type1 val_attribut1) :
_attribut1(val_attribut1), _attribut2("mon_resultat"), _attribut3(10)
{
}
```

Pour comprendre comment fixer les tailles du vecteur de vecteur de *int*, tester le bout de code suivant :

```
vector<vector<int> > A(3,vector<int>(3,1));
cout << "A = " << endl;
for (int i = 0; i < 3; i++)
{
  for (int j = 0; j < 3; j++)
    cout << A[i][j] << " ";
  cout << endl;
}
```

À présent, tester le code en remplaçant la première ligne par :

```
vector<vector<int> > A;
A.resize(3,vector<int>(3,-2));
```

b) *initialisation :* Le clown est un bon exemple pour tester votre code puisque la solution à l'itération 110 vous est donnée. Initialiser une grille de 50×50 ainsi :

```
_grid[21][24]=1; _grid[21][26]=1;
_grid[21][25]=1; _grid[19][26]=1;
_grid[20][24]=1; _grid[20][26]=1;
_grid[19][24]=1;
```

c) *play :* Attention à ne pas oublier de créer une grille temporaire qui conserve les résultats de l'itération précédente.

d) *nbAlivedNeighbours :* Une autre façon de créer un vecteur dont on connait déjà toutes les entrées est la suivante :

```
int neigh_x[] = {i-1, i , i+1, i-1, i+1, i-1, i , i+1};
```

e) *saveSolution :* On souhaite sauvegarder la solution à chaque itération dans un fichier dont le nom sera "sol_it.vtk" pour pouvoir l'afficher au format `Paraview`. L'entête d'un fichier `Paraview` pour une grille est le suivant (en gris tout ce qui est fixe, en rouge tout ce qui doit être adapté) :
vtk DataFile Version 3.0
cell
ASCII
DATASET STRUCTURED_POINTS
DIMENSIONS nbColonnes nbLignes 1
ORIGIN x y z
SPACING spx spy spz
POINT_DATA nbColonnes×nbLignes
SCALARS cell float
LOOKUP_TABLE default

Ensuite les valeurs stockées dans la grille peuvent être écrites ligne par ligne.

4. Vérifier que le résultat obtenu à l'itération 110 est correct. Pour afficher le résultat dans `Paraview`. Il faut ouvrir le logiciel et charger l'ensemble des fichiers solutions en même temps. Afin d'éviter de surcharger le dossier où le code se situe, nous allons créer un dossier de résultats. Pour cela définir (dans le .h) un attribut

```
std::string _results;
```

qui sera défini dans le constructeur avec le nom souhaité. Ensuite utiliser la commande *system* pour construire ce dossier :

```
system(("mkdir -p ./" + _results).c_str());
```

et enfin modifier le chemin des fichiers contenant votre solution.

5. Nous souhaitons donner la possibilité à l'utilisateur de choisir entre :

- Une initialisation par défaut, par exemple une initialisation aléatoire en utilisant :

```
int a = (rand() % 2); // a vaut aléatoirement soit 0 soit 1
```

- L'initialisation "Clown"
- L'initialisation "Canon" (donnée ci-dessous pour une grille de 50×50)

```
_grid[20][ 6]=1; _grid[23][21]=1; _grid[21][30]=1; _grid[37][43]=1;
_grid[21][ 6]=1; _grid[20][22]=1; _grid[22][30]=1; _grid[38][43]=1;
_grid[20][ 7]=1; _grid[21][22]=1; _grid[18][40]=1; _grid[40][43]=1;
_grid[21][ 7]=1; _grid[22][22]=1; _grid[19][40]=1; _grid[41][43]=1;
_grid[20][16]=1; _grid[21][23]=1; _grid[18][41]=1; _grid[42][43]=1;
_grid[21][16]=1; _grid[18][26]=1; _grid[19][41]=1; _grid[38][44]=1;
_grid[22][16]=1; _grid[19][26]=1; _grid[37][40]=1; _grid[40][44]=1;
_grid[19][17]=1; _grid[20][26]=1; _grid[38][40]=1; _grid[38][45]=1;
_grid[23][17]=1; _grid[18][27]=1; _grid[40][40]=1; _grid[40][45]=1;
_grid[18][18]=1; _grid[19][27]=1; _grid[37][41]=1; _grid[39][46]=1;
_grid[24][18]=1; _grid[20][27]=1; _grid[38][41]=1; _grid[19][21]=1;
_grid[18][19]=1; _grid[17][28]=1; _grid[40][41]=1; _grid[17][30]=1;
_grid[24][19]=1; _grid[21][28]=1; _grid[41][41]=1; _grid[43][42]=1;
_grid[21][20]=1; _grid[16][30]=1; _grid[42][41]=1;
```

Pour cela, **ajouter un deuxième constructeur** qui permet de récupérer le choix de l'utilisateur, dans le cas où il souhaite choisir :

```
// Premier constructeur (celui déjà  créé)
GameOfLife:GameOfLife(int numLines, int numCols)
: _numLines(numLines), _numCols(numCols), _results("Results")
{ ... };
// Deuxième constructeur
GameOfLife:GameOfLife(int numLines, int numCols, std::string userChoice)
: _numLines(numLines), _numCols(numCols), _results(userChoice)
{ ... };
```

Construire ce nouveau constructeur et adapter votre initialisation :

```
void GameOfLife::initialisation() {
   // Dossier pour mettre les résultats (dont le nom dépend du choix)
   system(("mkdir -p ./" + _results).c_str());
   if (_results == "Clown") // Clown
   { ... }
   else if (_results == "Canon") // Canon
   { ... }
   else // Initialisation par défaut !
   { ... }
}
```

Tester le code en modifiant le main.cc :

```
int iterationNb(110);
GameOfLife game(50, 50, "Clown");
game.initialisation();
game.saveSolution(0);
for(int i=1; i<=iterationNb; i++) {
   game.play();
   game.saveSolution(i);
}
```

6. Implémenter le code permettant de demander à l'utilisateur ce qu'il souhaite afin d'obtenir ce genre d'interaction dans la console :

```
[gameoflife] $ ./run
Choississez votre initialisation :
1) Initialisation par défaut
2) Initialisation Clown
3) Initialisation Canon
Votre réponse (1, 2 or 3) :
```

Ensuite implémenter un "switch" sur le choix de l'utilisateur. La difficulté réside dans le fait que vous souhaitez utiliser un constructeur différent suivant le choix de l'utilisateur, c'est-à-dire :

```
switch(userChoice)
{
case 1:
   GameOfLife game(50, 50);
   break;
case 2:
   GameOfLife game(50,50,"Clown");
   break;
case 3:
   GameOfLife game(50,50,"Canon");
   break;
default:
   cout << "Ce choix n'est pas possible." << endl;
   exit(0);
}
game.initialisation();
```

La compilation de ce code donne l'erreur suivante :

```
[gameoflife] $ g++ -std=c++11 -o run GameOfLife.cpp main.cc
main.cc:25:16: error: redefinition of 'game'
          GameOfLife game(50,50,"Clown");
                     ^
main.cc:22:16: note: previous definition is here
          GameOfLife game(50, 50);
```

En effet il n'est pas possible de définir deux fois l'objet "game" ! C'est là que toute l'utilité des **pointeurs** et leur **allocation dynamique** apparaît. Créer un pointeur de la classe GameOfLife et l'initialiser dans le "switch". Pour allouer dynamiquement la mémoire utiliser "new". Ne pas oublier que toute allocation dynamique par "new" doit être impérativement et explicitement rendue par le développeur avec "delete" (TP 2 Ex 5 Q 2).
Dernière remarque : quand votre objet est défini par un pointeur, l'appel d'une méthode se fait différemment :

```
game->initialisation();
```

7. Proposer à l'utilisateur de définir le nombre d'itérations souhaité sauf dans le cas du "Clown" où le nombre reste fixé à 110.