Université Savoie Mont Blanc,
UFR SceM
INFO702, M1 INFO

September 18, 2023

Warm-up C++, POO

(Passage par référence, classes, surcharge, polymorphisme dynamique)

Question 1. Utilisation des références

Réécrivez le code suivant en utilisant au maximum des références. Réécrivez ensuite le code en C++ objet, avec constructeur et operator+=.

```
struct Point {
  double x;
  double y;
};

struct Vecteur {
  double vx;
  double vy;
};

void cree( Point* ptr_p, double x0,
    double y0 )
{
  ptr_p->x = x0;
  ptr_p->y = y0;
}
void translate( Point* ptr_p, Vecteur v )
{
  ptr_p->x += v.vx;
  ptr_p->y += v.vy;
}

int main()
{
  Point p0;
  Vecteur v0 = { 1.0, 0.0 };
  creer( &p0, 4.0, 1.0 );
  translate( &p0, v0 );
}
```

Question 2. Classes ; surcharge d'opérateurs

Nous avons vu en cours comment faire une classe `Vecteur`, équipée de différents opérateurs. Nous allons faire une classe compagnon `Matrice`, qui représente une matrice carrée 2×2.

$$M := \begin{pmatrix} m_{00} & m_{01} \\ m_{10} & m_{11} \end{pmatrix}.$$

On veut pouvoir faire les opérations suivantes avec nos `Matrices`.

- *attributs*: Choisissez les attributs/données membres nécessaires
- *construction*: Initialiser une matrice à zéro, ou avec 4 coefficients, ou à l'aide de deux vecteurs colonnes.
- *accesseurs*: Accès en lecture/écriture aux coefficients de la matrice. On précisera en paramètres la ligne (0 ou 1) et la colonne (0 ou 1). On surchargera l'opérateur `operator()` en lecture et écriture.
- *opérations arithmétiques de matrices*: on peut multiplier une matrice par un double, additionner deux matrices, multiplier deux matrices. On rappelle que si A et B sont deux matrices de taille $n \times n$, et $P = A * B$, alors le coefficient en ligne i et colonne j de P vaut:

$$p_{ij} = \sum_{k=0\ldots n-1} a_{ik} * b_{kj} \quad \text{(ici } n = 2\text{, donc)} \quad = a_{i0} * b_{0j} + a_{i1} * b_{1j}$$

- *opérations matrices vecteur*: on peut multiplier une matrice par un `Vecteur`, ce qui retourne un `Vecteur`

- *matrices de rotation*: les matrices de rotation sont des matrices particulières (leur déterminant vaut 1) dont une propriété notable est qu'elles peuvent tourner les vecteurs dans le plan. On les forme ainsi, si θ est l'angle de rotation voulu:

$$R_\theta := \begin{pmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{pmatrix}.$$

  Ecrire une méthode *statique* `rotation( double theta )` qui retourne la matrice R_θ.

  On peut maintenant tourner un `Vecteur` de $\pi/4$ radians (45) ainsi:

```
Vecteur v( 3.0, 1.0 );
Matrice R45 = Matrice::rotation( M_PI / 4.0 );
Vecteur w = R45 * v;
```

- *déterminant, inverse*: le déterminant d'une matrice carré est important dans de nombreuses applications en calcul scientifique. Ecrire une méthode `det()` qui le calcule. Ecrire ensuite une méthode `inverse()`.

Question 3. Polymorphisme dynamique : jeu de dames

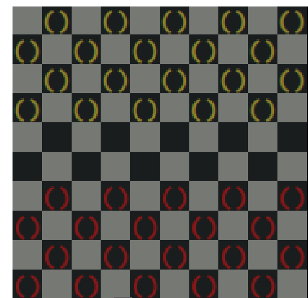
On voudrait modéliser un jeu de dames, composé d'un plateau de 10x10 cases, alternant noir et blanc, de pions de deux couleurs, et éventuellement de dames (2 pions l'un sur l'autre).

Chaque `Case` est donc noire ou blanche. Au maximum, elle a une `Piece` (abstraite) dessus (concrètement un `Pion` ou une `Dame`).

```
class Case {
  bool m_color; //< noir (false) ou
      blanc
  // Pointeur vers pièce dessus ou
      nullptr
  Piece* m_piece;
public:
  Case( bool color );
  void change( Piece* p );
  void affiche() const;
};
struct Damier {
  Case cases[10][10];
  Damier(); // initialise les cases
  void initJoueur( int j ); // et le
      joueur
  void affiche() const;
};
```

```
struct Piece {
  virtual void affiche() const = 0;
};
class Pion : public Piece {
  int m_joueur;
public:
  Pion( int joueur ); // Joueur 0 ou 1
  virtual void affiche() const override;
};
struct Dame : public Piece {
  Dame( int joueur ); // Joueur 0 ou 1
  virtual void affiche() const override;
};
```

On rappelle la fonction `console` qui permet de faire des affichages en couleur dans le terminal (`https://stackoverflow.com/questions/2616906/how-do-i-output-coloured-text-to-a-linux-terminal`)



```
string console( string code )
{ return "\033[" + code + "m"; }
auto normal      = console("0"); // revient à normal
auto clignotant  = console("5"); // clignotant
auto texte_rouge = console("1;31"); // rouge gras
auto texte_jaune = console("1;33"); // jaune gras
auto fond_noir   = console("40");   // fond noir
auto fond_gris   = console("47");   // fond gris
```

Complétez le code ci-desssus (les corps des méthodes) pour que l'on puisse créer un `Damier` et le visualiser ainsi. Un pion sera représenté par '()' gras, une dame par '@@' gras et clignotant.

```
...
Damier D;
D.initJoueur( 0 );
D.initJoueur( 1 );
D.affiche();
```

Question 4. Foncteur et polymorphisme dynamique : intégrale d'une fonction

On veut définir une fonction (C) dont l'objectif est d'estimer l'intégrale d'une fonction (mathématique de $\mathbb{R}$ dans $\mathbb{R}$) dans un intervale donné $[a, b]$. L'utilisateur fournit la fonction f (en tant qu'objet), les bornes a, b ainsi que le nombre de points n où on prend la valeur de la fonction. On rappelle que l'on peut estimer l'intégrale d'une fonction (ou l'aire sous la fonction si vous préférez) avec la formule suivante (dite des trapèzes):

$$\int_a^b f(x)dx \approx \frac{b-a}{n} \left[\frac{f(a)+f(b)}{2} + \sum_{i=1}^{n-1} f\left(a + i\frac{b-a}{n}\right) \right]. \tag{1}$$

- Définir d'abord une classe abstraite `Fonction`, qui impose que ses sous-classes aient une méthode `double operator()( double x )`.
- Ecrire ensuite la fonction `Integrer` qui prend en paramètres une fonction f, les bornes a, b et l'entier n. Peut-on passer la fonction f par valeur ? Peut-on la passer par référence, par pointeur ? Quelle est le passage le plus naturel ?
- On vous donne la fonction concrète ci-dessous.

```
struct Parabole : public Fonction {
  Parabole( double _c, double _d, double _e ) : c(_c), d(_d), e(_e) {}
  virtual double operator()( double x ) const
  { // cx^2+dx+e
    return c*x*x+d*x+e;
  }
  double c,d,e;
};
```

  Comment écrire le programme principal pour qu'il estime l'intégrale de $1 - x^2$ entre -1 et 1 avec 100 points.
- Si vous mettez 1000 points, combien de chiffres en plus seront justes ? 10000 ?, ...

Question 5. Polymorphisme dynamique : formes et détection de collision

On va regarder un algorithme "probabiliste" pour savoir si deux formes s'intersectent. L'idée est de ne tester que quelques points aléatoirement sur une des formes. Pour chacun, on regarde s'il appartient à l'autre forme. Si oui, on est sûr que les formes s'intersectent (on a détecté une *collision*), sinon on n'est sûr de rien (!) mais plus on tirera de points aléatoirement, plus on sera "sûr". Ce genre d'algorithme est utilisé dans les cas suivants:

- les formes que l'on compare ne sont pas "simples" (au sens où on pourrait trouver des solutions analytiques à ce problème),
- les formes que l'on compare sont variées (on ne peut pas tester précisément tous les cas possibles d'intersection),
- on ne dispose pas forcément d'un budget de calcul illimité, mais seulement un certain temps (genre dans un jeu temps-réel)
- ce n'est pas "grave" si on se trompe (genre dans un jeu temps-réel)

On note que ce type d'algorithme ne se trompe que dans un sens.

Un inconvénient (il y en a toujours) est qu'il faut que chaque forme fournisse quelques services:

- une méthode qui dise si oui ou non un `Point` est à l'intérieur de cette forme
- une méthode qui retourne un nouveau `Point` aléatoire dans la forme. Bien sûr, il vaut mieux que le tirage aléatoire soit le plus uniforme possible pour que l'algorithme maximise ses chances de détecter la collision.
- (optionnel) une méthode qui retourne la boîte englobante de la forme.

1. Proposer une classe abstraite `Forme` qui impose les deux opérations précédentes à toute sous-classe.

2. Ecrire la forme concrète `Rectangle`, qui définit un rectangle aligné avec les axes entre un point bas p et un point haut q.

   On utilisera la fonction ci-dessous qui retourne un nombre aléatoire uniforme dans l'intervalle $[0, 1]$.

```
double rand01() {
  return random() / RAND_MAX; // BSD system, rand() for POSIX systems.
}
```

3. On pourrait écrire aussi une forme concrète `Disque` sans trop de difficulté, là encore en la centrant en un point c, et en donnant un rayon r.

4. Ecrire maintenant la fonction `checkOnePoint( const Forme & f1, const Forme & f2 )` qui teste si un point de `f1` appartient à `f2`. Ecrire ensuite la fonction `checkManyPoints( const Forme & f1, const Forme & f2, int n )` qui teste alternativement si `n` points de `f1` appartiennent à `f2` et inversement.

   Pourquoi alterner entre les tests ? Quel est la probabilité que notre algoritme fonctionne ?

5. On peut ensuite enrichir considérablement nos formes en introduisant des transformations du plan, des groupages, des intersections, ce qui permet de construire des formes très complexes.

   Ecrire une classe `FormeDeplacee`, qui est une `Forme`, et qui référence une autre `Forme`, et qui fait la translation d'une forme par un vecteur donné à la construction (u, v).

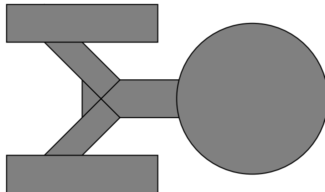
   Ecrire une classe `FormeTournee`, qui est une `Forme`, et qui référence une autre `Forme`, et qui fait la rotation d'une forme par un angle donné à la construction r.

   Ecrire une classe `FormeGroupee`, qui est une `Forme`, et qui référence deux autres formes, et qui représentre l'union des deux formes.

   Ecrire une classe `FormeIntersectee`, qui est une `Forme`, et qui référence deux autres formes, et qui représentre l'intersection des deux formes.

6. On voit qu'on a maintenant le moyen de créer des formes très complexes en combinant ces objets très simples. L'algorithme de détection de collision n'a finalement pas changé du tout !!

   Créer la situation suivante.

7. Ecrivez une fonction qui calcule une boîte englobante approximative de n'importe quelle forme.

8. Ecrire maintenant une fonction qui détecte les collisions entre une forme et un tableau de formes, et qui retourne soit -1, soit le numéro d'une forme touchée.

9. Pour aller plus loin. Tester juste un point n'est pas toujours efficace. En effet on ne recouvre pas de grandes zones. On propose donc l'amélioration suivante.

   - La méthode `randomPoint()` retourne un couple Point/double (p, r), qui est tel que le disque (p, r) est inclus dans la forme.
   - La méthode `isInside` devient une méthode `checkCollision( Point p, double r)`, qui teste si le disque (p, r) touche la forme.

   Faites évoluer vos définitions concrètes de ces méthodes dans les différentes classes concrètes. Le code ressemble beaucoup au précédent, mais les tests sont nettement plus efficaces.