Cours 06
Modélisation et Programmation

Constructeurs et opérateurs

16 mars 2022

Organisation du cours

1. Exemple jouet

   Présentation d'une classe illustrative de vecteur 2D

2. Constructeurs

   Étude de la copie de cette classe

3. Opérateurs

   Affichage, égalité et combinaisons linéaires

Exemple jouet

Classe Vecteur « perso »

```
1  class Vector2D {
2    public:
3      double* v;
4      /** Initialisation (par défaut : que des zéros) */
5      Vector2D(double x=0, double y=0) {
6        v = new double[2]{x,y};
7  };
8      ~Vector2D() { delete[] v; };
9  };
10 double norm(const Vector2D &vec) {
11   return sqrt(a.v[0] * a.v[0] + a.v[1] * a.v[1]);
12 }
```

On aimerait pouvoir faire des opérations standards avec ce vecteur,

- Définir un vecteur renormalisé
- Afficher ce vecteur avec `cout << vec << endl;`

Échec de normalisation

Définition un vecteur normalisé :

```
Vector2D a(2,1);
double r = 1.0 / norm(a);
Vector2D n(a.v[0] * r, a.v[1] * r);
```

Impossible si `v` est privé ! Et c'est maladroit.

Dans ce cours on va voir comment autoriser l'écriture suivante :

```
Vector2D a(2,1);
Vector2D n = a / norm(a);
```

- Quelles sont les opérations en jeu dans ces quelques lignes ?

Analyse des opérations

```
Vector2D n = a / norm(a);
```

1. Définition d'un `Vector2D` avec `a / norm(a)`
   - Besoin de savoir diviser un `Vector2D` par un `double`
   - Définition d'une méthode d'opérateur

     ```
     Vector2D Vector2D::operator/(const double &)
     ```

2. Copie de ce nouveau `Vector2D` dans `n`
   - Besoin de savoir copier un `Vector2D`
   - Définition d'un nouveau constructeur

     ```
     Vector2D::Vector2D(const Vector2D &);
     ```

Rq : Pour afficher un vecteur avec `cout << vec`, on définit

```
ostream &operator<<(ostream&, const Vector2D &)
```

Constructeurs

Copie naïve d'un Vector2D

Faisons par exemple un code de base qui « normalise » `a`

```
Vector2D n = a;
n.v[0] *= r; n.v[1] *= r;

cout << "Vecteur d'origine : ";
cout << a.v[0] << " " << a.v[1] << endl;
cout << "Vecteur normalisé : ";
cout << n.v[0] << " " << n.v[1] << endl;
```

```
└» ƒ ./a.out
Vecteur d'origine : 0.894427 0.447214
Vecteur normalisé : 0.894427 0.447214
a.out(32278,0x11014b600) malloc: *** error for object 0x600003d
08000: pointer being freed was not allocated
a.out(32278,0x11014b600) malloc: *** set a breakpoint in malloc
_error_break to debug
```

- Le vecteur `a` est modifié
- Le code plante

Copie maline d'un Vector2D

Dans cet exemple, `a.v` et `n.v` pointent à la même adresse !

- Lors de la copie, il faut allouer un nouveau pointeur.
- Si une nouvelle allocation n'est pas souhaitable, on sécurise la destruction avec un `shared_ptr` [1]

On définit alors le constructeur par copie

```
Vector2D::Vector2D(const Vector2D &a) {
  v = new double[2]{a.v[0], a.v[1]};
}
```

(le constructeur par copie existe par défaut, mais n'est pas adapté dans ce cas !)

```
└» ƒ ./a.out
Vecteur d'origine : 2 1
Vecteur normalisé : 0.894427 0.447214
```

1. https://en.cppreference.com/w/cpp/memory

À retenir.....

Rule of Three

If you need to explicitly declare either the destructor, copy constructor or copy assignment operator yourself, you probably need to explicitly declare all three of them.

Optionnel : la “Rule of Five” contient aussi le déplacement mémoire

- Est-ce que cela s’applique aux classes `cDistribution` ?

Corollaire (Rule of Zero) **:** Tant que possible, on essaie de construire des classes qui n’ont pas besoin de ces défintions explicites.

Pour cela, on utilise des objets standards de type `shared_ptr` , `vector` , `list` , etc.

- Mais c’est quoi au juste ce “copy assignment operator” ?

Opérateurs

Objectif syntaxique

Pourquoi devrait-on faire `Print(v)` plutôt que `cout << v` ?

Pourquoi ces codes donnent des résultats différents, maintenant ?

`Vector2D n = a;` vs `Vector2D n; n = a;`

- Il faut surcharger des opérateurs, explicitement

```
ostream &operator<<(ostream &, const Vector2D &)
Vector2D &Vector2D::operator=(const Vector2D &)
```

NB : L'opérateur `=` est une méthode de `Vector2D` tandis que `<<` est une fonction `friend`.

Rq : Au-delà de l'avantage syntaxique, l'opérateur `<<` permet aussi de verser les données dans un fichier !

La surcharge de Vector2D

Opérateur de copie :

```
Vector2D &Vector2D::operator=(const Vector2D &a){
  delete[] v;
  v = new double[2]{ a.v[0], a.v[1] };
  return *this;
}
```

NB : Contrairement au constructeur par copie, il y a un `return`.

Opérateur de flux :

```
ostream &operator<<(ostream &out, const Vector2D &a){
  out << a.v[0] << " " << a.v[1];
  return out;
}
```

- Pour chaque opérateur qu'on souhaite surcharger, il faut trouver la syntaxe spécifique en fonction de l'objectif.

Adaptation aux classes cDistribution