PG208
Programmation Orientée Objet / Langage C++
TP3

Y. Bornat A. Valade

April 8, 2024

1 La programmation orienté objet

1.1 Principes

Au fil des différents projets de programmation que vous avez dû mener, vous vous êtes certainement rendu compte que bien souvent, la vrai difficulté n'est pas le codage ou l'algorithme, mais la structure de données qui permettra de résoudre le problème de façon *efficace*. Derrière le mot *efficace*, vous pouvez utiliser le critère de votre choix :

- efficacité energétique (si vous êtes le client)
- rapidité de codage (si vous êtes le patron)
- réduction des bugs (si vous êtes le mainteneur)
- minimisation de l'effort intellectuel (pour ceux qui se reconnaitront par cette ligne)

Le principe de la Programmation Orientée Objet (POO) est de centrer la structure du programme sur la façon dont les données sont stockées et exploitées et non sur les fonctions. Evidemment, Une fois les données dans une structure, il faut également fournir une façon efficace de les exploiter, c'est pour cela que nous avons passé du temps au TP2 sur les fonctions et leur surcharges.

Un *objet* sera donc un `struct` ou équivalent, pour lequel tout un nombre de fonctions auront été écrites. La différence fondamentale avec le C est que, en C, on utilise un un `struct` en accédant aux champs qui le constituent. Pour rappel, si on a un `struct color`, on utilisera ses composantes `.red`, `.green` et `.blue` directement. En C++, et dans la programmation objet en général, cette approche est *fortement découragée*. On proposera plutôt des fonctions qui permettent d'accéder à ces valeurs. Donc `.red()`, `.green()` et `.blue()`. Cela signifie que les composantes seront stockées dans des variables ayant des noms différents. L'intérêt de cette approche est que si, pour une raison ou une autre, on désire coder une couleur différemment (luminance, chrominance et saturation par exemple), la classe sera mise à jour, mais pas la façon de s'en servir. C'est un principe de base : La façon d'utiliser les données doit être indépendante de la façon dont elles sont stockées.

2 just Go !

Dans ce TP, le prétexte que nous allons utiliser pour faire notre objet est la représentation de polynômes.

2.1 Création de l'objet

On commence tout doux, on va créer notre objet comme un struct. Le principe est de stocker un polynôme comme un vecteur de coefficients. On obtiendra donc le code suivant:

```cpp
#include <vector>

struct Polynomial {
    std::vector<double> m_coeffs;
};
```

Pour un programmeur en C, cette déclaration semble d'un intérêt limité car il n'est pas forcément nécéssaire de déclarer un nouveau type de données. Il suffirait de dire qu'on représente un polynôme par un vecteur, et tout est réglé. Mais pour faire des opérations sur les polynômes, un programmeur en C va faire un ensemble de fonctions (`add_poly()`, `mul_poly()`, `div_poly()` ...) ce qui va complexifier la syntaxe. Un programmeur *objet*, va donc créer son `struct Polynomial` et surcharger `operator+`, `operator-`, ... En effet, le comportement de ces fonctions est très différent pour les vecteurs et les polynômes. L'addition se fait en général élément à élément pour un vecteur, et donc impose des vecteurs de même taille. Alors que l'addition de polynômes n'impose absolument pas d'avoir des termes de degré équivalents.

Remarque : Le mot clé `typedef` est redondant en C++. Depuis C++11[1], la définition du `struct polynomial` telle qu'écrite au dessus crée automatiquement pour nous le type de même nom. Dans le texte, nous continuerons de parler de `struct Polynomial` pour bien faire la différence avec une autre notion vue prochainement dans le TP.

2.2 Création d'une fonction spécifique

Il y a beaucoup de choses à créer pour que `struct Polynomial` soit exploitable. Nous allons d'abord voir comment créer les fonctions spécifiques. La première d'entre-elles sera une fonction qui donne le degré du polynôme. Pour réellement *faire de l'objet*, nous allons faire en sorte que cette fonction soit accessible sous la forme `poly.degree()` ou `poly` est un `struct Polynomial`. Pour cela, la syntaxe est :

```cpp
#include <vector>

struct Polynomial {
    std::vector<double> m_coeffs;
    int degree();
};

int Polynomial::degree() {
    return m_coeffs.size() - 1;
}
```

Pour décoder : il faut déclarer la fonction `int degree()` à l'intérieur de `Polynomial`. Pour définir la fonction, celà se fait à l'extérieur, mais en précisant que la fonction fait partie du nom de domaine `struct Polynomial`, on fait le lien avec la déclaration. Cela permet aussi à la fonction d'avoir accès à l'ensemble des champs déclarés dans `Polynomial`.

2.3 Exercice

Créez la fonction `.coeffs()` qui renvoie un `std::vector` contenant l'ensemble des coefficients du `struct Polynomial`.

2.4 Accès au contenu

Pourquoi une fonction `.coeffs()` alors qu'il suffit d'accéder à `m_coeffs` ? Rappelez-vous que l'intérêt de la programmation objet est de dissocier la façon de manipuler les données de la façon de les stocker. On doit pouvoir changer la façon de mémoriser un polynôme sans changer la façon d'accéder à ses coefficients. D'où l'intérêt de dissocier la variable qui contient les coefficients et la fonction qui permet d'y accéder.

Pour cela, on va même plus loin en définissant ce qu'un utilisateur a le droit de manipuler et ce qui lui est interdit :

[1] Pour les plus curieux et curieuses, sous Linux vous compilez par défaut en C++17, donc l'option est valide aussi. Pour les utilisateurs Windows et Mac, il faut spécifier le standard avec l'option de compilation `-std=c++11`, `-std=c++14` ou `-std=c++17`. De manière générale, le plus simple est de compiler en C++17.

```
struct Polynomial {
public:
    int                 degree();
    std::vector<double> coeffs();
private:
    std::vector<double> m_coeffs;
};
```

En déclarant `m_coeffs` comme *privé*, on précise que les fonctions appartenant à `struct Polynomial` peuvent le manipuler, mais pas les autres fonctions. Cela protège donc la manipulation des mécanismes internes. La règle est que les changements ou mises à jour d'un objet ne doivent *jamais* modifier le comportement de tout ce qui est public.

Cette séparation permet deux comportements essentiels :

- L'utilisateur ne sera pas affecté en cas de changement de fonctionnement interne de la structure.
- Les données privées de la structure sont protégées de toutes valeurs incohérentes que l'utilisateur pourrait provoquer.

Un petit inconvenient pourtant : si `m_coeffs` est privé, il n'est plus possible d'initialiser la structure. Nous verrons à la prochaine section comment le faire proprement. Pour l'instant, il n'y a pas d'autre choix que d'ajouter une fonction `.set_coeffs` pour définir les coefficients.

2.5 Un peu de vocabulaire

Jusqu'ici, le vocabulaire a été flou et imprécis pour présenter les concepts avant les mots qui les identifient. Nous avons maintenant suffisamment avancé. Posons des mots pour bien se comprendre les uns les autres.

2.5.1 Classe

Une classe est le type d'un *objet*. Dans le cas de cette séance, la classe que nous utilisons est donc `struct Polynomial`. Comme la programmation objet utilise abondamment les classes, que l'usage de `struct` est lourd, et que la syntaxe de `typedef` est à oublier, on peut simplement utiliser le mot-clé `class` pour déclarer une classe.

La différence principale entre un `struct` et une `class` est que lorsqu'un objet est déclaré en tant que `struct`, tout son contenu est public par défaut, s'il est décaré en tant que `class`, il est privé par défaut. L'usage est d'utiliser `struct` quand il y a peu de fonctions associées, et `class` quand il y a tout un écosystème de définitions autour.

Par convention, le nom d'une classe commence toujours par une majuscule.

2.5.2 Instance

Pour faire court, c'est une variable dont le type est une classe. Concrètement, lorsque vous déclarez `std::vector<int> vec`, `std::vector` est une classe et `vec` est une instance de la classe `std::vector`.

2.5.3 Méthode

Une méthode est une fonction associée à une classe. Ainsi, pour notre classe `Polynomial`, on ne mentionnera plus les fonctions `.degree()` ou `.coeffs()`, mais les *méthodes* `degree` ou `coeffs`.

2.5.4 Attribut

Il s'agit d'une variable propre à une classe. dans le cas de `Polynomial`, `m_coeffs` est un attribut. Par convention, tous les attributs d'une classe sont privés.

2.5.5 Membre

Un membre d'une classe est quelquechose qui est déclaré dans sa définition. Ainsi, les Méthodes et les attributs d'une classe sont ses membres.

2.5.6 *this* is my name

Pour accéder aux membres d'une classe à l'intérieur de celle-ci, il suffit simplement de les appeler par leur nom. C'est comme cela que fonctionne les méthodes `coeffs` ou `degree`. C'est assez pratique mais, nous le verrons plus tard, cela peut parfois porter à confusion. De plus, une classe à parfois besoin d'accéder à sa zone mémoire. Pour répondre à ce besoin, chaque méthode a accès au mot-clé `this` qui est un pointeur constant vers l'instance à laquelle la méthode appartient. Donc si on veut être plus explicite, on peut réécrire `degree` comme suit:

```
int Polynomial::degree() {
    return (*this).m_coeffs.size() - 1;
}
```

Les habitués pourront faire la remarque que `this->m_coeffs` est une syntaxe plus agréable.

Nous verrons plus tard, avec les opérateurs, que c'est très utile pour qu'une méthode puisse renvoyer l'instance à qui elle appartient. Ça se fait simplement avec `return *this`.

3 Initialisation d'une instance

A ce stade, nous sommes face à un problème : comment l'utilisateur peut-il créer une instance de `Polynomial` alors qu'il n'a pas accès à l'attribut `m_coeffs` ? On pourrait bien sûr ajouter la méthode `set_coeffs` pour les définir. Mais il y a une technique plus subtile : l'écriture d'un constructeur.

3.0.1 Constructeur de classe

Le constructeur d'une classe est la méthode qui permet, à partir d'arguments déjà connus, de créer la structure de données de la classe. Dans notre cas, le constructeur de `Polynomial` doit récupérer les coefficients de notre polynôme et les stocker dans `m_coeffs`. Il porte le nom de la classe dont il est le constructeur. On le définira donc de la sorte :

```
class Polynomial {
    public:
        Polynomial(std::vector<double>& coeffs);
        int degree();
    private:
        std::vector<double> m_coeffs;
};

Polynomial::Polynomial(std::vector<double> & coeffs) {
    m_coeffs = coeffs;
}
```

Maintenant, il est possible de créer un polynôme de toutes pièces avec le code suivant :

```
Polynomial poly { {1, 2, 3} };
```

L'intérêt de l'initialisation uniforme est que le compilateur appelle directement le constructeur. Pour faire l'initialisation et l'affectation *à la main*, la syntaxe correspond à

```
Polynomial poly = Polynomial({1, 2, 3});
```

Cette deuxième syntaxe crée une nouvelle instance par l'appel à `Polynomial` puis l'affecte à la variable `poly`. Ce fonctionnement est découragé car il oblige la réservation de deux zones mémoire : une pour pour poly et une pour l'instance créée par le constructeur. Sur les très gros objets, cela peut réduire les performances.

L'intérêt du constructeur semble ici limité, mais il est essentiel. Si nous revenons aux TPs en C de l'année dernière, lors desquels il fallait faire une image, il était nécessaire d'initialiser l'image à l'aide d'un `malloc` pour réserver l'espace nécessaire au stockage des données pixel. Ce rôle est celui du constructeur.

3.0.2 Constructeur de copie

Ce qui est intéressant, c'est qu'il est possible de surcharger le constructeur. On peut par exemple créer un constructeur spécifique pour les polynômes de faible degré. Leur initialisation sera alors plus facile.

Si on définit le constructeur `Polynomial(double a, double b, double c);`, (et que l'on code la fonction), il devient très facile de créer un polynome de degré 2 grâce à la ligne `Polynomial id {0, 1, 0};`

Par défaut, le compilateur crée une surcharge appelée *constructeur de copie* qui permet de créer une instance différente, mais de même valeur. Ce comportement est très pratique, mais il a ses limites. Pour reprendre le cas de l'image du TP de C, un des attributs est un pointeur vers les données pixels. Le constructeur de copie va créer une nouvelle instance, avec une copie du pointeur, mais qui pointe vers les même données pixels, puisqu'il n'aura pas copié les données pointées. Il devient alors nécessaire de définir votre propre version du constructeur de copie.

3.0.3 Destructeur

De façon analogue, quand une instance n'est plus utilisée, il faut la détruire. Le compilateur gère cela très bien, sauf s'il y a une zone de données pointée. Il est en effet impossible de déterminer automatiquement s'il faut libérer ou pas les zones mémoires pointées. Il faut alors écrire un *destructeur* qui va libérer proprement la mémoire qui doit l'être. La définition d'un destructeur se fait à peu près comme celle d'un constructeur. deux différences : le nom, il commence par un `~`, les arguments, il n'y en a aucun. Ainsi, pour notre classe `Polynomial`, le destructeur sera

```
1  Polynomial::~Polynomial() {
2      std::cout << "Je suis un destructeur, Mouah ah ah ah !!!!" << std::endl;
3  }
```

Bien évidemment, il faut aussi déclarer le destructeur dans la description de la classe.

4 Surcharges classiques

4.1 Affichage

Lorsque l'on débute en objet, un des premières choses que l'on fait et d'afficher la valeur d'une instance pour débugger facilement. Pour cela, il y a deux stratégies : fournir une description exhaustive et techniquement correcte, ou fournir une description partielle, mais facile à lire. Pour être complet, nous allons voir les deux méthodes avec deux techniques différentes.

4.1.1 Implémentation d'une méthode de conversion

Le plus évident et de créer une méthode `to_string` qui ne prend pas d'argument. On se retrouve alors avec un code de ce genre :

```
1  std::string Polynomial::to_string() {
2      std::string s {"Polynomial({"};
3      for (double cf : m_coeffs) {
4          s += std::to_string(cf) + ", ";
5      }
6      s[s.size()-2] = '}';
7      s[s.size()-1] = ')';
8      return s;
9  }
```

4.1.2 Surcharge de l'opérateur de flux

Il est également possible de surcharger l'opérateur de flux sortant (`to_string`). Cet opérateur reçoit un flux et un objet en argument, et retourne un flux. Étant donné que l'opérateur (`<<`) n'est pas appelé comme une méthode classique, on le déclare comme une fonction libre (donc pas une méthode, mais une surcharge classique telle que vue au TP2). La surcharge suivante permettra donc un affichage direct :

```
1  std::ostream & operator<<(std::ostream & os, Polynomial & p) {
2      std::vector<double> c;
3      c = p.coeffs();
4      for (int i {p.degree()}; i>=1; i--) {
5          os << c[i] << "*pow(x," << i << ") + ";
6      }
7      os << c[0];
```

```
8         return os;
9     }
```

Le principe essentiel est de retourner le flux reçu comme premier argument. Ce dernier sera passé à l'élément suivant. En effet, la ligne `std::cout << a << b;` doit être lue comme `(std::cout << a) << b;`. Il est donc essentiel que `std::cout << a` retourne `std::cout` pour enchainer sur la sortie de `b`.

4.1.3 Surcharge de la fonction `std::to_string` (ou pas)

Il peut être tentant de surcharger la fonction `std::to_string`. Surtout quand on a fait le cowboy en fusionnant les espaces de nom. Mais c'est tout simplement impossible, précisément à cause des espaces de nom. La surcharge ne fera pas partie de la bibliothèque standard, et à ce titre, ne sera pas une vraie surcharge. La raison est que, pour être perenne, la surcharge doit être faite dans le même espace de noms que la fonction originale.

4.2 Application numérique

Cette partie est simplement le prétexte pour surcharger l'opérateur `()`. Rien de bien exceptionnel. Nous avons le choix entre déclarer une méthode dont la définition est `double operator()(double x);` ou une fonction libre dont la définition est `double operator()(Polynomial & p, double x);`. En choisissant la première solution, nous obtenons le code suivant :

```
1     double Polynomial::operator()(double x) {
2         double res {0};
3         for (std::size_t i{m_coeffs.size()}; i>0 ; i--) {
4             res = res*x + m_coeffs[i-1];
5         }
6         return res;
7     }
```

À condition bien sûr de déclarer la méthode dans la définition de classe.

4.3 Calcul formel

Il suffit de définir `operator+`, `operator-` et `operator*` en utilisant les polynômes comme argument. Ce qui est intéressant, c'est que quand ces opérateurs sont disponibles, il devient possible de créer un polynôme de façon réellement transparente. En effet, si on déclare `Polynomial x {{0 1}};`, il sera possible de définir un poylynôme avec une ligne de la forme `3*x*x + 2*x + 1`.

Bien sûr, si on surcharge `operator+`, `operator-` et `operator*`, il faut être logique et surcharger également `operator+=`, `operator-=` et `operator*=`. La différence est qu'on ne crée pas une nouvelle instance, mais on modifie l'instance passée en tant que premier argument. Si on déclare ces surcharges comme méthodes (ou fonctions membres), tout se passe bien. Mais si on les déclare comme fonctions libres, c'est le drame ! en effet, elles n'ont pas accès aux membres privés...

... sauf si elles sont *amies*.

5 Fonction *amies*

5.1 Definition

Une fonction *amie* d'une classe est définie comme une fonction libre, mais elle a accès à l'ensemble des membres (publics et privés) de la classe. Il faut donc bien retenir que cette fonction, bien que non membre, fait partie intégrante de la classe.

Dans le cas de la classe `Polynomial`, on la déclare comme à la ligne 13 du code suivant :

```
1     class Polynomial {
2         public:
3             Polynomial(std::vector<double> coeffs);
4             Polynomial(double c);
5             Polynomial(double b, double c);
6             Polynomial(double a, double b, double c);
7             ~Polynomial();
8
```

```
9             int degree();
10        private:
11            std::vector<double> m_coeffs;
12
13        friend std::ostream & operator<<(std::ostream & os, Polynomial & p);
14    };
```

5.2 Utilisation

Grâce aux fonctions *amies*, il est donc possible de définir ds fonctions libres qui ont accès à l'ensemble de la classe. L'intérêt est de pouvoir surcharger des fonctions ou opérateurs de façon transparente, ou, au contraire, de créer une fonction qui puisse facilement être surchargée. En effet, la surcharge ne doit pas être vue comme une façon d'occulter l'existant, mais de le compléter et de l'enrichir. De la même façon, il est possible de poser les bases d'un système par quelques classes, mais de laisser l'utilisateur l'enrichir grâce aux surcharges. Il est alors essentiel de laisser à l'utilisateur la liberté de faire evoluer les ressources.

5.3 Opérateurs : méthodes ou fonctions amies ?

5.3.1 Oui ! on a le choix

La très grosse majorité des opérateurs *unaires* peuvent être déclarés, au choix, comme une fonction membre[2] ou comme une fonction libre. La fonction membre ne prend pas d'argument, la fonction libre prend un argument dont le type est la classe (ou une référence vers la classe)

De même les opérateurs *binaires*, peuvent être déclarés comme fonction membre qui ne nécessite qu'un seul argument (le deuxième terme), ou une fonction libre qui reçoit deux arguments (les deux termes). Si le programme est cohérent, au moins un des deux termes sera la classe que nous sommes en train d'écrire.

Mais si les fonctions *amies* permettent d'accéder aux membres privés, comment faire le choix entre fonction libre ou fonction membre ?

5.3.2 La différence

Elle réside dans la permissivité que vous autorisez à l'usage. Si vous écrivez une fonction membre, vous imposez que le premier terme soit rigoureusement de la classe que vous avez créée. Si vous écrivez une fonction libre, vous permettez au compilateur de s'adapter à la situation grâce aux constructeurs disponibles.

Par exemple, si vous surchargez l'opérateur `+` dans une fonction membre de `Polynomial`, vous serez capable de calculer `p1+p2` où `p1` et `p2` sont des `Polynomial`, mais aussi `p1+a` où `a` est un `double` puisqu'il existe un constructeur à partir d'un `double`. Comme il existe un constructeur pour un `double` à partir d'un `int`, vous pouvez aussi calculer `p+i`. Par contre, il sera impossible de calculer `a*p`, parce que le premier terme doit nécessairement être un `Polynomial`.

Si l'opérateur `+` est surchargé dans une fonction libre, le compilateur sera capable de comprendre que pour `a+p`, il faut d'abord appeler le constructeur pour convertir `a` en `Polynomial`, puis appeler l'opérateur `+`. C'est généralement le comportement recherché car il conserve la commutativité de l'addition.

5.3.3 Usage

En C++, l'usage pour la surcharge d'un opérateur est de définir l'opérateur de modification (`+=`, `*=`, ...) comme fonction membre qui met à jour l'instance. Puis de définir l'opérateur binaire comme fonction libre qui crée une copie du premier terme, et appelle l'opérateur de modification sur la copie. Cela apporte toute la souplesse évoquée ci-dessus, sans avoir besoin de définir trop de fonctions amies.

[2] on ne parlera pas de méthode ici car on ne les appelle pas en mentionnant le nom de l'instance suivi du nom de la fonction séparés d'un point

6 Fonctions/méthodes constantes

Pour rappel, lors du passage des instances comme arguments ou résultat, le C++ effectue des copies par defaut. Il est parfois intéressant d'utiliser des références pour obtenir l'original, que ce soit par souci de performances ou par nécessité de le modifier. Dans le premier cas, il est est possible de récupérer une référence constante : on travaille bien sur l'original, mais le compilateur interdit de le modifier. C'est par exemple le cas pour la surcharge de l'opérateur `<<`. Mais le compilateur refuse un `const Polynomial &`. C'est parce qu'on fait appel à la méthode `degree`. En effet, le compilateur n'est pas capable de vérifier si, oui ou non, cette méthode modifie son instance. Résultat : dans le doute, il refuse.

La solution consiste à préciser que la méthode `degree` ne modifie pas son instance. Cela se fait grâce au mot-clé `const` placé à la fin de la déclaration (`int degree() const;`), ou lors de la définition de la méthode (`int degree() const { ... }`). Ce principe s'applique à la fois aux méthodes et aux fonctions membres, c'est pour cela qu'on utilise l'appellation générique *fonction constante*.

Par contre, ce n'est pas de la magie. En effet, si dans une fonction constante, on modifie l'instance ou on fait appel à une fonction non constante qui utilise une référence, la compilation échouera.

7 Exercices

Reprenez l'ensemble des méthodes vues précédemment, mais sans les approximations de concept qui ont permis d'aborder les concepts un à un.

Au final, il y aura donc besoin de :

- une classe `Polynomial` que vous ferez évoluer au fur et à mesure des points suivants.
- Un constructeur à partir d'un conteneur (vecteur ou array ou les deux).
- Un constructeur à partir d'au moins un flottant
- Une méthode `degree` (sans argument)
- Une méthode `coeffs` (sans argument)
- Une methode `to_string` (sans argument) qui renvoie un `std::string` contenant un code qui permet de recréer l'instance. Par exemple, si le polynôme vaut `x+1`, cette methode renverra `"Polynomial ({0, 1})"`.
- une methode privée `reduct` qui ne prend aucun argument, mais qui réduit le degré du polynôme si ses coefficients de plus haut niveau sont nuls.
- une surcharge pour l'opérateur de flux `<<` qui permette l'affichage en ligne sous une forme facile à lire comme `4*x*x + 2*x + 1` ou équivalent.
- une surcharge pour l'opérateur `()` qui permette les applications numériques Comme on utilise les `template`, cette fonction doit être la plus générique possible. En effet, il faut pouvoir calculer $P(z)$ avec z complexe alors que P a des coefficients réels (voir entiers).
- une surcharge pour les opérateurs `+=`, `-=`, `*= +`, `-` et `*`
- une surcharge pour la fonction `pow`. Il serait tentant ici de surcharger l'opérateur `^` à la place, mais il y a un gros problème : la précédence des opérateurs. En effet, si en C/C++ on écrit `x^2+1`, les règles définissent qu'il faut l'interpréter comme `x^(2+1)` et non pas comme `(x^2)+1`. Cela vient du fait qu'à l'origine, c'est opérateur est pensé pour coder l'opération `XOR`. Loin d'être utile, cette surcharge serait alors source de nombreux bugs, et au final contre-productive.

Pour chaque méthode/fonction, vous expliquerez en commentaire les stratégies que vous avez choisi. Notamment, si c'est une surcharge d'opérateur, pourquoi l'avoir mis en fonction libre ou en fonction membre. Lorsque vous recevez des arguments, pourquoi utiliser un passage par référence, ou pourquoi ne pas utiliser de passage par référence ... .

Dans un cadre habituel, la philosophie du C++ est de ne pas proposer de fonction/methode pour l'affichage. C'est à l'utilisateur final de définir le cadre et le format d'affichage.

8 Conclusion

Ben voilà, vous avez écrit une classe complète. Evidemment, il reste encore beaucoup de choses à implémenter, comme la recherche de racines, la division, la factorisation, ... Mais l'idée est là, le reste, c'est essentiellement du calcul mathématique et de l'algorithmique. Bien que ce soit intéressant, nous n'avons pas le temps de nous y attarder.

Une remarque assez importante en terme de stratégie : lors de l'ajout de nouvelles possibilités il faut orienter leur usage au minimum. C'est à l'utilisateur final de faire ses choix. Il faut donc soit proposer toutes les possibilités (mission impossible), soit laisser une autre bibliothèque gérer l'usage. C'est notamment pour cela que le C++ n'offre pas de méthode *toute faite* pour représenter les vecteurs. Le choix est laissé à l'utilisateur (soit il le fait lui-même, soit il choisi une bibliothèque qui correspond à son usage).