PG208
Programmation Orientée Objet / Langage C++
TP4

Y. Bornat A. Valade

April 11, 2024

La séance précédente nous a permis de voire les classes et leur principe. Cette séance sera dédiée à plusieurs petites choses bien pratiques. Que ce soit pour mieux gérer un code complexe, ou pour exploiter la force des concepts qu'elles contiennent. En effet, les classes ne sont pas *que* des types de données très évolués, on peut les utiliser dans des conditions bien différentes.

1 Diagramme UML

Les classes sont des types de données relativement complexes à gérer. Et encore, nous verrons plus tard que l'organisation de classes dans un projet peut vraiment devenir un casse-tête. A ce niveau, la documentation est un enjeu majeur, car personne ne peut gérer de grand projet seul, mais une documentation mal construite est contre-productive. Pour y remédier, un système de représentation standardisée a été adopté[1]. Il s'agit de la représentation *UML* (Unified Modelling Language). Ce standard propose, entre autres, une représentation visuelle des différents éléments d'un système, tels que les classes, les objets, les relations entre eux, les interactions et les comportements. UML s'applique également à la description de machines d'états, d'organisation (architecture système ou organisation sociale), *etc.* Nous nous limiterons à une approche rapide limitée à son application en *POO*.

Les classes sont représentées sous forme de rectangles divisés en trois parties : le nom de la classe, les attributs et les méthodes. Quand il y a une relation entre deux classes, ben on trace un trait ou une flèche entre elles. Pour le détail, c'est une suite de règles assez simples :

- Le nom de la classe est dans le compartiment du haut. Il est centré, écrit en gras et respecte la casse (Majuscules/minuscules).
- Les attributs sont écrits dans le deuxième compartiment. Chaque attribut est écrit sur une ligne séparée, avec sa visibilité, son nom et son type. Les deux derniers sont séparés par le caractère *deux-points*. La visibilité peut être publique (+), privée (-) ou protégée (#).
- Les méthodes arrivent en dernier. Chaque méthode est écrite sur une ligne séparée, sa visibilité, son nom avec ses paramètres entre parenthèses et type de retour. Le caractère : sépare la parenthèse fermante et le type de retour.
- lorsque le type d'un attribut de la classe est une autre classe[2] on trace une flèche entre la classe source (qui référence) et la classe cible (qui est référencée).
- si deux classes ont une référence croisée (chacune des deux a une référence sur l'autre), ce n'est pas une flèche mais un trait.
- pour une classe source, on indique le nombre de référence à côté de la base de la flèche (ou du trait si c'est une référence croisée)
- il est possible de décrire la relation avec deux ou trois mots max le long des traits.

[1] Notez bien qu'il s'agit d'un standard, pas d'une norme. Mais ce dernier est assez bien suivi.

[2] Techniquement il assez rare qu'une classe soit stockée dans une autre, on utilise beaucoup plus souvent des références.

- la disposition, les couleurs, *etc* sont laissés à l'appréciation artistique du concepteur.

Il se *pourrait* que, dans un avenir proche, vous ayez un rapport ou un gros compte-rendu à faire concernant un code en programmation orientéee objet. La communication avec des diagrammes UML étant systématique sur les projets en POO, entrainez vous sur les codes que vous allez écrire prochainement.

2 Programmation dite *par contrat*

2.1 Principe

Cette partie est un enrobage un peu théorique pour un concept que vous manipulez depuis un moment sans vous en rendre compte. Quand vous programmez, vous posez en permanence des invariants que vous vous obligez à respecter, et qui vous permettent de garantir que votre code fonctionne. C'est par exemple le cas pour une boucle qui parcours les éléments d'un vecteur avec un indice. L'invariant de boucle (l'indice désigne une case du tableau), pose une contrainte sur l'indice, il doit être compris entre 0 et la taille du tableau -1.

Dans les classes, c'est la même chose. Les différents attributs sont cohérents entre eux, mais rien ne le garanti s'ils sont affectés au hasard. Il faut aussi que les arguments des constructeurs permettent de garder la cohérence voulue. Utiliser une classe signifie aussi faire tout ce qu'il faut pour que ses données soient cohérente. On appelle ça la *programmation par contrat*. Par exemple, pour la classe `Polynomial` le contrat est que, `m_coeffs` contienne au moins une valeur et que, s'il y a plusieurs valeurs dans `m_coeff`, celle d'indice le plus élevé soit non nulle. (sinon, le degré du polynôme n'est pas lié au nombre de coefficients).

Pour gérer tout ça, le C++ a hérité du C une commande de vérification de condition rapide : les assertions.

2.2 Syntaxe

Pour utiliser les assertions, il faut d'abord inclure la bibliothèque correspondante : `#include <cassert>`. Ensuite, il suffit d'appeler la *macro*[3] `assert()`. Le seul argument qu'elle prend est une condition. Si la condition est vraie, rien ne se passe. C'est que tout va bien. Si elle est fausse, c'est qu'un contrat n'est pas respecté, et le programme plante méchamment.

Le langage ne permet pas d'ajouter de message explicatif, mais l'erreur donne le numéro de ligne ou s'est passé l'assertion négative et, bien souvent, recopie la ligne fautive (avec la condition donc). La technique souvent employée est donc d'ajouter la chaine de caractère à la condition comme suit :

```
1    assert(resp==42 && "La vie, l'univers, ....");
```

Techniquement le programme fait un `ET` bouléen entre `resp==42` et la chaine de caractère. Étant donné qu'une chaîne de caractères renvoie toujours `true`, il n'y a que le test qui est réellement pris en compte.

L'usage des commentaires reste possible, mais il n'est pas pratiqué car rien ne garanti que les commentaires de la ligne seront recopiés dans le terminal.

2.3 Intérêt

Cette technique présente de nombreux avantages :

- Elle est rapide
- Le lecteur peut rapidement identifier qu'on vérifie la validité de quelque chose.
- Elle est désactivale sans effort. En effet, le comportement de `assert` est défini en fonction d'une définition du pré-compilateur. Il suffit alors d'ajouter `#define NDEBUG` avant l'inclusion de `<cassert>`, et tous les tests seront ignorés.

[3] Si vous ne savez pas ce qu'est une macro, vous survivrez. On dira alors que c'est *presque* comme une fonction

Là ou le dernier point est intéressant, c'est que les définitions peuvent être faites dans la ligne de commande du compilateur (option `-D` pour *gcc/g++*). Ainsi, en compilant avec `-DNDEBUG`, aucun test de type `assert` ne sera fait, et on gagne en performances.

Mieux encore, dans le cas d'une compilation multifichier, il est possible d'activer cette option (ou pas) fichier par fichier.

2.4 Les tests unitaires

La notion de *test unitaire* est liée à la vérification des composants de base d'une bibliothèque. Le principe est de faire de nombreuses assertions élémentaires qui vérifient les contrats. Un développement saint multiplie les tests unitaires de l'ensemble des structures utilisée avant de les assembler dans un système complet. La vérification à l'échelle du projet est constituée des tests *d'intégration*, qui ne font pas partie des objectifs de ce cours. Encore une fois, si on effectue des `assert` pour les tests unitaires, ils peuvent être dans le code final, et simplement retirés grâce à la définition de `NDEBUG`. Le principe est de ne jamais retirer un test unitaire qui a été plac" dans le code pour éviter les régressions lors d'une mise à jour.

Par contre, les assertions sont faites pour détecter les anomalies dans le code (debugage d'une bibliothèque ou vérification de la validité d'arguments). elles ne *doivent pas* être utilisées pour les problèmes liés à l'environnement (erreur de saisie utilisateur, indisponibilité d'un fichier, coupure réseau, ...). Pour cela, il y a les exceptions !

3 Exceptions

A partir du moment ou il existe un mécanisme permettant de faire remonter des erreurs, il est possible de l'utiliser de façon à faire remonter les comportements exceptionnels, inattendus ou pour lesquels le code n'a pas été prévu.

Le mot-clé pour générer un tel évènement est `throw`. Deux grosses différences avec les assertions : il est inconditionnel (pas grave, on peut le placer dans un `if`), et il prend un argument (comme `return`). Le type (la classe) de l'argument va coder le type d'exception. Il existe de nombreux type, et chaque mise à jour de la norme en apporte de nouveaux. Pour C++11, certains des plus basiques sont : `std::bad_typeid`, `std::bad_cast` , `std::bad_weak_ptr`, `std::bad_function_call`, `std::bad_alloc`, `std::bad_array_new_length`. Attention, `std::bad_array_new_length` est une forme particulière de `std::bad_alloc`.

Il existe également des types plus élaborés : `std::invalid_argument`, `std::domain_error`, `std::length_error` , `std::out_of_range` et `std::future_error` qui sont des formes particulières de `std::logic_error`. Mais aussi `std::range_error`, `std::overflow_error`, `std::underflow_error`, `std::regex_error` et `std::system_error` qui sont des formes particulières de `std::runtime_error`.

Les types élaborés ont la particularité d'accepter une chaine de caractère lors de leur création. Il est ainsi possible d'écrire un vrai message d'explication du problème. Ils sont tous définis par `#include <stdexcept>`.

Histoire de vous faire la main, écrivez un programme tout simple, qui écrit du texte dans le terminal, lance une exception avec un message de votre choix, puis écrit une autre ligne de texte.

Évidemment, la dernière ligne de texte n'apparait pas. Par contre, vous aurez remarqué que le texte de l'exception s'affiche sans référence de ligne ou de fichier. En effet, comme une exception est supposée avoir des causes externes, ce n'est pas le programme qui est en cause. Le texte de description doit donc se cantoner à décrire le problème, et eventuellement ses causes supposées.

3.1 Récupération

Les exceptions ne sont pas que des assertions un peu modifiées. Elles ont une particularité bien plus intéressante : on peut les intercepter. Par exemple, on peut imaginer qu'une fonction `foo` appelle une sous-fonction `bar` qui a pour mission de charger des données dans un fichier et de renvoyer un résultat. Si l'accès au fichier échoue, `bar` ne peut pas renvoyer de résultat alors qu'elle y est obligée. Elle déclenche donc une exception. Si `foo` n'a rien prévu de particulier, l'exécution du programme s'arrête et l'exception est affichée comme une erreur. Mais si `foo` est préparée à l'éventuel déclenchement d'une exception, elle peut la rattraper, demander de spécifier un autre nom de fichier à l'utilisateur et retenter d'éxécuter `bar`

3.1.1 syntaxe

Pour générer une exception, on utilisera le code suivant (en une ligne):

```
1    throw std::runtime_error("On insulte joyeusement l'utilisateur si mauvais");
```

mais si vous préférez quelque chose de plus verbeux, on peut décomposer les étapes de la création :

```
1    std::string text {"Cher Utilisateur\n"};
2    text += "t'es nul !";
3    std::runtime_error exc {text};
4    throw exc;
```

Vous êtes bien sur libre de concevoir des message de taille quelconque. L'essentiel étant de bien identifier le problème.

Pour intercepter l'exception, la syntaxe est la suivante (à placer dans une fonction) :

```
1        // any code before ...
2        try {
3            // possibly faulty code
4        } catch (std::range_error& ex) {
5            // exception management
6            ex.what(); // retreive the exception message
7            throw ex;
8        } catch (std::runtime_error& ex) {
9            // another possible exception
10           std::cout << ex.what() << std::endl; // print the exception message
11       } (...) {
12           // all other exceptions
13           // if this happens, it is generally bad news
14           throw;
15       }
16       // any code after ...
```

Le bloc suivant `try` des lignes 2 à 4 est de longueur quelconque. Si la moindre exception se déclenche à l'interieur de ce bloc, il est interrompu et l'exécution passe les différents types possibles.

Selon le type de l'interruption, le bloc correspondant est éxécuté. Notez que `std::range_error` doit être testé *avant* `std::runtime_error`. Comme le premier est un cas particulier du second, les deux correspondent à un `std::runtime_error`. Si le test était fait dans l'ordre inverse, le second ne serait jamais utilisé car les exception auraient déjà été traitées par le premier.

Redéclencher ou pas l'esception avec `throw` ne dépend que de la volonté du concepteur et de ses contraintes.

3.1.2 Pourquoi c'est intéressant

Les exceptions énumérées ci dessus sont celles qui peuvent être indifférement interceptées ou libérées. Par contre, il est possible de passer n'importe quel type ou classe avec `throw`. Par contre, il devient impératif d'intercepter l'interruption. Si ce n'est pas le cas, on obtient alors une erreur au même titre qu'une assertion. Il sera impossible d'accéder à la valeur renvoyée, le message affichera uniquement son type.

On se retrouve donc avec un canal de retour de valeur indépendant de la valeur renvoyée par une fonction, et surtout non typé. Il est possible de renvoyer n'importe quel type ou classe à condition qu'il y ait une clause `catch` qui puisse la récupérer. On peut ainsi utiliser ce genre de code :

```
1    try {
2        subfunc();
3    } catch (int& i) {
4        std::cout << "got an int    = " << i << std::endl;
5    } catch (double& d) {
6        std::cout << "got a double = " << d << std::endl;
7    } catch (const std::string& s) {
8        std::cout << "got a string = " << s << std::endl;
9    } (...) {
10       std::cout << "Ouch ! got a real exception" << std::endl;
11       throw;
12   }
```

Par contre, il ne faut pas perdre de vue que les exception, comme leur nom l'indique, doivent rester exceptionnelles car elles sont (beaucoup) moins performantes que des retours de fonction classiques.

Leur rôle doit rester cantoné aux situations qui sortent du cadre normal de fonctionnement et pour lesquels il n'est pas rentable de concevoir un canal de retour mieux structuré.

3.2 Référencement

Il est possible (conseillé) de préciser dans le prototype d'une fonction si elle est susceptible de déclencher des exception, et, si oui de quel type. Une fonction dont on est sur qu'elle ne déclenchera jamais d'exception sera suivie du mot-clé `noexcept`. Par exemple :

```
1    int fact(int n) noexcept;
```

Au contraire, pour une fonction qui peut renvoyer une exception de type `int`, `double` ou `std::system_error`, on écrira :

```
1    int fact(int n) throw(int, double, std::system_error);
```

Ces ajouts de définition offrent deux avantages :

- L'utilisateur de la fonction sait s'il doit gérer des exceptions
- Le compilateur peut se permettre certaines optimisations quand il sait si une fonction ne produit pas d'exception.

Par définition, un destructeur de classe est toujours `noexcept` car le C++ ne peut gérer qu'une seule exception à la fois.

3.3 Limites

Les exception ne sont pas utilisables dans tous les cas. Notamment, leur comportement et leur syntaxe sont très différents dans les constructeurs. Dans le cadre de cet enseignement nous éviterons simplement de déclencher des exception depuis les constructeurs.

De plus, pour les exceptions comme pour les assertions, si le programme se termine, les destructeurs des variables ne sont pas appelés. Cela peut parfois poser problème.

3.4 Exercice

Pour cet exercice, l'ojectif est de créer une class `ifstream_exc` ou `ofstream_exc` (au choix) qui permet de lire ou écrire un fichier. L'objectif est de lancer une exception si l'ouverture du fichier ne s'est pas bien passée. (Tout le monde a certainement le souvenir d'avoir, au moins une fois, galéré sur un segfault donc la cause était l'utilisation d'un `FILE *` dont l'initialisation avait échoué, non ?). En C++, c'est pire, l'écriture ou la lecture ne se font pas, mais il n'y a pas de système de signalisation. On a juste les méthodes `good` et `fail` qui renvoient un booléen sur l'état du flux.

Comme il n'est *pas possible* de jeter des exceptions dans le constructeur, vous pouvez (au choix) ne pas ouvrir le fichier dans le constructeur, mais forcer l'appel à `open` (propre, mais oldfashion) ou provoquer l'exception dans l'opérateur de flux `<<` ou `>>`.

L'intérêt est bien sur de fournir au moins un programme qui intercepte l'exception et qui transforme un gros crash en petit message.

4 Gestionnaire de Ressource (*Handler*)

Maintenant, passons à une technique qui va nous rendre service ...

4.1 Principe

Bien que ça ne paraisse pas évident au premier abord, une classe en apparence très pauvre peut s'avérer être un excellent gestionnaire de ressources. Les ressources d'un ordinateur ne se limitent pas au temps processeur ou à la mémoire. En effet, une ressource peut être l'accès à un fichier, la communication avec un module externe, la carte graphique ou encore bien d'autres choses. Bien souvent, ces ressources ont besoin d'être ouveres et/ou réservées, et, après utilisation restaurées dans un état par défaut.

Quand tout va bien, tout va bien ! Les ressources se comportent correctement, le système est stable, mon chien m'obéit, le programme n'est pas buggé et la loi de Murphy n'est qu'un concept éloigné. Puis arrive le bug ou l'exception, qui interromp l'éxécution et laisse les ressources dans un état inexploitable. Dans certains cas, il faut redémarrer l'ordinateur, dans d'autres, les fichiers de sortie ne sont pas écrits et des données sont perdues ...

Heureusement, pour protéger les ressources nous avons maintenant les classes qui proposent un couple d'antagonistes magnifiques : le constructeur et le destructeur. On peut dédier le constructeur à la réservation de la ressource (comme on le ferait avec un `open` en C), et le destructeur à se libération. Ça n'a pas l'air de grand chose comme ça, mais ça présente de nombreux avantages :

- Cela évite les *oublis accidentels*. On est certain que la libération de la ressource sera appelée à la fin de son utilisation.
- On simplifie le code. En effet, si vous ouvrez une ressource dans une fonction qui a plusieurs `return`, il n'est pas nécessaire d'effectuer le travail de nettoyage à chaque fois. Il sera fait par le destructeur après le `return` et avant de continuer la fonction appelante.
- La ressource est fermée proprement, même si une exception est levée dans son éxécution. La gestion de la ressource concernée est donc plus robuste.

Techniquement, vous avez déjà utilisé ce concept avec les flux vers les fichiers. en C++, il n'y a pas besoin de fermer un fichier car c'est le destructeur qui s'en occupe. Donc pas de souci, même si on appelle n fois une fonction qui ouvre un fichier sans le fermer explicitement.

4.2 Les limites

C'est bien beau tout ça, mais les ressources ne seront pas protégées dans toutes les situations.

- La coupure d'alimentation contre laquelle on ne peut rien.
- Fermeture du processus par le système d'exploitation
- Exception non récupérable (il y en a, comme `std::bad_exception`)

4.3 Exercice

Plutôt que de passer du temps sur les concepts, on va passer à la pratique. L'objectif est de proposer un type de variable non volatile. Nous nous limiterons au type double pour éviter de passer trop de temps en codage.

Techniquement, cette variable est stockée dans un fichier. Lorsqu'on instancie un nouvel objet, le constructeur récupère un nom de fichier et une valeur par défaut (optionnelle). Si le fichier existe déjà, la valeur par défaut n'est pas utilisée, on charge la valeur présente dans le fichier.

La classe `pnv_double` va donc contenir :

- une chaine de caractères correspondant au nom de fichier pour le stockage non volatile
- un double : la valeur qui nous intéresse, nous l'appellerons `val`
- un constructeur qui, charge la valeur du fichier, ou la valeur par défaut
- un destructeur qui enregistre le double dans le fichier

En l'état, une fois l'instance initialisée, il est possible de modifier `val` comme une variable. La notation est juste un peu plus lourde puisqu'on l'appellera `instance.val` à chaque fois. Une fois la classe en fin de vie, le destructeur se chargera de sauvegarder la valeur du double.

Pour rendre les choses encore un peu plus fluides, il est possible de surcharger l'opérateur unaire `*` de `pnv_double` (déréférencement) et de lui faire renvoyer une référence vers `val`. Ainsi, modifier `*instance` modifiera directement `val`. Ça vous rapelle quelque chose ? d'où l'appellation `pnv_double`, Pointeur Non Volatile sur `double`.

Cet exemple est un peu simpliste. Il pourrait être associé à un template pour mémoriser n'importe quel type de variable. Mieux encore, un fichier pourrait contenir plusieurs valeurs, *etc.* Le reste n'est qu'une queston de temps.

5 Retour sur les pointeurs

Le dernier exemple montre en quoi les références remplacent les pointeurs dans de nombreuses situations. Les pointeurs tels qu'utilisés en C restent très dangereux, et l'objectif est de s'en passer complètement (objectif impossible puisque de nombreuses méthodes de la bibliothèque standard utilisent encore des `char *` au lieu de `std::string`).

5.1 Au revoir `malloc`

Après le passage par référence dans les fonctions, un grand usage des pointeurs est dévolu à l'allocation mémoire dynamique. En C elle était faite avec `malloc`. Il fallait créer un pointeur, réserver une zone mémoire ayant la bonne taille, puis l'initialiser.

Le C++ apporte un nouvel opérateur (`new`) qui présente l'avantage de calculer lui-même la taille de la zone nécessaire. Autre intérêt, le pointeur est typé dès le départ. plus besoin de convertir le type de données. La syntaxe est assez immédiate :

```
1    int * toto = new int;
```

Pour libérer la zone mémoire, le pendant destructeur est `delete` :

```
1    delete toto;
```

5.2 Pointeurs *intelligents*

L'inconvénient des pointeurs, reste qu'il est très difficile d'empêcher les fuites mémoire. Cela vient du fait que les zones mémoire sont identifiées par le pointeur qui les désigne. Si on perd le pointeur, il n'est plus possible de libérer la zone mémoire. Ce genre de comportement mène à de nombreuses fuite mémoire. Le C++ tente de fournir des outils pour y remédier : les pointeurs *intelligents* (par opposition aux pointeurs de C classique qualifiés de *nus*)

Le principal est un pointeur unique (`std::unique_ptr` disponible dans `<memory>`). Le principe est simple : c'est le seul pointeur qui fait autorité sur la zone mémoire concernée. Le seul qui contient les mécanismes lui permettant de libérer cette zone. Techniquement,le pointeur est enfermé dans une classe. Lorsque la classe est détruite, son destructeur est appelé, et ce dernier libère l'espace mémoire.

```
1    std::unique_ptr<int> pointeur_unique {std::make_unique<int>(valeur)};
```

Pour initialiser un pointeur unique, soit on lui attribue la valeur qui signifie *pointer sur rien* (`nullptr`), soit on appelle `std::make_unique<T>` qui reçoit en argument les éléments passés au constructeur de `T`. La ligne précédente montre l'initialisation d'un pointeur unique sur `int` avec `valeur` passé au constructeur. Cette syntaxe inhabituelle est liée aux protections du pointeur contre la copie.

Dans ce code, `pointeur_unique` est le gardien de notre zone de données. Si on perd sa valeur, la zone de données ne sera plus accessible (logique), mais la mémoire qu'elle occupe sera libérée (cool). Ce type de comportement protège des fuites mémoire, mais reste dangereux. En effet, ce n'est pas parce qu'une zone mémoire est libérée qu'elle était inutile. Il reste donc primordial de s'assurer de la bonne santé de la structure de données. de plus `pointeur_unique` ne peut pas être une variable locale, sans quoi le pointeur serait détruit très vite. Il convient donc de stocker tous les pointeurs uniques dans une structure pérenne. Par ailleurs, un pointeur unique ne peut pas être copié. Il faudra donc toujours le passer par référence.

Pour utiliser le pointeur, il y a deux solutions :

- avec un déréférencement classique (`*`), on obtient une référence vers la zone mémoire (donc l'objet pointé).
- avec la méthode `get`, on en obtient un pointeur nu, toujours vers l'objet pointé.

Reste que ... s'il n'est pas possible de copier un pointeur unique, comment le stocker dans un vecteur ou dans l'attribut d'une classe ? Pour cela, il y a la fonction `std::move`, qui prend une référence vers l'original, casse son association avec la zone de données et renvoie une *copie* (qui est en fait le nouvel original) qui sera acceptée pour copie.

5.3 Autres usages des pointeurs nus

Il existe encore quelques usages des pointeurs nus que nous n'avons pas couvert. Nous n'y passerons pas plus de temps car ce n'est pas l'objectif principal de ce cours et que certains sont assez récents. Sachez toutefois que si vous pensez avoir besoin d'un pointeur nu, il est fort probable qu'une version plus sécurisée aie été mise en place dans la norme.

5.4 Exercice

Pour se faire la main sur les pointeurs, nous allons implémenter une liste chainée (la plus simple des structures chaînées). La nature des élements de la chaine est laissée à votre préférence.

La vision objet change un peu les choses. En effet, en programmation impérative, les fonctions sont disponibles dans toutes les situations. En objet, s'il n'y a pas d'élément, il n'y a pas d'accès possible aux méthodes. Il faudra donc créer deux classes :

- `Liste` : concentre les méthodes et le pointeur (unique) vers le premier maillon de la chaine (ou vers `nullptr` si la chaine est vide). Par contre, elle ne contient pas d'élément. Les méthodes fournies doivent être (au moins) :
  - Un constructeur à partir d'un vecteur
  - `size` : nombre d'éléments dans la liste
  - `push_front` : ajouter un élément au début
  - `pop_front` : supprimer le premier élément et le retourner, déclenche une exception si la liste est vide.
  - `push_back` : ajouter un élément à la fin
  - `pop_back` : supprimer le dernier élément et le retourner, déclenche une exception si la liste est vide.
  - `to_string` : renvoyer une représentation au format chaine de caractère
- `Maillon` : la structure chainée à proprement parler. Elle contient le strict minimum : un élement, un pointeur unique vers le maillon suivant et quelques autres bricoles (ou pas) que je vous laisse déterminer.

Trucs & Astuces : Pour affecter une valeur à `nullptr`, il faut soit la passer au constructeur, soit faire appel à `std::move`. Le souci, c'est que `std::move` exige une référence. Il font donc bien veiller à toujours obtenir une référence vers l'original pour effectuer la copie.

S'il vous reste du temps, vous pouvez vous *amuser* à implémenter l'opérateur `[]`. Ou d'autres choses.

6 Conclusion

Il est un mot-clé qui n'a pas encoré été mentionné dans cet enseignement, c'est `auto`. Il désigne un type ou une classe déduit automatiquement en fonction du contexte (d'où son nom). Ce mot-clé permet d'éviter des écritures de types parfois fastidieuses. Par exemple, un tableau de cinq polynômes à coefficient réels s'écrit `std::array<Polynomial<double>, 5>`. Malheureusement, dans un cadre de formation, `auto` est très dangereux car il permet d'éluder des questions quant aux données manipulées. Ne pas se poser ces questions fait courir le risque de ne pas comprendre pleinement le problème. C'est pour cela que son usage est proscrit dans les travaux que vous devez rendre.

A ce stade, nous commençons à avoir bien fait le tour des nouveautés du C++ et de ce qu'apporte la vision objet. Il ne reste qu'un point que nous verrons lors de la prochaine séance, la possibilité de construire une classe en se basant sur une classe déjà existante pour en récupérer les propriétés.