Chapitre 5: Functional C++

Martin Quinson

Épisode précédent

- Implémentation C++
  - Représentation mémoire des objets triviaux: comme des structures C
  - Objets non triviaux = présence d'une vtable pour les méthodes virtuelles
  - Transtypage: Assez similaire au C
    - Objet (comme scalaires): perte d'information (zone mémoire plus courte, l'autre vtable est utilisée) donc pas de upcasting
    - Pointeurs d'objet: ne change que la façon d'interpréter la zone pointée
  - Mangling de symboles: règles de mutilation pour encoder les namespaces et types de paramètres dans le nom de symbole C
  - Invariance du C++: les redéfinitions ne peuvent pas raffiner le type des params ou retour
- Gestion automatique de la mémoire:
  - RAII
  - smart pointers: un objet englobant gère la mémoire d'un pointeur protégé.
    - `uniq_ptr` n'a pas d'opérateur de copie: `operator=(T t)=delete;`
    - `shared_ptr` fait le refcounting dans son opérateur de copie et son destructeur

1 Programmation fonctionnelle

- C'est le troisième paradigme de programmation accessible en C++: on a déjà parlé au cours 2:
  - Procédural C: découpe en procédure ayant des effets de bord; les données sont des globales
  - Orienté Objet: découpe en entités regroupant une partie de l'état global et les traitements qui s'y appliquent
  - Fonctionnel: déclaratif centré sur les traitements (inspiré des maths) Evaluation d'expressions plutôt qu'exécution de commandes
- Chaque approche a des avantages et des défauts:
  - Procédural proche de la réalité de l'ordinateur (M99), mais peu d'abstraction et pas de séparation
  - OO plus facile à concevoir pour certain (des choses font des actions), mais séparation de surface seulement (catastrophe en multithread). Tendance au bloat.
  - FP donne des codes plus courts et épurés, mais plus dur à écrire, et tendance au code golf impénétrable.
- Quelques composants qu'on retrouve souvent en FP
  - Fonction de premier ordre : manipuler des données de type "fonction", et currification
  - Programmation d'ordre supérieur: on passe des fonctions en paramètres d'autre fonctions
  - Données immutables pour moins de problèmes (surtout en matière de concurrence)
  - Fonctions pures (sans effet de bord, comme les fonctions maths, pour donner tjs le meme résultat quand on leur donne les même données)
  - Type algébriques et pattern matching (ça c'est très dur en C++)
- Nous allons voir comment ces choses sont possibles en c++ moderne

1.1 Programmation d'ordre supérieure

- En C, il faut faire des pointeurs sur fonction, et c'est très facile de faire n'importe quoi car le compilateur n'aide aucunement
- En C++, on a un type `std::function` qui permet de manipuler des choses invocable comme une fonction.
  - Toute classe dotée d'un opérateur d'invocation `ret operator()()` peut être utilisé comme une fonction.
  - Le type (pour une variable ou un paramètre) est comme `std::function<bool(int, Mytype)>`
- Cela se combine bien avec les algorithmes de la STL, comme `std::for_each` qui correspond à un map (cf l'exemple de code).
  - Il vaut mieux prendre l'habitude d'utiliser `std::begin(vect)` que `vect.begin()` car le premier fonctionne également avec les tableaux

1.1.1 Lambda

- Avantages des lambdas : moins de sucre syntaxique à écrire; pas de fonction à usage unique dont le nom pollue l'espace de nommage
- `[capture list] (parameter list) {function body}` : la tête, les paramètres, et le corps
  - La liste des captures donne les variables du contexte où se trouve la def de la lambda qui doivent être rendues disponibles à l'endroit où la lambda sera utilisée
  - On peut capturer par valeur, comme ici, ou par référence avec `[&i, &d]`.

  - On peut aussi capturer toutes les variables locales par copie avec `[=]`, ou capturer toutes les variables locales par référence avec `[&]`
  - Si on est dans une méthode de classe, on a probablement envie de capturer `this` aussi, car sinon la lambda ne s'exécute pas dans le contexte de la classe.
- Le mécanisme de capture n'est pas magique, le compilateur génère juste une classe à usage unique. Les variables capturées sont des champs de la classe (initialisés correctement par le constructeur) et donc accessible par l'opérateur d'évaluation `operator()()`.
  - Cela donne l'impression que les symboles utilisés par le corps de lambda **sont** les variables capturées (même nom et tout), mais en fait c'est une copie.
  - D'ailleurs, si la valeur des variables capturées change entre temps, la lambda ne le verra pas lors de son exécution
  - Si on veut que la lambda modifie les paramètres capturés, il faut le dire avec `mutable` à écrire juste avant le corps de fonction (là où dans d'autres contextes, on écrit `const`)
- Il faut parfois préciser le type de retour quand le compilateur n'arrive pas à déduire les types statiques `[capture list] (parameter list) -> type_retour {function body}`
  - On peut tjs post-fixer ainsi le type de retour (`auto fun() -> type`), mais je trouve que ça alourdit
- En C++14, on peut faire des lambda généralisées, c'est à dire des fonctions qui s'applique à tous les objets sans savoir leur type. Cf l'exemple de code.
- Au final, c'est très pratique de pouvoir stocker des fonctions dans des variables. Trop dommage que les lambda python soient limitées à une seule ligne à cause du manque d'accolades.

1.1.2 Currification

- Création d'une nouvelle fonction par application partielle des premiers paramètres d'une fonction.
  - C'est une idée de théoricien qui s'avère parfois utile, et c'est surtout très facile à faire en C++.
  - Soit on fait la lambda à la main, soit on utilise `std::bind`
- On peut aussi changer l'ordre des paramètres au passage avec les `_1` qui sont des placeholders des différents paramètres reçus par `std::bind`
- L'exemple classique est le générateur aléatoire, présenté sur la feuille, mais ça marche aussi avec le contexte d'exécution d'une requête sur une base de données ou autre

1.2 Données immutables et fonctions pures

- `const` indique qu'une variable est constante, ou qu'une fonction ne modifie pas ses paramètres.
  - Informer le compilateur lui permet d'optimiser et surtout de nous aider à chasser nos erreurs
- On dit qu'une telle fonction est **référentiellement transparente** puisque l'environnement ne voit pas la différence quand on remplace l'invocation de la méthode par la valeur qu'elle va retourner.
  - Il n'y a donc pas d'effet de bord (affichage, modification de l'état global, écriture sur disque)
  - Pour qu'un programme soit intégralement référentiellement transparent, il faut que les variables soient constantes après affectation. On peut alors remplacer leur nom par leur valeur.
- **Expression constante**: `constexpr`
  - ce décorateur sur une variable -> valeur connue à la compil
  - sur une fonction -> valeur connue à la compil SI les inputs le sont. Sinon c'est juste une const
- Programmer sans aucun effet de bord est très limité. Le programme ne peut pas être interactif.
  - Il faut quand même réduire le nb d'effets de bord pour simplifier la compréhension du prog

  - On peut faire un jeu complet du même genre que le projet, mais avec une seule variable mutable : l'état du monde. A chaque itération, on remplace le monde actuel par le suivant, qui est une copie: `world = update(world)`. On peut même le faire efficacement, et ça garantie l'absence de plusieurs catégories de bugs.

1.3 Types algébriques

- c'est pour définir des types sommes: soit ça, soit ça. Comme par exemple soit une référence vers un objet valide, soit le pointeur NULL. Ou encore une valeur python, qui est soit un entier, soit une chaine de caractères soit un nombre à virgule.
  - En C, on va faire des unions mais c'est très bas niveau et sans aucun garde fou. Si on a un entier et une chaine; on modifie l'entier, quelle proba pour que la chaine soit valide? Ca serait pire en C++: si j'ai deux champs dont un seul est valide à la fois, que faire dans le destructeur?
- `std::optional` (C++17): Une valeur valide, ou rien de bon (cf exemple)
  - Le constructeur sans paramètre fait une valeur invalide, que l'on peut créer explicitement avec `std::nullopt`
  - Constructeur avec paramètre fait une valeur valide, qui peut rendre le type du contenu déductible
  - `make_optional` permet d'utiliser auto ou de le passer en paramètre
  - Il y a une conversion implicite du type contenu vers l'optionnel valide
  - Les opérateurs de déréférencement sont invalides si la valeur l'est ! Il faut uitliser `value()` ou `value_or()` pour faire la vérification de validité
    * Damn besoin d'optimisation du C++
  - `https://www.bfilipek.com/2018/05/using-optional.html`
- `std::variant` (C++17):
  - C'est une généralisation où on peut faire la somme de plusieurs types valides.
  - Assez facile à créer, pas forcément pratique à utiliser car `get_if` travaille sur des pointeurs
  - On peut même faire la forme simplifiée du pattern matching (branchement structurel sur le type, sans considération des valeurs) avec cet objet et des fonctions lambda. Mais on va pas se mentir, la syntaxe est moche comme du C++. Il y a des propositions pour faire du branchement sur les valeurs, mais les syntaxes deviennent trop vilaines pour rentrer dans le standard C++ (c'est dire)

1.4 Les itérateurs

- On les utilise depuis longtemps de façon implicite pour parcourir les conteneurs de la STL. On va écrire un itérateur simple en TP.
- A l'usage, on demande au conteneur pour obtenir un itérateur. Il y a plusieurs types (selon qu'ils permettent de modifier la collection ou juste de lire, qu'il sont à usage unique, et le degré de liberté dans l'ordre de parcours), et chaque collection de la STL en offre plusieurs:
  - begin/end: début et fin
  - cbegin/cend: idem mais ce sont des itérateurs constants que je ne peux pas modifier
  - rbegin/rend: itérateurs en sens inverse, ie en commençant par la fin
  - crbegin/crend: constant et sens inverse
- Attention, certaines opérations invalident les itérateurs (entre autres toutes les modifications de la collection). Cf la doc.
- Simple à implémenter avec les opérateurs:

  - `operator*` retourne l'objet derrière l'itérateur en ce moment
  - `operator++` (et éventuellement `operator--`) décalent l'itérateur d'un élément
  - `operator==` et `operator!=` comparent les opérateurs entre eux (pour savoir si le parcours est terminé). Pour comparer les éléménts, faut d'abord déréférencer
  - `operator=` décale l'opérateur. Pour assigner une nouvelle valeur à l'emplacement pointé par l'itérateur, faut dérérencer.
- Il faut apporter des informations en plus pour pouvoir les utiliser dans la STL (par exemple expliquer comment ajouter des choses à la collection à l'endroit de l'itérateur). L'implémentation n'est ni compliquée ni très élégante. Vous irez chercher sur internet quand vous aurez besoin, mais ça ne sera pas pour ce module.
  - `using iterator_category = std::forward_iterator_tag;`
  - `using difference_type = std::ptrdiff_t;`
  - `https://internalpointers.com/post/writing-custom-iterators-modern-cpp`
  - `https://users.cs.northwestern.edu/~riesbeck/programming/c++/stl-iterator-define.html`

1.5 Ranges et views

- La STL offre des algorithmes sur les conteneurs, ce qui était très en avance sur son temps
  - map -> `std::for_each(std::begin(v), std::end(v), OP)`
  - filter -> `std::remove_if`
  - reduce/foldLeft -> `std::accumulate(std::next(std::begin(v)), std::end(v), v[0], OP)`
  - foldRight -> `std::accumulate(std::next(v.rbegin()), v.rend(), v.back(), OP)`
- Mais la syntaxe est assez pénible (il faut donner deux itérateurs – début et fin), et on peut pas combiner, chaîner les opérations comme en shell
- Les ranges et les vues de C++20 permettent de faire ça très simplement
  - Un range est simplement un itérateur que l'on peut parcourir
  - Une vue est une opération sur un range. La vue ne possède pas les données, qui restent propriété du range. C'est assez classique, conceptuellement
  - On peut combiner très simplement, et on peut même utiliser le fait que les vues sont paresseuses à l'évaluation. Cf le code proposé.
  - `std::views::iota(N)` retourne l'infinité des nombres à partir de `N`
  - `std::views::take(N)` ne garde que les `N` premiers éléments de la vue
  - Quelques vues définies:
    * `all()` (tous les éléments)
    * `filter(pred)` (ce qui vérifie un prédicat booleen)
    * `transform(lambda)` (pour faire un map)
    * `take(N)` (prend que les N premiers), `take_while(pred)` (prend tand que le prédicat dit vrai)
    * `drop(N)` (jette les N premiers) `drop_while(pred)` (jette tand que le prédicat dit vrai)
    * `join()`, `split`, `common` (filtrage inter-vues), `reverse`
    * `element(i)` (les Iiemes elements des tuples), `keys()`, `values()`
- Le gros défaut est que cette partie du standard C++20 n'est pas encore inclu dans les compilateurs en ce début d'année 2021.
  - Il faut donc utiliser la bibliothèque `range-v3`, qui est inclu dans toute les bonnes distributions
  - Il faut aussi compiler avec `--std=c++20`

  - En plus, le zip n'est pas dans C++20 pour de sombres raisons techniques et il faudra attendre C++23 pour l'avoir par défaut. `range-v3` a de beaux jours devant elle.

1.6 Éléments FP absents du C++

- Évaluation paresseuse: l'évaluation C++ est stricte (=non paresseuse), bien sûr, mais on peut facilement implémenter des systèmes de memoisation comme on verra en TP.
- le pattern matching structurel ne fait pas partie du langage. Il y a des tentatives de qqch, mais il faut savoir être raisonnable et passer au Rust après un moment
- Le système de type est mieux que le C, mais c'est encore loin de l'isomorphisme de Curry–Howard (qui dit qu'un prog bien typé en Coq est isomorphe à une preuve). Là encore, Rust est l'étape suivante, même si je ne sais pas si c'est la dernière étape du chemin.
- En CC, on peut faire beaucoup (mais pas tout) le FP en C++. Mais la syntaxe est parfois rebutante. Et encore, vous avez pas vu les messages d'erreur que ça produit, punaise.

2 Programmation générique

- La programmation générique est quand on écrit des fonctions qui peuvent s'appliquer à tout type de données, indifféremment.
- Habituellement, c'est limité aux langages interprétés, mais C++ permet de faire ça.
- On a déjà croisé des templates de méthode au premier cours, pour écrire une fonction `min` utilisable pour tous les types ayant `operator<()`
- On peut comprendre ces choses comme une extension du préprocesseur, où du code est produit en cas de besoin lors de la compilation.
- Le compilo réalise là où la template est utilisée qu'il lui faut générer du code supplémentaire.
  - C'est assez troublant car ça veut dire qu'on a parfois un code qui fonctionne, puis quand l'appelant fait une instanciation qui n'était pas testée avant, ca fait une nouvelle erreur sur le code de la template.
  - Mais cela permet de belles choses, y compris la STL qui veut dire Standard Template Library.
- STL = Standard Template Library, par Alexander Stepanov qui s'intéresse à la prog générique dès 1979, intégré au standard C++ dès l'origine. Un peu par hasard.
  - decltype
  - Sémantique des types de retour
  - Templates récursives: juste dire que c'est possible et que ça ouvre la porte

2.1 Patrons de classe

- On a un début d'exemple sur la feuille. C'est assez simple à comprendre : on a pas précisé le type des champs stockés, ce qu'on fera à l'usage
- On peut avoir plusieurs paramètres
- On utilise des patrons de classe depuis longtemps. Ex: `std::vector<int>`
- ultra puissant, mais source de messages d'erreurs incompréhensibles

2.2 Concepts C++20

- Pour l'instant, on ne contrôle pas trop ce qui rentre dans les templates. On peut faire des `const_assert` sur les traits des types comme dans le code présenté, mais c'est assez limité. Les `const_assert` sont très bien en général, mais pas adapté à ce cas.
- Avec C++20, on pourra poser des contraintes telles que "Comparable" ou "Decrementable" sur les types, et même déclarer ses propres contraintes en imposant aux types d'avoir des fonctions du nom+profil souhaité.

3 Conclusion sur le module

- L'objectif est de vous apprendre à faire des programmes non triviaux
  - Dans le projet, l'objectif était de réécrire le code à chaque question pour l'adapter aux nouvelles demandes
  - On ne va pas faire de vous des ingénieurs, mais les meilleurs chercheurs en biologie sont capables de remplacer leurs laborantins en cas de besoin.
  - Rapport à la recherche: usage. Je ne pense pas qu'il y ait des chercheurs en C++
- On a choisi le C++ pour cet objectif car c'est le langage roi
  - Il offre tous les paradigmes, avec des performances difficilement égalables
  - C'est le plus utilisé pour les tâches d'ingénierie compliquées
  - Il a un caractère de cochon. Quand on sait l'utiliser, on sait utiliser les autres.
- Place de ce module dans le cursus:
  - Prérequis de ce module: Le C pour les aspects pratiques, la théorie des langages de prog pour les réflexions sur les paradigmes
  - Suite du module: sorte de prérequis pour les modules utilisant Java comme langage (Jézéquel en M1)
  - Ce qui manque: méthodes de conception (merise/TDD), écrire des tests, programmation concurrente
- L'examen:
  - Feuille de pompe A4 recto verso de votre main
  - Très proche de ce qu'on a fait en TD/TP/Projet (écriture de petits codes sur papier, lecture de code fourni pour faire des schémas UML de l'organisation et/ou le schéma mémoire du tas et de la pile)