Chapitre 4: Below and Beyond C++

Martin Quinson

Épisodes précédents

- Héritage
  - Principe: factoriser du code, comme du copy/colle en mieux
  - Cacher l'arbre d'héritage est une bonne idée
  - Vocabulaire: mere/fille, spécialise/dérive/généralise
  - transtypage: surtout any `static_cast` et any `dynamic_cast`
- Polymorphisme
  - Principe de substitution de Liskov
  - Redéfinition méthode (override) dans fille != surcharge (overload) autre prototype
  - Liaison dynamique (type statique, type dynamique)
  - classe abstraite
  - Héritage multiple
- Design OOP, et UML
  - Composition (a des): flèche à tête de coté propriétaire
  - association (réciproque): trait avec la quantité en légende
  - Héritage (est un): flèche fille->mère
- Implémentation de l'héritage en mémoire
- Exceptions
  - Pas de new, pas de finally, on catch par référence

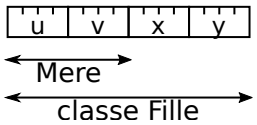
I) Implémentation de C++

- L'objectif est de comprendre comment plusieurs mécanismes de C++ sont implémentés, pour mieux savoir les utiliser.

I.1) Héritage d'objets

I.1.a) Objets triviaux

- Trivial en C++ = des choses facilement copiable, ie non-polymorphique
- On observe le code fourni "héritage en mémoire"
  - On a une classe mère et une classe fille, on initialise des instances de chaque
  - a2 est une instance de mère dans laquelle on range une fille
  - b2 est une instance de fille dans laquelle on range une mère, mais c'est interdit d'après Liskov.
  - Une classe plus spécialisée peu se faire passer pour une plus générique, mais pas le contraire.
- Qu'est ce qui se passe en mémoire ?
  - Le printf de la ligne 18 affiche `sizeof(a)=8; sizeof(b)=16; sizeof(a2)=8`
  - (sur mon ordi – `sizeof(int)=4` dans g++). Schéma mémoire correspondant:

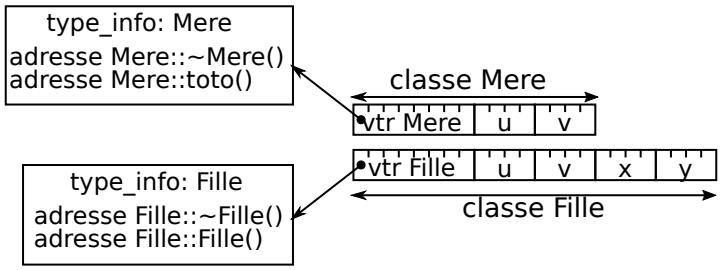


- La représentation mémoire est extrêmement efficace
  - les champs de la fille sont derrière ceux de la mère
  - C'est pour ça qu'on peut transtyper fille -> mère (avec une perte d'information)
  - Cela permet l'héritage sans aucun overhead, ce qui a fait le succès du C++ au début
- Et si héritage multiple ?
  - Les champs sont pris dans l'ordre d'héritage: Si C hérite de A et B, les objets C ont les champs A, les champs B puis les champs C.-
- Et les champs statiques ?
  - Placés à coté des globales, dans le segment texte du processus. Comme les locales statiques, finalement.
- Et si on ajoute des méthodes (non virtuelle) ?
  - Elles sont ajoutées au segment texte du processus, comme les fonctions C, avec des règles de mutilation (mangling) pour passer des noms C++ aux identificateurs C.

I.1.b) Objets non-triviaux

- Comment sont gérées les méthodes virtuelles?
- On regarde maintenant le code d'héritage polymorphique.
  - Qu'est ce que la méthode `toto()` affiche pour chaque objet?
    - `a.toto()` -> `Mere`, sans surprise
    - `b.toto()` -> `Fille`, sans surprise
    - `a2.toto()` -> `Mere` bien qu'on ait rentré un objet fille !!
      - C'est le type statique décide de l'appel, il n'y a pas de polymorphisme.
      - C'est comme la perte d'information liée au transtypage d'objets triviaux ci-dessus.

    * `a3->toto()` -> `Fille`, comme on s'y attendait au début : le type dynamique peut s'exprimer car il n'y a pas eu de perte d'info lors de l'affectation
  - Quelles sont les tailles des objets?
    * a:16, b:24, a2:16 (sur ma machine).
      · tous ces objets ont grossi de 8 octets par rapport à la version sans méthode virtuelle.
      · Des méthodes non virtuelles, n'auraient pas fait changer la taille: `sizeof(a)=8`
    * a3: 8. Preuve que les pointeurs sont de taille 8 sur mon ordi, et c'est la solution
      · On ajoute un pointeur par classe (pas de différence mère ou fille)
- Virtual method table
  - Pointeurs vers des tables stockées à part (mis en commun entre les objets de la même classe, probablement segment text)
  - Il y a bien une seule table virtuelle (sauf si héritage multiple). D'ailleurs, 24=16+8, pas 16+8*2



- – Les méthodes sont toujours dans le segment de code du processus
  - L'appel `a2->toto()` est compris comme `(* (a2->vtr[1]) )(a2)`
    * On cherche un pointeur sur fonction dans la table puis on l'invoque
    * On retrouve le `self` que Python rend explicite dans les méthodes de classe, qui est là dans la forme compilée
- Discussion et comparaison
  - Invoquer une méthdode virtuelle est plus lent qu'invoquer une méthode non virtuelle (un déréférencement de pointeur en plus, et un risque de prédiction de branche ratée), mais ça reste très efficace
    * On peut pas faire mieux sans JIT (prototypes de JIT dans LLVM)
  - Les choses sont un peu plus compliquées en cas d'héritage multiple, mais c'est très comparable
  - Explique que C++ ne permette pas d'ajouter des méthodes à la volée (par opposition aux langages à Duck-typing comme le Python ou Javascript): taille de vtable doit être connue à la compil
  - C++ ne permet que le simple dispatch (choix de la méthode en fonction du type d'un receveur unique), tandis que d'autres langages permettent le double dispatch (choix de la méthode d'un opérateur en fonction du type des deux opérandes), mais c'était pas implémentable avec une vtable simple comme ça.
- Et dans le cas d'héritage multiple? Il y a plusieurs vtables dans la classe
  - `https://prog.world/c-vtables-part-1-basics-multiple-inheritance/` présente une session gdb pour explorer

I.2) Décoration de noms

- En anglais, ça se dit "name mangling", dont la traduction litérale est plutôt "déformation, broyage". Perso j'aime assez "mutilation de noms"
- À l'origine, les compilateurs C++ étaient des transpilers, des compilateurs source-to-sources ciblant le C. Il fallait donc que les identificateurs respectent les règles C.

- Même ensuite, les éditeurs de lien ne fonctionnaient que pour le C, ce qui fait que le mangling a perduré
- Le C++ a bcp plus de fonctionnalités que le C (classes, namespace, surcharge, redéfinitions, etc).
- En particulier, on a souvent plusieurs fonctions de même nom en C++, alors que c'est impossible en C
  - L'éditeur de liens a donc seulement le nom des symboles dans sa table, ce qui force la décoration de noms à inscrire des meta-data dans le nom du symbole généré
  - namespace, classe, types des paramètres sont ajoutés autour du nom de méthode
- Il n'y a pas de norme, pour laisser liberté à l'implémenteur ou au cas où l'éditeur de lien comprendrait le C++. La norme incite même à utiliser une décoration différente afin de ne pas mélanger quand le reste de l'ABI ne correspondrait pas (exception handling, vtable layout, stack frame padding).
  - Mais certaines plates-formes ont standardisé l'ABI quand même, pour la compatibilité. En particulier, g++ et clang++ ont la même ABI. Quand elle change, ca complique la vie des distributions linux
  - 1) le nom commence par `_Z` pour dire qu'il est décoré
  - 2) S'il y a des namespace et classes, `Nnombre`. `N` pour signaler le fait, et le nombre pour donner la taille de ce bout. Termine par `E`
    Sinon, le nom de la fonction est seulement préfixé de la taille du nom (sans `N` ni `E`)
  - 3) le type des paramètres, avec `i` pour `int`, `v` pour `void` et `c` pour char
  - 4) modificateurs `const` et autres sur les paramètres
- Chose importante : le type de retour n'est pas dans la décoration, donc on ne différencie pas dessus
- On peut désactiver la décoration en mettant `extern "C" {...}` autour des symboles à ne pas décorer

I.3) Covariance et contravariance

- Quelle est la méthode invoquée, à la fin? On peut répondre maintenant que l'on comprend le layout memoire
- Variance d'un constructeur de type:
  - covariant: accepte des sous-types mais pas de super-types
  - contravariant: accepte des super-types mais pas des sous-types
  - bivariant: accepte à la fois des super-types et des sous-types
  - invariant: n'accepte ni super-types ni sous-types
  - Ca devient drôle quand on combine avec la spécialisation/héritage de collections (mais pas en C++)
- Affectation (paramètre, affectation variable): covariance pour respecter le principe de Liskov. On peut passer une Fille là où une Mère est attendue
  - Si c'est une copie d'objet, il y aura transtypage avec amputation
  - Si c'est un pointeur, on utilisera l'autre vtable
- Redéfinition:
  - Les paramètres sont invariants que ce soit copie d'objet ou passage de pointeur (sinon le mangling ne fonctionne plus)
    * C++ n'autorise pas les redéfinitions à spécialiser les paramètres : le compilo veille
    * Si on défini quand même des méthodes covariantes ou contravariantes, ce sont ne sont pas des redéfinitions mais des réécritures. On ne peut pas utiliser `override` sous peine de compil error. Une réécriture est comme une surcharge intergénérationnelle. Le twist est que la méthode de l'ancêtre est masquée et impossible à invoquer.
  - Le type de retour est covariant si c'est un pointeur, et invariant si c'est une classe

* Exercice: covariance de la feuille

```
struct A {
   virtual void boom()  { std::cout << "A::boom\n"; }
};
struct B : public A {
   void boom() override { std::cout << "B::boom\n"; }
};

struct Up {
   virtual A bidule()  { return A();     }
   virtual A* machin() { return new A(); }
   virtual void truc(A a)   { a.boom();  }
   virtual void chose(A* a) { a->boom(); }
   virtual void chouette(A a) { a.boom();  }
   virtual void muche(A* a)   { a->boom(); }
};
struct Down : public Up {
   // ERROR: invalid covariant return type for ‘virtual B Down::bidule()’
   //B bidule() override  { return B(); }
   // OK,
   B* machin() override { return new B(); }
   // ERROR: marked ‘override’, but does not override
   //void truc(B b) override { b.boom();  }
   // ERROR: marked ‘override’, but does not override
   //void chose(B* b) override { b->boom(); }

   void chouette(B b) { b.boom();  } // OK, mais c’est un overwrite
   // la méthode de l’ancêtre est masquée, réécrite. overload intergénérationel n’existe pas
   void muche(B* b) { b->boom(); } // OK

};

int main()
{  Up up; Down down; A a;  B b;

   up  .bidule().boom(); // A::boom
   down.bidule().boom(); // A::boom (pas de surcharge vu que c’est interdit)
   up  .machin()->boom();// A::boom
   down.machin()->boom();// B::boom
   std::cout << "------------\n";
   up.truc(a);   // A::boom
   up.truc(b);   // A::boom (b est amputé lors de l’affectation au paramètre)
   up.chose(&a); // A::boom
   up.chose(&b); // B::boom
   std::cout << "------------\n"; // down tout comme up: pas de surcharge réussie
   down.truc(a);
   down.truc(b);
   down.chose(&a);
```

```
50    down.chose(&b);
51    //down.chouette(a);// Up::chouette(A) masqué par réécriture, Down::chouette(B) utilisée.
52    down.chouette(b);  // Downcasting d'objets interdit en C++
53    //down.muche(&a);// Up::muche(A) masquée par réécriture, Down::much(B*) utilisée.
54    down.muche(&b);  // Downcasting de pointeurs toléré mais très mauvaise idée
55    return 0;
56 }
```

- Les templates: il faut tout faire soit-même
  - Invariant par défaut: les bouts de code générés n'ont pas de relation d'héritage entre eux
  - Mais on peut le faire en surchargeant `operator=()`
  - Les containers STL (vecteur, map, set) sont invariants, cf exemple
  - `std::pair<T *, U *>` et `std::tuple<T *, U *>` sont covariants
  - `std::shared_ptr` `std::unique_ptr` (sur lesquels on revient bientôt) sont covariants
  - `std::function<R *(T *)>` est covariant sur le return, et contravariant sur les paramètres...
  - `http://cpptruths.blogspot.com/2015/11/covariance-and-contravariance-in-c.html`
- On en sait maintenant assez pour comprendre le casse-tête de Beugnard
  - En C++, que des erreurs de compilations, rien au runtime. Autre histoire avec des `dynamic_cast`
  - Erreurs de compilations quand on généralise les paramètres vis-à-vis du type statique du receveur
    - `d.cov(top)` $\rightsquigarrow$ `Down::cov()` demande un `Middle*` et on tente une généralisation `Top*` $\Rightarrow$ error
    - `?.inv(top=` $\rightsquigarrow$ idem: `Middle*` attendu donc généralisation `Top*` refusée
    - `{u/ud}.contra( {top/mid} )` $\rightsquigarrow$ Même genre. `Bottom*` attendu, toute généralisation refusée
    - `d.contra(top)` $\rightsquigarrow$ `Down::contra()` demande `Middle*`, tentative `Top*`
  - Les colonnes `u` et `d` sont assez simples car type statique = type dynamique, donc liaison à la compil
  - Comment expliquer la colonne `ud` ?
    - Si le type dynamique s'exprimait seul, ça ferait `Down` tout le temps
    - Si le type statique s'exprimait seul, ça ferait `Up` tout le temps.
    - Pour comprendre la réponse, il faut se demander si les méthodes de `Down` sont `override` ou pas
    - Comme C++ n'autorise pas la variance, seule `inv()` accepte `override` quand tout est `virtual`
    - Conclusion: à la compilation, on détermine la méthode cible (son mangled name), et à l'exécution on cherche une méthode de cette signature là dans la vtable
    - au passage, si on retire "virtual" de `Up::inv`, le type dynamique ne peut pas s'exprimer car la méthode n'est plus dans la vtable, donc la méthode non virtuelle de `Up` s'applique même sur un type dynamique `Down`.
- Discussion
  - Il y a beaucoup de différences entre les langages OO : voir la page d'Antoine Beugnard pour les détails.
  - La sémantique C++ est faite pour permettre une implémentation aussi efficace que possible (invariance paramètres)
  - D'autres langages sont plus permissifs
    - Certains sont même contravariants: Eiffel et Dart autorisent à passer plus générique lors d'une affectation (sens inverse de Liskov), même si ça risque de poser pb si on utilise les champs spécialisés sur la généralisation passée à la fin. Ces langages lèvent une erreur runtime si le cas se présente

II) Gestion automatique de la mémoire

- Les pointeurs C sont un outil aussi puissant que dangereux à utiliser. On peut se tromper de 1000 façons avec les pointeurs, et le langage C++ apporte plusieurs mécanismes pour réduire les risques.
- On a déjà vu les références, qui sont des pointeurs dont on sait qu'ils sont valides (une référence ne peut pointer que sur un objet donné). En échange, on ne peut pas faire d'arithmetique des références comme on fait de l'arithmetique des pointeurs. On ne peut donc pas les prendre pour des tableaux.
- On va voir deux mécanismes pour éviter les fuites mémoire : RAII et pointeurs intelligents

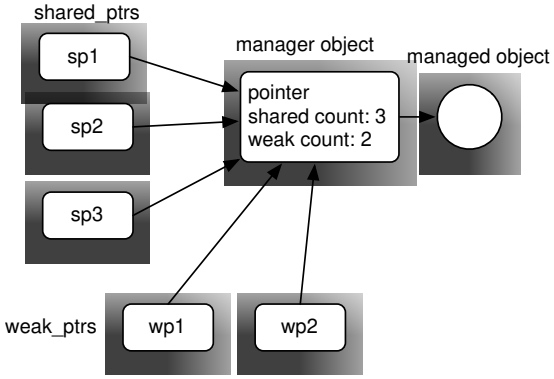
II.1) RAII (Resource Acquisition Is Initialisation)

- Il s'agit d'utiliser les objets automatiques pour ne pas avoir à gérer la mémoire à la main. Le compilateur invoque automatiquement le constructeur des **objets automatiques** quand ils rentrent dans le scope, et leur destructeur quand les objets sortent du scope. Par construction, ils sont vivants tant que la pile d'appel est dans ou au delà du cadre de pile où ils se trouvent. C'est très pratique
- Le principe du RAII est de créer des classes d'objet qui gèrent seules les ressources dans les constructeurs/destructeur pour ne plus avoir à s'en occuper.
- On regarde le motivating example, **code sans RAII** de la feuille d'exemples de code.
  - Le pointeur p est visiblement un espace de travail de cette fonction, alloué au début de la fonction, et libéré quand on sort. Ok, ça devrait marcher, mais on peut faire mieux.
  - (1) c'est pas si simple si y'a des exceptions. On peut tout mettre dans un gros try/catch, mais c'est affreux à lire et en plus ça devient très pénible s'il y a plusieurs ressources à éventuellement libérer.
  - (2) la dernière ligne est tellement obvious qu'on voudrait pouvoir s'en passer si possible.
- Le second exemple: **code avec RAII** fait ça. On n'a plus rien à faire à part créer la variable au début puis l'utiliser
  - Pour que ça marche, il faut un peu de magie C++, avec trois opérateurs dont un de conversion et deux pour "forwarder" les appels.
  - On est d'accord, cet exemple est sans intéret puisque `std::vector` ferait tout ça et bien plus. Mais bon.
- On peut faire du RAII pour la mémoire, mais pas seulement. Les fichiers vont se refermer tout seul de la même façon.

II.2) Smart pointers

- ce sont des objets prêts à l'emploi pour faire du RAII sur la gestion mémoire. Ils se chargent de faire new/delete automatiquement quand on n'en a plus besoin. Il en existe 3 sortes.
- `std::unique_ptr` c'est exactement l'implémentation du RAII vu avant. On peut le mettre en haut de méthode, où il fait new. On pourra pas déplacer son contenu facilement, et il fera delete à la fin. C'est super, mais on ne peut pas les déplacer ou les retourner à l'appelant. La ligne 15 du code d'exemple lève une erreur de compilation.
  - D'ailleurs, comment est-ce réalisé ? Simple: l'opérateur d'affectation est marqué `delete`
- `std::shared_ptr` c'est la même chose mais on peut avoir plusieurs "références intelligentes" sur le même objet, qui n'est détruit que quand la dernière référence est détruite.
  - Comment ça marche? Simple refcounting.
  - Il y a un objet intermédiaire qui (1) garde la référence à l'objet géré (2) compte le nombre de références.

- Les références modifient l'objet intermédiaire dans leur ctor/dtor, et la dernière ref détruit l'objet intermédiaire ainsi que l'objet géré



  - Bien evidement, si on fait n'imp (comme retourner le pointeur nu dans l'objet), le smart pointer ne peut plus nous protéger
  - En fait si on fait `make_shared`, on n'a qu'un seul malloc, assez grand pour contenir à la fois le gestionnaire et l'objet géré
  - Les opérateurs `*` et `->` sont bien définis pour qu'on puisse utiliser le smart ptr comme une référence. (CODE: heritage et smart pointers)
    * Les opérateurs `==` et `!=` sont surchargés pour tester le pointeur géré
  - On peut faire un alias de type pour ne pas écrire `std::shared_ptr<MyType>` à tout bout de champ: `using MyTypePtr = std::shared_ptr<MyType>;`
    * On pourrait aussi faire un typedef, mais les alias sont conseillées en C++ car certaines choses avancées sont possibles avec using mais pas avec typedef (quand on fait du templating avancé)
- **Héritage et smart pointers**
  - Si l'affectation entre objets est OK, alors l'affectation entre smarty l'est (ie, les smart pointers sont covariants)
  - Les transtypages nécessitent un outil: `std::static_pointer_cast`, qui est une template de plus fournie par la STL
- `std::weak_ptr` pour lutter contre les références cycliques. Elles pointent sur l'objet pour retrouver l'objet géré, mais ne participent pas au refcounting. En pratique, c'est pas très pratique non plus. Il faut juste savoir que ça existe.
- Conclusion: avec les smart pointers, on n'écrit plus jamais new/delete soit-même, et les fuites mémoires sont un souvenir ancien.
  - Note: `std::make_unique()` dont y'a besoin pour ne même pas écrire `new` à l'initialisation est C++14, mais il devient très difficile de trouver un compilateur qui ne comprenne pas le C++14, de nos jours.

II.3) Déplacement mémoire

- On veut parfois déplacer un `unique_ptr`, ou bien passer l'ownership d'un `shared_pointer` sans jouer le jeu du refcounting (pour gagner du temps)
- `std::move()` permet de faire juste ça. Après le move, le pointeur original est mis à `nullptr` (comme s'il avait fait release)
  - D'ailleurs ce n'est pas limité au smart pointer, mais ça permet d'informer le compilateur du flot de durée de vie des objets afin qu'il puisse optimiser et éviter des copies.
  - Quand une fonction crée une chaine avant de la renvoyer, la copie est une perte de temps puisque

l'original va être détruite.
  - C'est un sujet complexe, et je ne suis pas sûr de tout comprendre moi-même

II.4) Référence rvalue

- Une référence rvalue (de type `T&&`), c'est comme une référence habituelle (dite lvalue) dont on ne peut pas changer la valeur.
  - En pratique, c'est une référence dont le propriétaire me donne l'ownership
  - C'est donc pas une qestion de const ou non, mais plutot de mouvement mémoire. On fait ça quand on veut informer le compilo pour qu'il optimise les copies mémoires inutiles.

II.5) Règles informelles pour mieux programmer en C++

II.5.a) Utiliser le compilateur à son avantage

- Activer un max d'options de warning, et compiler en `-Werror` (comme en C, quoi)
- Utiliser des outils d'analyse statique comme `clang-tidy`, le Clang static analyzer, Sonar ou LGTM (les deux derniers sont des programmes privateurs).

II.5.b) Autres conseils divers

- préférer les initialisations avec des accolades
- marquer "override" les fonctions redéfinies
- Google: utiliser les références qu'en const pour éviter la copie mémoire, mais préférer les pointeurs pour ce qui doit être modifié (afin de mieux voir ce qui peut être changé coté appelant)
- La plupart des conseils du livre de Scott Meyer (Effective Modern C++) s'y prete bien

II.5.c) Règles de 3, de 5 et de 0

- C'est un truc de programmeur C++98 comme le DRY/SPOT pour éviter les erreurs de conceptions sachant que le compilo crée des méthodes spéciales quand ça lui chante
- La règle est : si une classe définie l'une des trois méthodes suivantes, elle doit définir les trois: destructor, copy constructor, copy assignment operator
  - On a besoin de ces méthodes quand y'a un état non scalaire, comme des choses sur le tas ou des fichiers ouverts.
- Avec le C++11, on complète la règle avec deux méthodes supplémentaires: move constructor, move assignment operator qui ne font pas une copie profonde, mais détruisent l'objet de départ
- La règle de 0 dit que les classes qui ont l'une des 5 méthodes ne devraient avoir que celles-ci. Separation of concern, elles ne s'occupent que de la gestion de la mémoire.
  - Les autres méthodes ne devraient avoir aucune des 5 méthodes (elles sont marquées `=delete`).
  - Mais bon, c'est un mantra de programmeur, faut pas en faire une religion.