Table des matières

Philosophie et frontières

Pourquoi ce volume?

Après avoir couvert les bases de ce qu'est un objet [POOv00], les mécanismes et concepts clés que sont l'héritage, le polymorphisme et l'abstraction [POOv01] et après avoir examiné des thématiques plus avancées comme la programmation générique [POOv02], nous réfléchirons maintenant sur la POO pour poser un jugement critique sur ses avantages et ses inconvénients, et nous examinerons des manières sophistiquées d'appliquer la POO à des problèmes réels. Nous explorerons aussi un certain nombre de thématiques plus avancées, mais tout de même importantes, à propos de la POO contemporaine.

Ce volume propose une réflexion sur la POO et ses applications sous plusieurs aspects, principalement à l'aide de C++ mais aussi avec Java, C# et VB.NET.

Certaines thématiques avancées ne sont pas possibles avec tous les langages OO, alors nous ne pourrons pas systématiquement transposer chaque idée dans plusieurs langages.

On y verra, entre autres choses :

- **les bons et les moins bons aspects de la POO**. La POO est un outil merveilleux mais ce n'est pas une panacée; mieux vaut être capable de choisir le bon outil pour chaque tâche. Nous examinerons aussi **ce que signifie être OO**, un sujet susceptible de générer bien des débats, et nous essaierons de voir **comment transposer en technique dans un langage OO ce qui est un concept dans un autre**;
- comment **tester et mesurer dans un contexte OO**. Après tout, on ne peut en même temps prôner la réutilisation massive des objets sans réexaminer la manière par laquelle on mesure la vitesse avec laquelle est développé le logiciel;
- nous étudierons le problème très complexe de la **persistance des objets**, de même que celui (subtil mais plus accessible) du **clonage d'objets**;
- nous ferons un examen plus en profondeur des **divers conteneurs standards** à notre disposition, allant au-delà du survol, sommaire et applicatif, fait dans [POOv02], et examinant au passage des stratégies pour choisir le bon conteneur pour chaque tâche;
- nous verrons comment **prendre le contrôle de la mécanique d'allocation dynamique de la mémoire**, abordant (brièvement) au passage le concept d'**allocateur**. Ce sujet, bien que très technique, nous donnera une perspective plus riche sur le modèle OO et sur les mécanismes fondamentaux qui le sous-tendent;
- nous examinerons une **application du modèle OO** à la fois simple et plus complexe que celles proposées dans les volumes précédents, pour voir comment peuvent interagir les diverses stratégies couvertes dans ces quelques volumes et pour jeter un coup d'œil sur la **mise en commun de stratégies OO et de schémas de conception**;
- nous verrons comment il est possible de procéder à l'**internationalisation** de programmes à l'aide de stratégies OO;
- nous examinerons des thématiques conceptuelles novatrices mais qui apportent d'importantes opportunités d'optimisation et une plus grande flexibilité à la rédaction de classes de grande qualité, soit la **sémantique de mouvement** et les **références sur des *rvalue***, introduites sommairement dans [POOv02];
- nous verrons sous une lueur nouvelle le concept de **conversion explicite de types**;

- nous essaierons de définir ce que signifie **être évaluable**;
- nous verrons ce qu'est la **réflexivité**[1], soit la capacité de décrire un objet en ses propres termes, de même que ce qu'est une **métaclasse**, dont les instances sont des classes;
- nous examinerons quelques **applications de la programmation générique**, en particulier la technique des **traits**;
- nous examinerons aussi les **concepts**, destinés à représenter des contraintes applicables à un type générique. Bien que ce volet de C++ 11 ait été remis à plus tard, faute de maturité suffisante, il demeure qu'il s'agissait d'une des plus importantes avancées à l'agenda pour ce standard, et les travaux à son sujet se poursuivent dans l'espoir que le prochain standard permette de mettre les concepts de l'avant; enfin
- nous aborderons une approche orthogonale à la POO en dissertant brièvement sur ***la programmation orientée aspect,*** ou POA.

Schémas de conception et idiomes de programmation

Ce document, tout comme les documents qui le suivront, présentera un certain nombre de schémas de conception et d'idiomes de programmation. Ces deux termes désignent des pratiques reconnues et répandues, à un point tel qu'elles en sont venues à porter un nom.

⇒ Un **schéma de conception** (en anglais : *Design Pattern*) est une pratique applicable à la vaste majorité des langages de programmation (typiquement, des langages OO).

⇒ Un **idiome de programmation** est une pratique applicable aux langages de programmation respectant certaines prémisses. Pensez à l'idiome RAII, qui requiert le support à des destructeurs déterministes.

Nous identifierons ces pratiques lorsque nous les rencontrerons. Les schémas de conception et les idiomes de programmation représentent un langage commun dans le monde du développement logiciel, une codification des pratiques. Reconnaître et savoir nommer ces pratiques est un atout pédagogique qu'il ne faut pas négliger.

[1] J'utilisais auparavant le terme *Réflexion*, un calque de l'anglais, mais un chic étudiant du nom de ***Richard Gilbert*** m'a mis sur la piste d'un nom plus approprié. Merci!

À propos de la forme

Vous trouverez à divers endroits des petits encadrés indiquant ***Réflexion 03.n*** (pour divers entiers ***n***). Ces encadrés soulèveront des questions qui méritent une discussion plus approfondie et pour lesquelles les réponses peuvent surprendre ou ne pas être aussi banales qu'il y paraît. Des ébauches de réponses seront proposées pour chacune dans ***Annexe 01 – Discussions sur quelques réflexions*** choisies.

Puisque ces notes se veulent un appui à l'apprentissage de la POO, pas d'un langage particulier, mais puisqu'il faut aussi utiliser (au moins) un langage pour appuyer formellement nos idées et notre discours, le cœur de ce document utilise un langage OO, C++, pour ses exemples. Cependant, vous trouverez à certains endroits dans le document des sections intitulées ***Dans d'autres langages*** qui exploreront les ramifications des sections précédentes en Java, C# et VB.NET, ce qui vous permettra de faire le pont avec d'autres technologies. Ces pages ont des bordures différentes des autres, et vous pourrez les omettre si ces nuances ne sont pas au cœur de vos préoccupations.

Enfin, notez qu'au moment d'écrire ces lignes, la norme la plus récente du langage C++ est **C++ 14**, un ajustement à l'énorme mise à jour que fut C++ 11, le vote pour officialiser C++ 17 (la prochaine mise à jour significative du langage) est en cours, et les travaux sur ce qui devrait être C++ 20 vont bon train. Les volumes plus poussés de cette série de notes de cours couvrent des aspects des plus récentes versions de C++, et montrant parfois comment il est possible d'en arriver au même résultat à partir de versions antérieures du langage, ou en indiquant les raisons qui motivent certaines adaptations de la norme.

Remarques de grande personne

Le document que vous examinez présentement contient des trucs, des conseils, des techniques et des concepts (parfois littéralement).

Sachez dès maintenant qu'il n'existe aucun substitut de qualité pour la réflexion personnelle (et approfondie) et l'esprit critique. Devenir un programmeur Ferengi[2] n'a pas que des inconvénients (c'est bien mieux que de ne pas réfléchir à sa propre pratique, entre autres choses), mais il demeure que l'acte de programmer est en partie scientifique, en partie artisanal, et en partie esthétique (eh oui!).

Conséquemment, lisez ce qui suit en ayant à l'esprit que oui, l'auteur sait ce dont il parle, mais oui, c'est de *votre* lecture, de *votre* esprit, de *votre* appréhension du monde et du processus créatif de développement qu'il s'agit, en fin de compte.

Le monde est plein d'horreurs conceptuelles. On n'a qu'à penser à :

- l'interface trop abstraite[3] (à un certain stade, on perd notre temps, à moins d'avoir vraiment quelque chose à dire);
- l'interface trop riche[4], qui ressemble à un couteau suisse. Retenez qu'un objet devrait faire une chose et bien la faire; trop en faire n'augmente pas l'utilité d'un objet, mais la réduit en le liant de trop près à un domaine d'application spécifique;
- des abus d'enrobage[5]; et…
- …des abus d'enrobage déjà abusifs[6].

Soyez éveillé(e). Vous avez votre rôle à jouer. Votre voix compte. Agissez de manière responsable. Les gens capables de jongler à ce niveau d'abstraction et de bien le faire sont trop peu nombreux; chaque personne compte.

[2] http://www.codinghorror.com/blog/archives/001225.html

[3] http://thedailywtf.com/Articles/I_Object.aspx

[4] http://thedailywtf.com/Articles/Implements_ISwissArmyKnife.aspx

[5] http://thedailywtf.com/Articles/Class_IntWrapper.aspx

[6] http://thedailywtf.com/Articles/A_Wrapper_Wrapper.aspx

À propos du modèle OO

Le modèle OO est un modèle intéressant à plusieurs égards, mais il n'est évidemment pas parfait (quel modèle le serait?), et il faut réfléchir à certaines de ses carences. Étrange, sans doute, pour des notes destinées à accompagner un cours de POO que de mettre en relief que la POO est *une* approche valable, pas *la seule* approche valable, pour aborder plusieurs problèmes.

Dans bien des cas, la POO s'inscrit dans un amalgame complexe de stratégies et permet de faciliter la définition de schémas de conception, d'algorithmes génériques, d'idiomes particuliers et ainsi de suite. Nous ne voudrions sans doute pas revenir en arrière, vers une époque où la POO était à la fois peu connue et peu comprise, mais avec la maturité vient le recul. Mieux vaut comprendre les forces et les faiblesses d'une approche pour l'utiliser au maximum de son potentiel que de se faire croire que le marteau doré[7] a enfin été trouvé.

L'un des informaticiens les plus célèbres de l'histoire de cette jeune science, **Edsger W. Dijkstra**, aurait envoyé[8] à propos de l'approche OO :

« *Object-oriented programming is an exceptionally bad idea which could only have originated in California.* »

Merci à mon collègue et ami **Vincent Echelard** pour cette perle.

La POO est d'une utilité inestimable mais il ne s'agit pas d'une panacée, et il ne faut pas chercher à l'aborder (ou à la vendre) comme telle.

La POO est complexe à définir de manière universelle

Il est difficile de définir ce que signifie *être OO* de manière satisfaisante pour tous les intervenants impliqués dans le domaine et intéressés par le sujet. La publicité est imposante, les intérêts commerciaux sont omniprésents et les dogmes s'entrechoquent fréquemment.

La section ***Être OO*** (plus bas) propose un tour d'horizon des caractéristiques propres à divers modèles OO, car il y en a plusieurs, et discute (de manière succincte) de quelques tenants et aboutissants des divers choix philosophiques possibles. Malgré les meilleurs efforts des penseurs, définir ce que signifie être OO entraîne presque inévitablement des guerres de religion. La réponse des uns comme des autres tend à être, en résumé : *être OO, c'est faire ce que mon langage préféré* (ou *ma plateforme favorite*) *permet de faire*. Exprimé autrement : « les lacunes de mon modèle de prédilection sont moins importantes que ses forces ».

Selon le poids accordé aux diverses caractéristiques jugées importantes ou essentielles au modèle OO par une plateforme ou un langage donné, le modèle OO résultant offrira des avantages et des inconvénients différents. La question est donc guidée par les besoins d'entités commerciales et de groupes d'intérêt soucieux de voir leur manière de penser (et la couverture de leurs besoins propres) dominer le portrait et prendre force de loi[9].

---

[7] Celui qui fait en sorte que tous les problèmes ressemblent à des clous. Pour paraphraser **Alisdair Meredith** dans https://twitter.com/AlisdairMered/status/523628373768028160 : « …mais quand ce marteau est C++, tous les problèmes ressemblent à des doigts ».

[8] Voir http://thinkexist.com/quotation/object-oriented-programming-is-an-exceptionally/375335.html

[9] Les choix pédagogiques des professeurs et des maisons d'enseignement teintent aussi, clairement, cette définition.

Le principe SOLID

Plusieurs associent l'approche OO non pas à des concepts langagiers mais bien à un groupe de « principes de sain design ». Ce groupe de principes est associé à l'acronyme anglais SOLID, un acrostiche pour les principes suivant :

- *Single Responsibility* : une vocation pour chaque chose (objet, classe, fonction);
- *Open / Closed* : ouvert pour extension, fermé pour modification;
- *Liskov Substitution* : les classes dérivées respectent les mêmes invariants que leurs classes parents;
- *Interface Segregation* : préférer les interfaces spécifiques aux interfaces trop générales;
- *Dependency Inversion* : mieux vaut dépendre des abstractions que de leurs réifications.

La difficulté de déterminer ce qui doit être considéré essentiel ou fondamental au modèle OO fait partie du groupe de facteurs qui rendent difficiles la conception de SGBD OO vraiment universels, et qui compliquent la conception de métalangages capables de rejoindre l'idée d'objet de manière telle qu'ils pourraient servir de pont technologique entre les diverses philosophies.

Nous reviendrons sur ces questions dans la section ***Sérialisation et persistance*** et dans la section ***Concepts ou techniques*** qui proposent des stratégies pour remplacer par des techniques de programmation certains concepts pouvant ne pas être couverts dans un langage OO ou l'autre.

La POO ne propose pas toujours la meilleure approche

Pris au sens strict, le modèle OO ne se prête pas à certaines opérations purement algorithmiques et dissociées de toute subjectivité. Une opération comme `min(a,b)`, par exemple, est plus une fonction appliquée aux paramètres `a` et `b` qu'une méthode de `a` ou de `b`.

Les langages se voulant si purement OO qu'ils ne permettent pas même de créer des fonctions globales doivent appliquer des rustines (des *patches*) à leur modèle conceptuel, souvent par l'emploi de méthodes de classe un peu bidon, pour exprimer des idées aussi simples. Dans une expression comme `Math.min(a,b)`, l'utilité de `Math` en tant que classe est discutable[10] – entre autres, est-ce vrai que l'algorithme `min()` est de nature strictement mathématique?

Un de mes anciens étudiants, **Pierre Boyer**, a relevé, dans son essai de maîtrise, que l'approche OO est souvent moins intuitive que l'approche procédurale pour des programmeuses débutantes ou des programmeurs débutants. En effet, le monde est peuplé d'objets inanimés sur lesquels les individus opèrent; l'approche OO tend à nous faire penser ces objets en termes d'entités actives (à la limite *autonomes*) avec lesquels nous dialoguons.

Admettons qu'il paraît plus simple, pour la majorité des gens, de penser tourner les pages du journal que de demander poliment au journal de tourner ses propres pages.

[10] En Java, version `1.6`, tous les membres de la classe `Math`, sans exception, sont des membres de classe… La pertinence de l'introduction d'une classe dans un tel cas se limite au regroupement des opérations et des constantes sous un même chapiteau, mais la classe `Math` hérite de tous les membres de la classe `Object`, son parent, incluant des outils de synchronisation et de sérialisation qui, pour elle, sont superflus. Un paquetage ou un espace de noms suffirait amplement.

La POO ne se prête pas toujours au format de documentation le plus utile

Le fait que le modèle OO demande une adaptation dans le mode de pensée des gens venant d'un monde structuré implique aussi une modification des modes de recherche de ces gens. La documentation générée par des produits comme *Javadoc*[11] ou *Doxygen*[12] se présente sous une forme « classe par classe », et demande donc de réfléchir d'abord *à qui*, puis *à quoi*.

Dans certains cas, ce saut se fait simplement. En Java par exemple, plusieurs penseront d'abord à la classe `String` pour chercher la liste des constructeurs en quête d'un constructeur permettant, par exemple, de créer une chaîne contenant `50` fois le symbole `'*'`.

Quand, par contre, le *quoi* n'est pas évident, la question de savoir chercher efficacement la bonne classe ou la bonne méthode devient une question beaucoup plus difficile, s'adressant en fait d'abord à des *initié(e)s* du modèle OO et du langage choisi.

Des cas en apparence simples comme chercher la méthode permettant de concaténer deux chaînes en Java demandent des connaissances de niveau méta quant au langage, à savoir que la classe `String` y est immuable, et que la concaténation demande de créer de nouvelles chaînes ou de passer par une autre classe (`StringBuffer`), au choix. Même pour des informaticien(ne)s d'expérience, ce bond est complexe et chargé de culture locale. Le monde `.NET` n'a pas fait mieux (classes `String` et `StringBuilder`).

Il y a évidemment des raisons pour cette dualité, qui ont trait à l'encapsulation en l'absence de constantes par instance dans ces langages. Référez-vous à l'***appendice 01*** de [POOv01] pour des détails.

Dans des cas où l'équipe de développement cherche à réaliser des opérations, la documentation *basée classe* de la POO typique tend à forcer la recherche sur une base structurelle, à partir du nom de la classe alors qu'en fait, c'est souvent là l'inconnue.

À tout hasard, illustrez par vous-mêmes cette situation en examinant l'aide en ligne standard de Java (qui est en soi un excellent outil pour la majorité des cas) en cherchant à trouver le bon type de panneau ou le bon type de zone de texte[13] pour permettre l'édition de code colorié. Il y en a plusieurs, et certains sont meilleurs que d'autres, mais la recherche par opération n'est pas naturelle et vous devrez plutôt chercher par nom de classe et par parenté.

Avis aux intéressé(e)s : la recherche dans MSDN pour le même type d'information n'est pas plus simple. Essayez de trouver comment faire en sorte qu'un programme puisse démarrer un programme externe à partir des outils de l'infrastructure `.NET`… Avez-vous les mêmes intuitions que les concepteurs de cette infrastructure?

[11] http://java.sun.com/j2se/javadoc/

[12] http://www.stack.nl/~dimitri/doxygen/

[13] … et je vous aide ici en vous donnant un indice sur les types de contrôles les plus susceptibles de vous être utiles; la recherche est plus ardue quand on ne possède pas au préalable cette piste.

La POO dépend d'une relation implicite de confiance

La POO repose sur un contrat de confiance entre les développeurs d'une classe, d'une hiérarchie de classes ou d'une bibliothèque toute entière, et les utilisateurs de ces classes. Cette confiance est, dans la majorité des cas, méritée :

- on s'attend d'un objet, par respect du principe d'encapsulation, à ce qu'il assure sa propre intégrité du début à la fin de son existence. Un objet peut habituellement offrir cette garantie grâce à ses constructeurs et à son destructeur, du moins pour les langages qui supportent pleinement cette symétrie. S'il n'offre pas de finalisation déterministe, comme c'est le cas avec Java ou avec les langages .NET, la pleine encapsulation n'est possible qu'avec l'aide du code client qui devra lui donner explicitement l'opportunité de procéder à ses opérations de nettoyage en invoquant des méthodes prévues à cet effet (typiquement une méthode `Dispose()` ou une méthode `close()`). Cela dit, heureusement, Java et C# vont maintenant dans cette direction;
- en particulier, dans pratiquement tous les langages OO, un objet rigoureux sera sécurisé dans un contexte de multiprogrammation car, à travers l'encapsulation il contrôlera les accès à ses états et gérera une part importante des problèmes potentiels de synchronisation le touchant. Un compilateur de qualité devrait permettre de réduire (sans les faire disparaître) les coûts associés à cette protection sur les « performances » du code client;
- un objet devrait rapporter les cas exceptionnels survenant lors d'opérations tombant sous l'égide de l'une ou l'autre de ses méthodes. Ainsi, utiliser un objet ne devrait pas empêcher le code client d'avoir accès à un diagnostic plein et entier de la situation, et ce en tout temps;
- un objet devrait gérer pleinement les considérations de confidentialité et de sécurité qui gravitent autour de ce qu'il représente; *etc.*

Cela dit, indiquer qu'un objet assure sa pleine et entière intégrité, qu'il garantit le respect de ses invariants, ou encore indiquer qu'il assure la stabilité de ses états du début à la fin de sa vie, n'est pas toujours chose suffisante. Il arrive en effet qu'on veuille tenir compte de l'état initial d'un objet et qu'on se demande *ce que représente un état stable* pour un objet donné.

Quelques exemples concrets :

- un flux nouvellement créé est-il nécessairement ouvert?
- une chaîne de caractères par défaut a-t-elle un contenu de taille zéro?
- quel est l'état d'une matrice par défaut? S'agit-il d'une matrice plein de zéros ou d'une matrice identité?
- devrait-on fermer explicitement chaque fichier ou se fier au destructeur d'un objet représentant ce fichier pour réaliser cette tâche[14]?

La maxime de **Scott Meyers**, à l'effet qu'il soit sage de rendre simple les opérations saines et difficiles les opérations qui ne le sont pas, s'applique ici comme ailleurs. Une bonne interface devrait nous inviter à travailler correctement.

De même, nous devrions examiner les possibilités que nous offre un objet avant de s'en servir : par exemple, si un objet expose une méthode permettant de savoir s'il est vide, nous devrions la préférer à une approche indirecte pour en arriver au même constat (par exemple, vérifier si sa taille est zéro), car de manière générale, la présomption du code client devrait être que, si un objet offre un service, alors ce service est probablement au moins aussi bien implémenté à l'interne que ce que nous pourrions faire nous-mêmes de l'extérieur.

Ici comme ailleurs, il y a bien sûr des nuances…

[14] La réponse varie d'un langage à l'autre, ce qui complique l'écriture de bibliothèques accessibles à travers du code client écrit à l'aide de plusieurs langages de programmation distincts.

Les questions sont nombreuses et trouvent toutes une réponse valide et (typiquement) unique dans un langage de programmation donné ou pour une bibliothèque donnée. Le problème, surtout pour qui débute en POO, est que chaque réponse est en grande partie *culturelle*, et que l'état initial, tout comme les actions finales d'ailleurs, attendus ou présumés d'un objet donné sont en général facilement devinés par les programmeurs OO d'expérience, mais le sont beaucoup moins facilement par les programmeurs OO occasionnels ou récents.

L'informaticien(ne) doit faire confiance aux objets, mais pour ce faire, il lui faut les comprendre et bien saisir leurs rôles et responsabilités. Avec l'encapsulation stricte (et avec l'héritage, le polymorphisme, l'abstraction et ainsi de suite), la véritable confiance ne se peut qu'à l'aide d'une documentation très claire.

Une erreur dans l'utilisation d'objets clés[15] peut mener à des problèmes importants. Une classe exigeant que ses ressources soient libérées explicitement, comme c'est souvent le cas avec des objets .NET ou Java[16], mènera à des programmes dont l'exécution entraîne une dégradation des ressources disponibles si un programmeur se trompe dans sa compréhension du comportement attendu de ses instances.

Documenter rigoureusement les méthodes clés d'un objet diminue le niveau d'incertitude dans chaque cas, mais on souhaite que l'utilisation d'objets soit *maximalement intuitive*. C'est là un but souhaitable mais qui n'est qu'approchable, pas atteignable, dans l'absolu.

La POO est plus riche que l'usage qu'on tend à en faire

Les informaticien(ne)s tendent à avoir recours à des recettes qui ont fait leurs preuves. Les schémas de conception et les idiomes de programmation sont des illustrations claires de la valeur intrinsèque de cette habitude. Comme le diraient les anglophones : *If it ain't broke, why fix it?*

Comme dans toutes les approches de programmation, cela dit, la créativité et l'innovation ont leur place, et une place importante par-dessus le marché. Il existe souvent plusieurs manières de résoudre un problème, de concevoir une solution, de rédiger une classe ou un algorithme.

L'enseignement de la POO, surtout en début de parcours, tend à proposer les recettes, les façons de faire éprouvées. L'industrie s'attend, chez l'informaticien(ne) en début de carrière, à un vocabulaire qui comprend ces recettes, et s'attend à ce que ces informaticien(ne)s soient capables de les appliquer. L'informaticienne moyenne et l'informaticien moyen comprendront ces recettes et les mots par lesquels elles seront exprimées. Faciliter la communication et la compréhension est productif. Vous avez sûrement déjà remarqué que, plus nous progressons dans cette séquence de documents, et plus nous nous écartons de certains principes un peu simplistes énoncés au tout début de notre cheminement.

On sous-utilise la POO. Trop peu nombreuses et trop peu nombreux sont celles et ceux qui ont le réflexe de penser à plusieurs petits objets collaborant entre eux plutôt qu'à des hiérarchies un peu trop chargées de classes qui accumulent un bagage trop lourd pour leurs besoins.

---

[15] Ceux représentant les entrées/ sorties ou les ressources externes à un programme sont de bons cas d'étude.

[16] … à cause de l'emploi de moteurs de collecte automatique d'ordures et, en Java, à cause de l'absence de véritables mécanismes de destruction.

L'objet idéal est petit, souvent même vide, et représente un concept clair et univoque. L'héritage et le polymorphisme sont des outils puissants et importants, mais l'association, la composition et l'agrégation (je dirais même plus : la *collaboration*!) sont plus importants encore. Penser petit pour faire de grandes choses.

Plus une classe fait de choses et moins elle est utile, du fait qu'elle est alors fortement associée à un domaine spécifique et, par conséquent, utilisable dans un moins vaste éventail de champs d'application. Une bonne classe, en général, fait peu de choses, mais ce qu'elle fait, elle le fait bien.

L'équilibre entre pédagogie traditionnelle de l'approche OO (enseigner ce que tout le monde dans l'industrie sait et s'attend à ce que l'on sache) et approche OO saine (penser petit, modulaire, assembler, déterminer les relations) est difficile à trouver. Même dans ces notes, malgré la conscientisation de l'auteur, les classiques et la tradition ont servi de rampe de lancement. Construire, puis déconstruire pour mieux reconstruire.

Le présent document, tout comme ceux qui suivent (pour celles et ceux qui pousseront plus loin leur réflexion), déconstruit dans plusieurs directions. À certains égards, ce qui est contenu dans les sections subséquentes peut être choquant, être hors normes.

L'approche OO est un outil intellectuel d'une puissance inouïe, mais la créativité qui mène à en tirer le maximum est difficile à enseigner et à proposer en milieu industriel. La résistance est forte pour qui souhaite sortir du cadre.

Être OO

Ce qui suit traite de l'approche OO dans une acception large et multilingue; pour appuyer ce point, voici la définition que C++ donne d'un objet :

« *An object is a region of storage* » ([intro.object] dans le standard)

Cette définition sied bien C++. Visiblement, nous serons plus exigeants dans ce qui suit.

En C++, particulièrement, l'avènement des concepts (voir ***Contraintes et concepts***) influencera une bonne partie de la réflexion ci-dessous.

Le titre de cette section pourrait aussi s'écrire sous une forme interrogative, soit *Être OO*? Cette section semble simple, mais n'est pas banale. En effet, après avoir suivi un cours de POO, pouvons-nous déterminer ce que signifie *être OO*? Pouvons-nous délimiter les frontières de ce que signifient des idées comme :

- le langage XYZ est un langage *pur OO*;
- le langage XYZ est un langage *OO hybride*;
- le langage XYZ est *plus OO que* le langage ZYX?

Pourtant, ces formules font office de boulets dans plusieurs guerres de mots, qui sont trop souvent des guerres idéologiques, dont les fondements tendent à être boueux, arbitraires.

En fait, les idées associées à *être OO* sont nombreuses, et aucun langage – à ma connaissance – ne les embrasse toutes pour le moment. En fait, dans certains cas, introduire l'une de ces idées résulterait en une forme d'antagonisme face d'autres, alors que chacune, prise individuellement, seraient qualifiée d'idée importante du modèle OO par bien des expert(e)s.

Le problème que constitue définir ce que signifie *être OO* contribue à la complexité du problème de la persistance des objets.

Nous reviendrons sur cette question dans une section ultérieure.

Vous remarquerez, à la lecture de cette section, que ce document fait des choix parmi ceux proposés dans la liste des idées OO qui suit. Ces choix sont pragmatiques, et sont guidés à la fois par un fort sens des priorités (l'encapsulation et le polymorphisme comme fondements les plus importants de l'approche OO) et d'esthétique.

Malgré l'évidence de l'énoncé, soulignons que ces choix sont, d'abord et avant tout, des *choix*; un autre professeur, guidé par d'autres critères, aurait pu faire des choix différents dans certains cas. N'en faites pas, je vous prie, une guerre de clochers.

Chaque sous-section ci-dessous décrit une condition *nécessaire*, au moins aux yeux de certain(e)s expert(e)s, pour être OO, mais aucune n'est *suffisante* lorsque prise isolément. La matière à débats est de déterminer le sous-ensemble qui sera à la fois nécessaire et suffisant, ce qui n'est pas une mince tâche. Évidemment, cette section est écrite à la première personne, alors elle exprime la vision de l'auteur et n'engage que lui.

L'objet comme unité organisationnelle

La première caractéristique fondamentale d'un langage OO est qu'il doit permettre de **regrouper, *dans une même entité*, les données et les opérations sur ces données**.

Que le modèle OO propose d'abord et avant tout un formalisme rigoureux du concept de type est un indice fort à l'effet qu'un bon modèle OO devrait offrir un système de types solide et opérationnellement homogène.

La définition traditionnelle d'un type de données (pas nécessairement d'un objet au sens contemporain du terme) nous dit qu'un type de données est fait, comme son nom l'indique, de données et d'opérations sur ces données, sans toutefois exiger que les deux ne fassent partie d'un tout intégré. Les langages OO qui se respectent doivent formaliser le tout et regrouper les deux sous un même chapiteau.

Avant même de faire état de classes et d'instances, un langage OO doit permettre le regroupement de données et d'opérations mutuellement apparentées. *Les objets sont d'abord et avant tout des unités cohérentes d'organisation*.

La séparation de ces regroupements en **classes** et en **instances de ces classes**, pour distinguer modèle et actualisation du modèle, est une manière d'implémenter le regroupement de manière utile.

La plupart des langages OO aujourd'hui proposent aussi une saveur ou l'autre de l'idée de métaclasse (voir la section ***Les métaclasses***, plus bas) une abstraction dont les instances sont elles-mêmes des classes. Nous y reviendrons.

Cette catégorisation est semblable à celle qui distingue les types des variables dans les langages procéduraux.

On en vient presque systématiquement à conférer aux classes comme à leurs instances la dualité données/ opérations sur ces données. Ceci met de l'avant l'acception traditionnelle des objets, selon laquelle une classe est un objet *instanciable* alors qu'une instance est un objet *instancié*.

Encapsulation et regroupement

L'encapsulation présume le regroupement, au moins logique, mais le regroupement et l'encapsulation sont deux sujets distincts. Le terme anglais *Data Hiding* est une partie de l'encapsulation (l'idée que certains membres d'un objet aient des qualifications d'accès différentes de celles attribuées à d'autres membres du même objet) mais ne suffit pas à définir l'encapsulation prise dans son acception pleine et entière.

Encapsulation

À mon sens, il serait difficile de qualifier de véritablement OO un langage qui n'offrirait pas un support complet de **l'encapsulation**.

Par encapsulation, j'entends la capacité d'un objet de garantir son intégrité et, dans la mesure de ses possibilités, celle du système auquel il appartient, du début à la fin de son existence. Souvenons-nous que garantir son intégrité, pour un objet, signifie assurer le respect de ses invariants entre chaque invocation de ses méthodes.

J'ajoute ici *celle du système auquel il appartient* pour mentionner cette réalité qu'un objet convenablement encapsulé devrait non seulement garantir l'intégrité de ses propres données, mais devrait fermer les fichiers qu'il aura ouvert, terminer les *threads* qu'il aura lancés, fermer les liens de communication qu'il aura ouverts et, lorsque l'existence de l'objet aura atteint sa fin, *laisser le système dans lequel il évolue dans un état aussi proche que possible de celui dans lequel ce système aurait été à ce moment si l'objet en question n'avait jamais existé*. À mon sens, donc, un objet devrait être un bon citoyen des systèmes dans lesquels il est utilisé.

Exprimé autrement : à mon sens, la finalisation fait partie intégrante de l'encapsulation.

Par conséquent, pour implémenter correctement le concept d'encapsulation, un langage OO se doit d'offrir :

- un contrôle subjectif sur le début de l'existence des objets, donc supporter l'idée de constructeur ou permettre la systématisation de schémas de conception tels que la fabrique;
- un contrôle sur l'accès aux membres d'un objet. Pensez aux idées de privé et public, de même que (à un moindre niveau) celles de protégé et d'ami;
- un contrôle subjectif sur la fin de l'existence d'un objet, donc supporter l'idée de destructeur déterministe ou de finalisation déterministe, idéalement sous la responsabilité de l'objet lui-même ou, faite de mieux, sous la responsabilité explicite du code client; et
- la capacité pour un objet de décrire ses propres invariants d'une manière respectée et supportée par le langage. Ceci inclut la possibilité d'exposer des méthodes garantissant l'intégrité de l'objet pendant leur exécution. Ceci inclut les méthodes `const`, en jargon C++, mais aussi les contrats d'Eiffel et de C#.

Les langages à collecte automatique d'ordures simplifient la récupération de la mémoire mais compliquent la gestion des autres ressources, faute de finalisation déterministe. Ceci explique mes réserves face à la capacité (ou non) de ces objets d'offrir un niveau acceptable d'encapsulation : la dépendance envers le code client pour nettoyer derrière les objets, visiblement, montre que les objets ne peuvent pleinement gérer les ressources sous leur gouverne.

Polymorphisme

Le **polymorphisme** est une autre clé fondamentale du modèle OO; plus fondamentale même que l'héritage, sous ses diverses acceptions (héritage d'implémentation, héritage d'interfaces, héritage simple ou multiple, virtuel ou non, privé, protégé ou public). La capacité d'obtenir, à partir d'un lien indirect sur une abstraction, un comportement spécialisé sans avoir à inférer le type effectif de l'objet réellement référé est l'un des plus importants atouts opérationnels du modèle OO. Simuler cette fonctionnalité dans un langage procédural est possible mais très désagréable.

Scott Meyers, dans les diapositives d'une de ses formations en ligne, utilise cette définition du polymorphisme, définition à la fois large et puissante : *Polymorphism is the use of multiple implementations through a single interface*. La beauté de cette définition est qu'elle s'ouvre à des implémentations indépendantes même de l'héritage, et va à l'essence de l'idée.

Le polymorphisme est aussi l'une des clés les plus difficiles à bien assimiler du modèle. Par exemple, la capacité de différencier la signature d'une méthode par autre chose que par son nom est-elle conceptuellement équivalente au polymorphisme dynamique à travers une abstraction? La distinction entre les mots anglais *Overriding* et *Overloading* est importante[17]. Il est trop facile de décomposer le mot en ses racines pour *plusieurs* et pour *formes* et d'omettre la caractéristique dynamique à la racine historique de ce mécanisme.

Si vous tombez sur un volume qui parle de *Run-Time Binding* pour parler de polymorphisme (pas mal en soi) et de *Polymorphism* pour parler de la capacité pour plusieurs méthodes d'avoir le même nom (beaucoup moins correct), méfiez-vous. Le concept est beaucoup plus riche que ne le suggère ce maigre sucre syntaxique.

Avec la programmation générique, il est maintenant possible de réaliser du véritable *polymorphisme statique*, mais il s'agit là d'une technique avancée.

[17] … même si je n'utilise jamais ces termes dans mes cours parce que, n'étant pas anglophone, je n'arrête pas de les mêler! En général, *Overriding* est associé au polymorphisme et *Overloading* à la capacité de distinguer des opérations par la signature plutôt que par leur seul nom.

Héritage

La plupart des langages OO supporteront aussi **l'héritage**. Souvent, on parlera d'*héritage* pour parler d'héritage d'implémentation, et on écrira spécifiquement *héritage d'interfaces* lorsqu'on voudra faire la distinction entre les deux. Certains concepteurs de langages OO reconnus[18] prétendent même que l'héritage au sens d'héritage d'implémentation est une erreur qu'ils corrigeraient s'ils pouvaient reculer dans le temps.

À mon avis, c'est là aussi une réaction dogmatique et malheureuse. Le problème de fond, selon moi, est que les langages à héritage simple seulement (Java et les langages .NET en font partie) ramènent toutes les abstractions fondamentales à une seule et même racine (habituellement une classe `Object`, mère directe ou indirecte de toutes les autres – celles de ces deux plateformes se ressemblent, d'ailleurs, sans être identiques).

Ces racines uniques tendent à devenir un peu trop grasses pour leur propre santé, et utiliser une seule abstraction primitive entraîne un abus des conversions explicites de types, donc une perte d'abstraction causée par un des bris systématiques d'encapsulation.

Aujourd'hui, ce problème est (heureusement) compensé en partie par l'injection de généricité dans plusieurs de ces langages.

De même, l'héritage simple seul force des prises de position improductives, discutées dans le chapitre sur l'héritage multiple de [POOv01]. On peut compenser ces prises de position un peu arbitraires par l'introduction d'interfaces, mais en bout de ligne, tous ces langages compensent pour leur refus d'introduire l'héritage multiple.

Conceptuellement, l'héritage multiple permet des hiérarchies plus horizontales et un découpage plus propre des fonctionnalités. C'est un choix plus difficile à implémenter pour les concepteurs d'un langage, mais plus flexible sur le plan de la représentation pour les programmeurs. Offrir une hiérarchie d'objets sérieuse est chose plus élégante quand il est possible de construire sans faire une fourche parfois arbitraire dans une hiérarchie à héritage simple seulement.

Les idées de groupement, de spécialisation et de hiérarchisation, qui accompagnent le trio encapsulation, héritage polymorphisme, forment deux espèces de trinités de la perception traditionnelle de ce qu'est un langage OO.

Composition, agrégation et association

Je considère implicite la capacité de **procéder à la conception d'objets par composition ou par agrégation**. Si un langage offre l'héritage sans offrir la possibilité de décrire des composés ou des agrégats[19], je serai bien curieux de lire la philosophie conceptuelle qu'il véhicule.

La **relation d'association** est aussi une exigence de tout système OO pragmatique. Qu'un langage OO ne propose aucun mécanisme permettant aux objets d'interagir réduirait drastiquement, c'est le moins qu'on puisse dire, son potentiel commercial.

---

[18] Je pense en particulier à **James Gosling**, créateur de Java, entre autres dans une entrevue disponible sur http://www.artima.com/intv/gosling34.html, mais les concepteurs de modèles client/ serveur tels que COM tiennent fréquemment un discours semblable.

[19] Ce n'est pas impossible : il existe *beaucoup* de langages de programmation sur cette planète! Vous en trouverez quelques-uns dans [hdProgLang].

Abstraction

L'**abstraction**, prise au sens technique de capacité pour un langage d'exposer des interfaces pures ou des classes abstraites, est une marque de commerce des langages OO ayant eu du succès, sans pour autant être une nécessité du modèle. Sincèrement, je ne m'en passerais pas.

Personnellement, je trouve frustrant un langage se disant OO mais me refusant la capacité de définir des abstractions pures quand j'en sens le besoin intellectuel. Du point de vue de l'élégance expressive, à mon sens, l'abstraction est une conséquence du polymorphisme : le constat que, parfois, une classe (ou une interface) puisse parfois servir de stricte garantie polymorphique pour le code client.

Pour offrir un support convenable de l'abstraction, il n'est pas nécessaire d'offrir un mot clé pour les interfaces, les méthodes abstraites et les classes abstraites, mais il est nécessaire de permettre la mise en place d'un équivalent opérationnel direct de ces idées.

Équivalence opérationnelle des types

Si l'**équivalence opérationnelle des types** n'est pas une nécessité de l'approche OO, cela reste une marque par laquelle on peut reconnaître le caractère hybride ou non d'un langage. L'équivalence opérationnelle permet d'offrir, si cela s'avère pertinent, les mêmes services aux classes et aux types primitifs.

> Dans certains langages, comme Actor et Smalltalk, il n'y a pas de types primitifs du tout. Tous les types de données sont des objets, même le plus humble booléen. On obtient alors un système de types très homogène Malheureusement, ce choix a les défauts de ses qualités, et on paie souvent en performance le gain d'élégance ainsi réalisé.

La ***surcharge d'opérateurs*** [POOv00] s'inscrit dans cette veine, tout comme les traits (plus loin dans le présent volume); le premier permet de définir des classes qui s'utilisent comme des types primitifs (pensez à une classe `rationnel`, par exemple [hdRatio], dont chaque instance représente un nombre rationnel), alors que le second permet d'enrichir un programme par des algorithmes applicables autant à des objets munis de méthodes qu'à des entités primitives, et ce à partir d'une strate d'abstraction non intrusive.

L'équivalence opérationnelle est aussi un atout pour la véritable généricité, applicable à tous les types (pas seulement aux classes). Pouvoir initialiser ou copier de manière homogène une donnée de quelque type que ce soit est un atout inestimable dans la rédaction de code de qualité.

Au sens de cette rubrique, Java et les langages .NET sont plus hybrides que C++. Pris selon d'autres acceptions (voir la section ***Aucune opération sans objet***, plus bas), évidemment, l'interprétation serait différente.

Autre élément fondamental de l'équivalence opérationnelle des types : ***la capacité pour une instance de tout type, sans exception, d'être constante***. Devoir implémenter la constance par immuabilité, donc par type, est contraire à l'idée-même d'équivalence opérationnelle des types car cela crée une dichotomie entre ce qui est possible à l'aide de types primitifs et ce qui est possible pour les classes.

> Notez que les langages ne sont pas seuls responsables de ce manque. Sous .NET, par exemple, plusieurs langages interagissent sur le plan binaire à travers une même machine virtuelle, et implémenter la constance sur une base objet demande un soutien de la machine virtuelle elle-même (pas un petit changement, c'est le moins qu'on puisse dire).

Conséquence de la programmation générique, elle-même conséquence de l'équivalence opérationnelle des types : la capacité de faire de véritables conteneurs génériques applicables à tous les types sans exception, et sans avoir recours à de la magie de la part du compilateur (le *Boxing* de .NET et de Java, qui introduit implicitement au besoin une conversion de type primitif à équivalent objet et inversement).

Aucune opération sans objet

Plusieurs langages se disant plus OO que d'autres ont cette caractéristique de n'offrir **aucune fonction globale**, donc de n'offrir **que des méthodes**, et ce même s'ils font le choix pragmatique d'offrir à la fois des types primitifs et des objets.

Sous cette optique, Java est plus OO que C#, qui est à son tour plus OO que C++. Il n'est pas possible, en Java, d'écrire un sous-programme qui ne soit pas une méthode. C# impose la même contrainte, mais permet d'utiliser des délégués pour fins polymorphiques, ce qui introduit un niveau conceptuel intermédiaire dans le portrait. C++ considère les fonctions globales comme des ajouts externes acceptables à l'interface d'un objet et utilise ce mécanisme pour enrichir, par exemple, la gamme de types auxquels s'appliquent les opérateurs sur des flux.

Cela dit, si on oublie la fonction `main()` de C++, il est possible en C++ de faire du code identique à du code Java, sans la moindre fonction globale. L'option *que des méthodes* n'est pas la plus pragmatique[20], et certaines opérations usuelles (le opérations `min(a,b)` et `max(a,b)` sont des exemples classiques en ce sens) ne sont fondamentalement pas des opérations subjectives, ce qui mène à des implémentations OO où les objets n'ont vraiment comme seule qualité que celle de jouer un rôle de regroupement[21], comme le font les espaces nommés ou les modules traditionnels de langages comme Modula ou Pascal.

[20] La position philosophique de C++ est que l'approche OO n'est pas toujours la meilleure solution, alors en ce sens C++ est un hybride pragmatique (multi-paradigmatique) par choix.

[21] Par exemple à des trucs comme `Math.min(a,b)`, alors que le concept de `min()` rejoint celui de relation d'ordre génériques et transcende celui des mathématiques au sens arithmétique auquel s'attendent les gens qui explorent la classe `Math` de plus près.

Sérialisation et persistance

La capacité d'écrire un objet sur un flux ou d'extraire un objet d'un flux est étroitement associée à la capacité d'utiliser des objets avec des bases de données.

Une section entière sera consacrée à la question de la persistance des objets plus loin dans ce document.

En bref, la **sérialisation** est une problématique pour laquelle il existe une solution subjective (un objet est tout à fait en mesure de savoir comment se décrire sur un flux, et peut donc projeter sa représentation sur un média). Le problème de fond est que la ***désérialisation*** est une opération qui n'est fondamentalement pas subjective parce que, dans sa définition, elle opère sur un objet qui n'existe pas encore.

Un langage OO qui ne permettrait pas de projeter des objets sur un flux sous une forme reconnaissable (ou *reconstituable*), puis de récupérer ces objets ultérieurement empêcherait la rédaction de plusieurs programmes fondamentalement utiles; la sérialisation sur un flux est donc un problème pour lequel il existe des solutions viables dans la plupart des langages OO.

Les bases de données OO constituent un problème d'un ordre différent, de par les différences philosophiques à l'égard de ce qui est un objet : si un objet conçu à partir d'une philosophie est entreposé dans une telle base de données, quelle forme prendra-t-il s'il est extrait à l'aide d'un langage dans lequel la philosophie est différente?

Réflexivité

La sérialisation est parfois simplifiée dans un moteur qui offre un plein support à la réflexivité. Il en va de même pour la `POA`, sur laquelle nous reviendrons aussi plus loin dans ce document.

Une section entière sera consacrée à la question de la réflexivité plus loin dans ce document.

La **réflexivité**, en bref, est la capacité qu'ont les objets de se décrire eux-mêmes et d'être conceptualisés, structurellement, à partir du langage. Cette description se fait habituellement de manière dynamique, à l'aide d'objets disponibles dans les programmes : une classe représentant l'idée de classe, une classe représentant l'idée de méthode, des facilités pour instancier une classe par son nom, et ainsi de suite.

La réflexivité est souvent dynamique : au prix d'un ralentissement à l'exécution, un programme pourra décortiquer la structure d'une classe et l'instancier sans la connaître *a priori*. Avec C++[22], l'accent est mis sur le travail fait à la compilation pour éliminer tout impact qui ne soit pas explicitement souhaité sur le comportement du programme à l'exécution, mais une forme de réflexion statique est possible à l'aide de techniques de programmation générique [POOv02] ou de métaprogrammation.

Les langages offrant un support complet de la réflexivité dynamique permettent aux programmes de procéder à une forme d'introspection sur leur propre code et sur le langage en soi. Certains vont même jusqu'à permettre à un programme de modifier le sens de certains éléments du langage à l'aide duquel il est écrit.

[22] Il y a bien le *Run Time Type Information*, ou `RTTI`, incluant les opérateurs `dynamic_cast` et `typeid`, mais ces outils entraînent un coût à l'exécution. Certaines approches novatrices sont présentement explorées, en particulier [StrouSell] de ***Bjarne Stroustrup*** et ***Gabriel Dos Reis***, qui préconise la génération de deux sorties par compilation : l'une faite de code optimisé, et l'autre contenant des métadonnées d'accompagnement.

Héritage et design OO

Ce qui suit est fortement inspiré des textes de **Barbara Liskov** et de **Herb Sutter**.

L'héritage, surtout dans sa déclinaison publique, est une relation à très fort couplage. Après tout, l'enfant *est* (au moins) de la même nature que ses parents, structurellement parlant. On entend par couplage la difficulté de changer quelque chose sans affecter le code client.

Un principe de design parmi les plus importants est que **le couplage entre deux entités logicielles devrait être aussi fort que nécessaire, *pas plus***. La POO, bien appliquée, aide à appliquer ce principe. Si deux entités sont en relation l'une avec l'autre, un couplage plus faible facilite la modification de l'une entité sans affecter l'autre. Ceci allège l'entretien du code et réduit les coûts de développement.

Sachant cela, plusieurs considérations doivent être mises en valeur. Tout d'abord, examinons quelques types de couplage, du plus fort au plus faible.

Pour un objet, **l'exposition publique d'un membre** est le plus fort couplage envisageable, au sens où tout (tous les sous-programmes, tous les objets, vraiment tout) peut y accéder directement et de manière incontrôlée.

Qui dit accessible ne dit pas seulement modifiable mais bien utilisable. En ce sens, exposer même une constante de manière publique entraîne un cauchemar du point de vue de l'entretien (obligation de tenir à jour les noms et les types).

```
class X {
public:
   int f() const;
   int val; // Horreur!!!
};
```

Il faut donc presque toujours éviter cette clause pour les attributs d'une classe, et la restreindre à certaines méthodes. De même, on voudra restreindre la surface publique d'un objet au minimum pour réduire ce qui sera accessible au code client.

Le mot « presque » ici mérite une explication : puisque le recours à l'encapsulation tient d'abord et avant tout à assurer le respect des invariants du type encapsulé, un type pour lequel aucun invariant ne doit être maintenu peut exposer publiquement des attributs sans que cela n'entraine de préjudice. Notez par contre qu'il faut alors s'assurer que ce choix de n'avoir aucun invariant à garantir doit être une décision de design qui soit pleinement assumée : une fois un attribut exposé publiquement, nous ne pouvons plus revenir en arrière sans briser le code client. Voir [hdAttPub] pour plus de détails.

L'amitié est la plus forte catégorie de couplage autre que l'exposition publique d'un membre, du fait qu'elle donne aux amis, qu'il s'agisse de sous programmes ou de classes, un droit d'accès à tous les membres d'une classe, peu importe leur qualification de sécurité.

Qu'il s'agisse d'une fonction ou d'une classe, une amie contribue, de manière extrusive, à l'interface d'une classe.

Contrairement à certaines croyances dogmatiques, l'amitié peut améliorer la sécurité d'une classe si elle est appliquée avec sagesse (voir [POOv02] pour des détails).

```
class X {
   friend class Y;
   friend int f(X);
};
class Y { /* ... */ };
int f(X);
```

L'exposition protégée d'un membre est, contrairement à ce que plusieurs croient, à peine préférable à l'exposition publique des membres. Les accès incontrôlés de l'extérieur sont alors éliminés mais les enfants (en nombre arbitrairement grand) de la classe lui causent les mêmes problèmes de gestion que le feraient les classes qui ne lui sont pas apparentées.

```
class X {
protected:
   int f() const;
   int val; // Horreur!!!
};
```

En POO, un parent ne connaît pas plus ses enfants que ses clients. Exposer un attribut protégé est donc *pratiquement* toujours une très mauvaise idée. Il y a par contre plusieurs cas pour lesquels une méthode protégée est une stratégie raisonnable.

L'héritage public est une relation à très fort couplage, du fait que le code client peut tenir compte de la relation existant entre l'enfant et son parent… et, en pratique, le fera. En ce sens, l'enfant devient indissociable de son parent du fait que le monde entier peut tenir compte de leur relation.

```
class X : public Y {
   // ...
};
```

L'héritage protégé est une relation à moins fort couplage que l'héritage public mais qui entraîne son lot de problèmes dans l'entretien du code (comme toutes les relations protégées d'ailleurs), au sens où l'enfant devient foncièrement associé à son parent du fait que sa propre descendance est susceptible d'en tenir compte.

```
class X : protected Y {
   // ...
};
```

L'héritage privé est une relation à plus faible couplage que les autres héritages du fait que même la descendance de la classe enfant ne peut tenir compte de cette relation, qui devient un simple élément d'encapsulation. Puisque la relation parent/ enfant n'y est connue que de l'enfant, cette relation se rapproche d'une relation de composition (du point de vue du code client, les deux mènent au même résultat).

```
class X : Y { // privé
   // ...
};
```

Enfin, **les relations de composition, d'agrégation et d'association** sont à plus faible couplage encore que la relation d'héritage privé.

Cette affirmation ne s'avère bien entendu vraiment que dans la mesure où ces relations sont cachées dans les sections elles-mêmes marquées privées de l'objet.

```
class X {
   Y y;
   Z *z;
   // ...
};
```

Lors de la présentation de l'héritage [POOv01], l'accent a été mis sur une vision un peu simplifiée des relations entre objets :

- il fut mentionné que l'héritage devrait s'appliquer pour les relations au sens du verbe être (relations du type `Is-A`); et
- il fut mentionné que la composition, l'agrégation et l'association se prêtaient mieux à des relations sur la base du verbe avoir (relations du type `Has-A`).

Cette vision simplifiée (pour ne pas dire *simpliste*) des relations entre objets nous fut raisonnablement utile. Elle peut être reproduite dans des langages commerciaux et répandus mais dont le système de types est sémantiquement moins riche que celui de C++ et couvre les cas les plus importants. Dans la pratique, pour réaliser des designs plus complets et plus solides, il nous faut pousser la réflexion plus loin.

Notez que les méthodes polymorphiques, normalement, ne devraient pas être exposées publiquement. Il est préférable de qualifier une méthode polymorphique de protégée et d'en encadrer les invocations par une méthode publique non polymorphique, tel que proposé par l'idiome `NVI` [hdNVI].

Ne pas abuser de l'héritage public

Règle générale, qui dit ami, dérivé public, membre public ou membre protégé dit en fait *inclus dans la surface visible (dans l'interface) de l'objet*, donc connu et utilisable par d'autres. L'héritage public insère donc en quelque sorte une information supplémentaire et visible dans l'interface de l'enfant, même s'il ne s'agit que du simple fait que l'enfant est un cas particulier du parent, car de ce fait, l'enfant devient susceptible d'être exploité comme s'il était un cas particulier de son parent.

Ce constat est porteur de sens. Entre autres :

- l'héritage public a pour effet secondaire d'insérer une dépendance entre l'enfant et son parent, dépendance dont il faut prendre soin et dont on ne peut plus éviter de tenir compte. Toute modification à l'interface publique du parent entraîne une cascade de conséquences chez lui, mais aussi chez toute sa descendance publique (ou protégée); de même
- si le parent n'expose pas de méthode virtuelle, alors il est peu probable que l'héritage public ait été un bon choix. Notez que cela demeure vrai même si le parent du parent expose une méthode virtuelle, puisque dans ce cas la véritable relation d'héritage public est probablement celle entre le parent et le parent du parent.

Une bonne maxime pour ne faire le choix de l'héritage public que dans les cas où de choix est judicieux[23] est de penser l'héritage public un peu à l'envers des habitudes : `D` devrait dériver de `B` non pas parce que `D` veut utiliser sa partie `B` mais bien *parce que* `D` *veut être utilisé comme un cas particulier de* `B`.

L'héritage public, visiblement, se prête surtout aux spécialisations polymorphiques. Mêler héritage public et parent non polymorphique est une démarche suspecte.

Penser l'héritage public au sens d'une relation `Is-A` tel que proposé par le principe de substitution de **Barbara Liskov**[24] est une bonne pratique si :

- la relation décrite est vraiment une relation `Is-A`; ou encore
- il s'agit d'une relation intime au sens opératoire telle que `Works-Like-A` (*fonctionne comme un*, *s'utilise comme un*).

[23] J'ai lu cette sage recommandation à plusieurs endroits, en particulier dans [ExcCpp] par **Herb Sutter**, qui dit la tirer lui-même de **Marshall Cline** et **Greg Lomow**, *C++ FAQs*, Addison-Wesley, 1995.

[24] Voir http://www.objectmentor.com/resources/articles/lsp.pdf pour des détails.

Le problème de l'héritage public est qu'on le mêle souvent à des relations semblables mais pas tout à fait conformes (comme `Is-Almost-A`, ce qui pourrait se traduire par *ressemble à un* ou par *est presque un*). C'est là que la sauce se gâte. On peut dépister ces faux amis en se posant les deux questions suivantes :

- le parent et l'enfant ont-ils les mêmes exigences?
- le parent et l'enfant offrent-ils les mêmes promesses?
- les invariants de l'enfant sont-ils identiques à ceux du parent?

Le cas des classes `Rectangle` et `Carre`, que nous avons pourtant nous-mêmes utilisé dans notre discussion de l'héritage [POOv01] à cause de son caractère visuel et simplet, est un cas où recourir l'héritage public n'est *pas* une bonne idée.

En effet, tel que nos exercices l'ont démontré, les critères de validité d'un `Rectangle` et d'un `Carre` peuvent entrer en conflit les uns avec les autres. Modifier la largeur d'un `Carre` se voulant cohérent entraîne un effet de bord susceptible d'être inattendu pour qui pense avoir affaire à un `Rectangle`. En effet :

- si changer la hauteur d'une forme est une opération polymorphique, si un programme tient à jour une liste d'instances de `Rectangle` et double la largeur de chacune, il se trouve que la hauteur de celles qui sont aussi des `Carre` changera du même coup;
- si changer la hauteur d'une forme n'est pas une opération polymorphique, alors si un programme passe une référence sur un `Carre` à une fonction acceptant une référence sur un `Rectangle`, la fonction en question pourra traiter le `Carre` comme un `Rectangle`, et briser l'invariant selon lequel sa hauteur et sa largeur doivent concorder

L'héritage public, dans une situation où une dépendance entre deux classes existe mais ne se manifeste pas au sens opératoire, est une faute de design. Dans le cas du `Carre` et du `Rectangle`, il se peut qu'une relation d'héritage protégé ou privé existe, mais si du polymorphisme doit apparaître dans le portrait, alors il est préférable qu'il se situe à un autre niveau (p. ex. : un parent commun `Forme`, s'il y a une relation opératoire à ce niveau entre les classes `Carre` et `Rectangle`, ou encore un parent commun `Dessinable` si les deux doivent accepter de se dessiner par un appel polymorphique).

Relations plus intimes

Les relations structurelles sont mieux exprimées en termes d'héritage privé ou protégé, du fait que la dépendance entre enfant et parent est alors plus locale. Règle générale, une relation *est implémenté en fonction de* est mieux représentée par l'héritage privé ou protégé.

Les cas où l'héritage privé ou protégé sont envisagés méritent souvent qu'on se questionne à savoir si une relation à plus faible couplage encore (p. ex. : composition, agrégation) ne serait pas préférable. La réponse à cette question est assez simple : on conservera la relation d'héritage quand il est important que l'enfant puisse au moins se considérer lui-même comme un cas particulier de son parent, ou dans le cas où l'enfant a besoin, dans le cadre de ses fonctions, d'accéder aux membres protégés du parent.

Dans le doute, privilégier les relations à faible couplage (composition, agrégation, association) aux relations à plus fort couplage tend à donner de bons résultats.

Relations à très faible couplage

La programmation générique [POOv02] permet d'exprimer des relations à couplage très faible, soit des relations *Est implémentable en fonction de*. En effet, le code générique n'exploite que des opérations exposées par les entités sur lesquelles il opère, sans nécessairement être conscientisé quant à la nature propre de ces entités.

Lorsque les opérations requises par le contrat d'un type générique ou d'un algorithme générique se limitent à des opérations de construction, de copie, à des opérateurs et à l'application d'opérations sur le type en question, il devient possible de manipuler de manière indifférenciée types primitifs et objets en tant que tels.

Lorsque le contrat est plus riche, une option gagnante est de penser un tel algorithme en termes de **traits** (voir ***Comprendre les traits***, plus loin). Ceci le rendra applicable à l'ensemble des types pour lesquels il est raisonnable de le faire.

Les délégués [POOv02], typiques du monde `.NET`, offrent ce type de relation à faible couplage de par leur polymorphisme basé sur la signature, mais à un niveau d'abstraction beaucoup moins élevé du fait qu'ils ne sont pas applicables à tout type. On trouve des forces et des faiblesses analogues chez `std::function` de C++ et chez les pointeurs de fonctions de C.

En C++, à partir de C++ `17`, les concepts (voir ***Contraintes et concepts***) amèneront cette relation à très faible couplage – relation ne reposant que sur le respect des opérations contractuellement énoncées, de manière non-intrusive – à un tout autre niveau.

Choisir ses armes – les conteneurs standards

Nous avons abordé précédemment [POOv02] quelques conteneurs proposés par la bibliothèque standard de C++, en particulier `std::vector`.

Le choix de `std::vector`, qui se comporte comme un tableau dynamique, en tant que conteneur de base pour fins d'enseignement (et pour la majorité des tâches) n'est pas innocent. Pour citer **Bjarne Stroustrup** dans [CppPerf], p. 40 :

> « *Another example is that std::vector is a more compact data structure than std::list. A typical std::vector<int> implementation will use about three words plus one word per element, whereas a typical std::list<int> implementation will use about two words plus three words per element. That is, assuming sizeof(int)==4, a standard vector of 1,000 ints will occupy approximately 4,000 bytes, whereas a list of 1,000 ints will occupy approximately 12,000 bytes. Thanks to cache and pipeline effects, traversing such a vector will be much faster than traversing the equivalent list. Typically, the compactness of the vector will also assure that moderate amounts of insertion or erasure will be faster than for the equivalent list. There are good reasons for std::vector being recommended as the default standard library container*. »

De même, pour fins de démonstration dans un cours de systèmes en temps réel, j'ai fait une série de tests pour comparer les performances des vecteurs standards et des tableaux bruts. Les tests en question (et l'analyse qui les accompagne) sont disponibles dans Internet [hdVecTab], mais le constat le plus clair est que `std::vector` est un outil de *très* grande qualité, et qu'il est difficile d'atteindre les mêmes seuils de performance à l'aide de tableaux bruts.

Le vecteur standard est en général un bon choix de conteneur par défaut. Il tend à être plus compact que les autres, à être d'usage intuitif, et à offrir le niveau de performance le plus élevé pour les tâches les plus communes.

Cela dit, en programmation, il y a lieu de choisir ses armes avec soin. Chaque catégorie de conteneur a ses avantages et ses inconvénients, et il est souvent important de choisir le bon conteneur pour les bonnes opérations pour obtenir le programme le plus efficace possible.

Conteneur et modèle organisationnel

Si tous les conteneurs offrent un certain nombre de services communs (on pense entre autres à `begin()`, `end()`, `clear()`, `empty()`, `size()` et `push_back()`), certains offrent des services particuliers qui reflètent leur rôle organisationnel.

Par exemple, une pile (`std::stack`) offre un service d'insertion sur le dessus de la pile (`push()`), un service de suppression de l'élément du dessus de la pile (`pop()`) et un service de consultation du premier élément sur le dessus de la pile (`top()`).

Une file (`std::queue`) expose quant à elle un service d'insertion insérant à la fin de la file (`push()`) et un service de suppression au début de la file (`pop()`).

Sous `STL`, les conteneurs qui se distinguent par une stratégie d'insertion et d'extraction plutôt que par la complexité de stratégies standards sont en fait des ***adaptateurs*** pour des conteneurs plus que des conteneurs à part entière. Nous y reviendrons.

Le choix d'un conteneur sera principalement basé sur son comportement opératoire, qui dépendra en partie du type d'organisation qui y est modélisé.

Complexité algorithmique

Tout conteneur standard offre des garanties de complexité opérationnelle pour chaque opération qu'il implémente, ce que tout objet correctement documenté devrait d'ailleurs faire. Le site officiel de `STL` [Stl] contient d'ailleurs une page [StlCplx] expliquant cette démarche.

Ainsi, si les contraintes organisationnelles d'un conteneur importent moins que la manière par laquelle on y insérera des éléments, supprimera des éléments ou recherchera un élément, alors on prendra soin d'étudier l'utilisation faite du conteneur et d'estimer les opérations les plus susceptibles d'y être sollicitées (de même que la fréquence de sollicitation attendue pour chacune et, si possible, le contexte, critique ou non, dans lequel seront faites ces opérations[25]).

La bibliothèque `STL` est très bien documentée. En plus des invariants des classes, des préconditions et des postconditions des algorithmes et des méthodes, on y trouve la complexité algorithmique des divers algorithmes qui en font partie (voir par exemple la documentation de `std::sort()` [StlSort]). La complexité opérationnelle des opérations sur les itérateurs les plus simples est documentée clairement [StlFwIt], et il en va de même pour les détails des conteneurs [StlCtnr] ou des séquences [StlSqnc]. Ce sont, dans chaque cas, les seuils de complexité opérationnelle nécessaires; un conteneur spécifique pourra implémenter une version locale d'un algorithme donné qui offrira une performance nettement supérieure à ce seuil[26].

Les implémentations rigoureuses de bibliothèques standards, incluant la spécification du standard à partir desquelles sont implémentées les bibliothèques standard de C++, détaillent la complexité opérationnelle de chaque opération, permettant à un programmeur de faire une sélection éclairée.

---

[25] On comprendra que certains systèmes peuvent tolérer des délais quand les circonstances s'y prêtent mais peuvent aussi avoir une tolérance de beaucoup moindre à d'autres moments. Si certaines contraintes sont jugées critiques, alors celles-ci doivent être respectées rigoureusement, et ce même si la performance d'ensemble du système diminue par le fait même.

[26] Par exemple, plusieurs conteneurs offriront une méthode `swap()` interne qui sera beaucoup plus efficace sur eux que ne l'est l'algorithme standard `std::swap()`, du moins si on l'applique naïvement.

Passer d'un conteneur à l'autre

Utiliser un conteneur dans un objet est un choix porteur de conséquences. Le choix d'un conteneur influence le type d'opérations disponibles pour organiser l'information, mais aussi le coût, en temps ou en espace, de ces opérations, donc la qualité d'exécution des systèmes dans lesquels l'objet sera utilisé.

Typiquement, pour faire un choix entre les conteneurs généralistes, on dira que :

- si les insertions et les suppressions sont faites à la fin du conteneur et si la plupart des opérations faites sur ses éléments sont des parcours de séquence ou des accès directs à un élément donné, alors `std::vector` est à privilégier;
- si les insertions et les suppressions sont faites aux extrémités du conteneur, alors `std::deque` est à privilégier (ce qui explique que l'adaptateur `std::queue` repose par défaut sur un `std::deque`); et
- si les insertions et les suppressions sont faites à des endroits arbitraires dans le conteneur, s'il importe de pouvoir supprimer des éléments d'un conteneur sans invalider des itérateurs sur d'autres éléments du même conteneur, ou encore s'il importe que la mémoire utilisée diminue lorsque les éléments sont supprimés du conteneur[27], alors c'est sans doute `std::list` qui sera privilégié.

Les autres conteneurs sont plus spécialisés. On les choisira souvent à partir de critères plus spécifiques aux problèmes rencontrés. Un inventaire plus complet est offert dans la section ***Les conteneurs à notre disposition***, plus loin.

Il est possible qu'un objet puisse déterminer des étapes dans sa vie utile où certaines opérations seront plus présentes que d'autres. Par exemple, il se peut qu'une phase de l'utilisation d'un objet implique un nombre important d'insertions ou de suppressions dans l'un de ses conteneurs, puis qu'une autre phase implique un nombre important d'accès à des éléments de ce conteneur par leur position ou encore un nombre important de recherches d'éléments dans ce conteneur.

Sachant que certains conteneurs, par exemple les listes doublement chaînées (`std::list`), offrent des opérations d'insertion et de suppression d'éléments beaucoup plus rapides que ne le font les vecteurs, mais que l'accès direct au *i*ème élément d'un vecteur ou la recherche d'éléments dans un vecteur trié sont beaucoup plus rapides que leur contrepartie dans une liste, on est en droit de se demander quel type de conteneur choisir :

- doit-on mesurer la proportion de temps investie dans chacune des phases?
- sait-on quelles sont les attentes d'un usager dans un cas comme dans l'autre?
- estime-t-on raisonnable de payer pour un ralentissement dans un cas pour compenser pour l'accélération dans l'autre?

[27] Avec C++ 03, la mémoire consommée par un `std::vector` ne rapetisse jamais (il y a des trucs pour contourner cela, bien entendu). Sous C++ 11, `std::vector` offre une méthode `shrink_to_fit()` pour compacter un vecteur dont la capacité dépasserait le nombre d'éléments.

L'une des solutions est de passer d'un type de conteneur à l'autre au besoin. En effet, copier un conteneur dans un autre est presque nécessairement une opération de complexité linéaire ($O(n)$ pour un conteneur de n éléments). Le choix du moment pour passer d'un conteneur à l'autre peut dépendre d'une connaissance *a priori* du problème (un savoir *statique*, donc connu au moment de la compilation) ou de conditions mesurées alors que le programme s'exécute (plus complexe, et peut mener à des stratégies plus coûteuses qu'avantageuse si fait de manière imprudente, mais possible tout de même).

On ne peut habituellement utiliser l'opérateur d'affectation ou un constructeur par copie pour copier un conteneur quelconque dans un autre d'un type différent. Par contre, tous les conteneurs standards offrent un constructeur de séquence [POOv02], permettant de les construire à partir d'une séquence à demi ouverte.

À titre d'illustration, l'exemple ci-dessous lit un nombre arbitraire d'entiers dans une liste d'entiers (insertion peu coûteuse), puis construit un vecteur à partir de cette liste (remarquez la notation). Le vecteur est trié par un algorithme standard et ses éléments sont affichés à la console.

```
#include <iostream>
#include <list>
#include <vector>
#include <algorithm>
#include <iterator>
int main() {
   // ...using...
   list<int> lst;
   for (int val; cin >> val; )
      lst.push_back(val);
   vector<int> v(begin(lst), end(lst));
   sort(begin(v), end(v));
   copy(begin(v), end(v), ostream_iterator<int>{cout, " "});
}
```

Les conteneurs à notre disposition

Il existe plusieurs conteneurs standards à la disposition des développeurs, assujettis aux règles susmentionnées et respectant les règles définies sur les conteneurs standards de manière à permettre qu'on leur applique des algorithmes standard.

Nous avons déjà mentionné les classes `vector` et `basic_string` (surtout dans ses déclinaisons `wstring` et `string`) qui représentent des chaînes délimitées. Voici les autres conteneurs standards définis par la bibliothèque standard, et en grande partie par `STL`, de même que les catégories selon lesquelles ces conteneurs sont répartis. Il va de soi que tous ces conteneurs font partie de l'espace nommé `std`.

Dans les sections qui suivent, j'utiliserai ***sous-séquence*** pour parler d'une séquence dans un conteneur et j'utiliserai de manière libre les termes ***temps*** et ***complexité*** en les considérant comme interchangeables au sens de la complexité algorithmique. Notez qu'une complexité dite ***constante amortie*** est constante en général mais avec des pointes occasionnelles (lors de réallocations, principalement), pointes qui sont amorties sur l'ensemble.

Séquences

Une séquence définit un parcours des éléments d'un conteneur. On peut y insérer, en supprimer et y retrouver un élément. Notez toutefois que les algorithmes standards de `STL` ne peuvent pas manipuler la structure sous-jacente d'un conteneur (entre autres parce qu'ils sont applicables à des tableaux bruts, qui ne peuvent changer de dimension), ce qui limite le sens des mots « insérer » et « supprimer ».

Les conteneurs standards de C++ `03` sont :

- la classe **`vector`**, qui est l'implémentation à ultra haute performance du bien connu tableau dynamique. Les éléments de ce conteneur sont contigus en mémoire, ce qui est excellent pour la gestion des accès à l'antémémoire du processeur. L'accès au *`i`*ème élément d'un `vector` est fait en temps constant, alors que l'insertion et la suppression d'un élément à une position arbitraire s'y font en temps linéaire (à la fin du conteneur, le temps d'insertion et de suppression est constant amorti). L'ajout ou la suppression d'un élément à ce conteneur invalide tous les autres itérateurs qui s'y trouvent[28];
- la classe **`deque`**, qui représente une liste doublement chaînée de blocs d'éléments. Elle permet d'accéder au *`i`*ème élément selon une notation d'indice dans un tableau, en temps linéaire selon le nombre de blocs (ce qui se rapproche d'un temps constant sur le nombre d'éléments dans bien des cas). L'insertion et la suppression d'éléments s'y font en temps constant aux extrémités et en temps linéaire ailleurs. L'ajout ou la suppression d'un élément à ce conteneur invalide tous les autres itérateurs qui s'y trouvent;
- la classe **`list`** représente une liste doublement chaînée d'éléments. L'insertion et la suppression d'un élément ou d'une sous-séquence y sont rapides, mais la recherche du *`i`*ème élément y est de complexité linéaire. L'insertion et la suppression d'éléments dans une `list` n'invalide pas les itérateurs déjà détenus sur des éléments encore dans la liste.

[28] Supprimer un élément d'un vecteur décale tous les éléments qui le suivent d'un rang vers la gauche, et invalide automatiquement les itérateurs sur ces éléments. Ajouter à la fin risque d'entraîner une réallocation de la mémoire sous-jacente et un déplacement en mémoire de tous les éléments. Dans un cas comme dans l'autre, les itérateurs avant insertion ne peuvent plus être considérés valides.

Avec C++ 11, s'ajoutent à cette liste :

- la classe **`array`**, représentant un tableau de taille fixe et connue à la compilation. Ce type est générique sur la base du type `T` des éléments et du nombre `N` d'éléments (donc `array<int,3>` est un type représentant un tableau de trois `int`). Cette classe est très proche des tableaux bruts dont la taille est connue à la compilation, offrant précisément les mêmes performances mais avec des services supplémentaires (par exemple, la validation des bornes lors d'accès à un élément si la méthode `at()` est utilisée, ou encore la possibilité d'en consulter la taille); et
- la classe **`forward_list`**, représentant une liste simplement chaînée d'éléments d'un certain type. Cette classe est moins flexible que list mais consomme moins de mémoire pour chaque nœud, ce qui peut en faire un choix intéressant si les opérations qui y seront appliquées s'y prêtent.

Évidemment, rien ne vous empêche d'enrichir cette liste de séquences par des conteneurs de votre propre cru; c'est un exercice très instructif.

Conteneurs associatifs

Un conteneur associatif est un conteneur dynamique dont les éléments sont associés à des clés, donc qui est organisé à l'aide de paires {nom, valeur}. On peut y retrouver un élément à l'aide de sa clé, y insérer un élément, y supprimer un élément, mais on ne peut en général pas y insérer un élément à une position spécifique; chaque conteneur associatif a ses propres règles organisationnelles, qui sont pour lui des invariants.

Le type **`pair`**, représentant une paire nom/valeur (associant une valeur d'un type à une valeur d'un autre type) est souvent utilisé comme outil organisationnel sous-jacent. Une instance de `pair` expose deux attributs publics (eh oui!), `first` et `second`, correspondant aux deux éléments se trouvant associés dans la paire.

La classe **`set`** représente un ensemble dans lequel les éléments ne peuvent être dupliqués, alors que la classe **`multiset`** représente un ensemble dans lequel les éléments peuvent être dupliqués. Les deux sont définis dans `<set>`.

Certains algorithmes standards comme ceux conçus pour calculer l'union, l'intersection ou la différence entre deux ensembles, sont applicables à plusieurs conteneurs mais s'avèrent (sans surprise) particulièrement efficaces lorsque appliqués aux conteneurs `set` et `multiset`.

```
#include <string>
#include <set>
#include <iostream>
#include <iterator>
#include <algorithm>
// ...using...
template <class C>
   void aff(const string &intro, const C &c) {
      cout << intro;
      for(auto &val : c)
         cout << val << ' ';
      cout << endl;
   }
int main() {
   const string T0[]{ "coucou", "allo", "bingo" };
   const string T1[]{
      "coucou", "pigeon", "bingo", "canari"
   };
   set<string> A{begin(T0), end(T0)},
               B{begin(T1), end(T1)},
               C;
   aff("Ensemble A: ", A);
   aff("Ensemble B: ", B);
   cout << "Union: ";
   set_union(begin(A), end(A), begin(B), end(B),
             ostream_iterator<string>{cout, " "});
   cout << "\nIntersection: ";
   set_intersection(
      begin(A),end(A),begin(B),end(B),
      ostream_iterator<string>{cout, " "}
   );
   cout << endl;
   set_difference(begin(A), end(A), begin(B), end(B),
                  inserter(C, begin(C)));
   aff("Ensemble C (différence de A et B): ", C);
}
```

La classe **map**, qui tient à jour l'équivalent opérationnel d'un conteneur dont les éléments sont d'un type et les indices d'un autre type.

La notation indicée laisse croire que `map` cache un tableau, mais la plupart des implémentations reposent sur une structure arborescente pour accélérer les recherches.

Comme dans le cas d'un `set`, une `map` ne conserve qu'un seul élément associé à une clé donnée. La classe **multimap** représente une `map` pour laquelle un nombre arbitrairement grand d'éléments peuvent être associés à une même clé. Les deux sont définis dans `<map>`.

Notez que l'exemple proposé à droite utilisée une `map` dont les clés sont des chaînes de caractères étendus (des instances de `wstring`) plutôt que des chaînes conventionnelles à cause de l'accent dans `Août`. Cela explique que les littéraux soient précédés d'un `L`. Une `map` pourrait très bien avoir des clés de type `string`, `bool`, `int` ou autre.

```
#include <iostream>
#include <string>
#include <map>
// ...using...
int main() {
   map<wstring, int> mois;
   mois[L"Janvier"] = 31;
   mois[L"Février"] = 28; // souvent
   mois[L"Mars"] = 31;
   mois[L"Avril"] = 30;
   mois[L"Mai"] = 31;
   mois[L"Juin"] = 30;
   mois[L"Juillet"] = 31;
   mois[L"Août"] = 31;
   mois[L"Septembre"] = 30;
   mois[L"Octobre"] = 31;
   mois[L"Novembre"] = 30;
   mois[L"Décembre"] = 31;
   wcout << L"Juin -> " << mois[L"Juin"]
         << endl;
   auto cur = mois.find(L"Juin");
   auto pred = cur, succ = cur;
   ++succ;
   --pred;
   wcout << L"Précédent (ordre alphabétique): "
         << pred->first << endl;
   wcout << L"Suivant   (ordre alphabétique): "
        << succ->first << endl;
}
```

Conteneurs associatifs sans ordonnancement

Il existe aussi des conteneurs présents dans STL mais pas dans C++ 03. Un exemple important est ceux qui représentent des conteneurs reposant sur des techniques de « hachage »; ce sont des conteneurs associatifs mais pour lesquels il n'y a pas d'ordonnancement a priori pour les éléments. Ces conteneurs sont pleins de trous et consomment plus de mémoire que leurs cousins, mais sont optimisés pour l'insertion et pour la recherche.

Sous C++ 11, ces classes sont intégrées à la bibliothèque standard et sont affublées des noms **unordered_set**, **unordered_map**, **unordered_multiset** de même que **unordered_multimap**, respectivement. Ces conteneurs utilisent des clés qui sont *hashées* pour identifier rapidement l'indice des éléments à insérer ou à retrouver; la classe hash sert à cet effet.

Adaptateurs de conteneurs

Il existe aussi des classes qui ne sont pas tant des conteneurs que des adaptateurs de conteneurs. Ces classes enrobent un conteneur pour y implémenter des opérations particulières.

- la classe **stack** appliquée à un conteneur donné représente une pile[29] et expose toutes les méthodes standard du conteneur auquel elle s'adapte, en plus d'offrir les opérations spécialisées push(), pop() et top() qu'on s'attend à voir sur une pile;
- la classe **queue** appliquée à un conteneur donné représente une file[30] et expose toutes les méthodes standard du conteneur auquel elle s'adapte, en plus d'offrir les opérations spécialisées qu'on s'attend à voir sur une file; enfin
- la classe **priority_queue** appliquée à un conteneur donné représente une liste triée et expose toutes les méthodes standard du conteneur auquel elle s'adapte.

Par défaut, stack reposera sur deque (mais vector aurait été un bon choix); queue reposera aussi sur deque; alors que priority_queue reposera sur vector pour la vitesse avec laquelle il st possible de le trier.

La classe stack est définie dans <stack> alors que les classes priority_queue et queue sont définies dans <queue>.

[29] Stratégie *dernier arrivé, premier sorti*, ou *Last In, First Out* (LIFO).

[30] Stratégie *premier arrivé, premier sorti*, ou *First In, First Out* (FIFO).

Cordes

À titre d'exemple d'un conteneur `STL` qui ne fait pas partie du standard de C++ (mais peut tout de même être utilisé par des programmes C++ si ceux-ci ont accès aux fichiers appropriés), discutons brièvement de **cordes**.

Une corde, ou `rope`, représente un conteneur orienté vers les opérations applicables à de longues séquences de caractères. L'idée derrière ce type est d'offrir un mécanisme de support pour des éditeurs de texte, des courriels ou d'autres outils manipulant des chaînes de caractères arbitrairement longues.

La classe `rope` est encore en développement, ce qui explique que le fichier d'en-tête `<rope>` et la classe `rope`, de même que ses spécialisations `crope` (une corde de `char`) et `wrope` (une corde de `wchar_t`) font partie de `STL` mais ne font pas partie du sous-ensemble de `STL` inclus dans les bibliothèques standards `ISO` de C++.

Les chaînes standards, donc le type `basic_string` dans ses diverses déclinaisons, sont plus rapides que les cordes pour des manipulations applicables à un caractère à la fois. Les cordes sont quant à elles plus rapides pour manipuler de longues séquences de caractères (concaténer deux documents, par exemple, ou insérer un paragraphe quelque part dans un texte de `100 Mo`).

Groupes de bits

La classe **`bitset`** n'est pas un véritable conteneur mais représente une suite ordonnée de bits dont la taille est fixe. Elle permet de réaliser de manière efficace des opérations primitives sur des bits.

La beauté de `bitset` est que le substrat sous-jacent est géré avec élégance et efficacité, de manière à être compact et à faciliter les accès à un bit donné. Le prix à payer pour cette efficacité est de déterminer la taille du `bitset`, exprimée en nombre de bits, dès la compilation du programme.

```
#include <bitset>
#include <iostream>
// ...using...
int main() {
   bitset<16> demiMot(0x0303);
   demiMot |= 0x00c0;
   cout << demiMot << '\n';
   cout << demiMot.to_ulong() << endl;
}
```

Expressions constantes généralisées, ou `constexpr`

Avec C++ 11, une avancée technique importante est l'avènement des expressions constantes généralisées, identifiées par le mot clé **`constexpr`**.

La problématique

L'idée est la suivante : il arrive que des fonctions retournent des constantes connues à la compilation.

Par exemple, imaginez l'extrait de la classe `Note` proposé à droite. Cette classe expose (entre autres) les services `valeur_min()`, `valeur_max()` et `seuil_passage()` dont les valeurs reposent sur des constantes entières connues à la compilation.

Puisque `Note` est générique sur la base du type `T` utilisé pour la représentation de sa valeur, il n'est pas possible de garantir que `T` soit un entier, donc que des constantes de classe de type `T` puissent être initialisées à même la déclaration de la classe `Note`.

```
template <class T>
   class Note {
      enum {
          MIN = 0, MAX = 100, PASSAGE = 60
      };
      T val;
   public:
      T valeur() const {
         return val;
      }
      static T valeur_min() {
         return static_cast<T>(MIN);
      }
      static T valeur_max() {
         return static_cast<T>(MAX);
      }
      static T seuil_passage() {
         return static_cast<T>(PASSAGE);
      }
      // ...
   };
```

Les développeurs de la classe `Note` perdent alors une opportunité d'optimisation intéressante : en souhaitant gagner en généralité, ce qui est compréhensible (et souhaitable!), ils se trouvent à perdre l'accès aux seuils clés d'une `Note<T>` sur une base statique, ce qui les empêche par exemple d'écrire un programme comme celui-ci :

```
// ...inclusions et using...
int main() {
   // illégal... C'est triste!
   vector<Eleve> classement[Note<int>::valeur_max()- Note::valeur_min() + 1];
   // ...
}
```

Un tel programme a pourtant son intérêt : placer des instances d'`Eleve` dans des créneaux selon leur note obtenue est une chose raisonnable. Cependant, *il n'est pas possible de créer un tableau automatique à partir d'une taille qui ne soit pas une constante entière statique*.

Pour récapituler : ici, `Note<int>` aurait pu définir les seuils minimal, maximal et de passage comme étant des `static const T` puisque `T` est le type `int`. Dans le cas d'un réel, cela aurait été impossible. Le souci de généricité qui a mené à implémenter ces trois seuils sous forme de méthodes bloque, en retour, des avenues de programmation pertinentes.

Ce type de problème n'est pas si rare; on le rencontre même dans la bibliothèque standard elle-même[31]. Pour un langage comme C++, qui ne fait pratiquement pas de compromis sur la question de la performance, il s'agit d'un irritant très net.

[31] Le type générique `numeric_limits`, qui décrit des traits sur les types numériques, a la même limite (par exemple, `numeric_limits<T>::min()`, de la bibliothèque `<limits>`, retourne une constante connue à la compilation, mais demeure une méthode plutôt qu'une constante pour couvrir à la fois les entiers et les autres.

La solution passe par le langage

Il arrive que la solution à des problèmes de programmation passe par des bibliothèques ou par des techniques de programmation, mais dans ce cas-ci, la clé qui permet de résoudre le problème passe par un changement au langage et affecte le travail du compilateur.

Le changement en question est l'ajout d'un mot clé, `constexpr`, signifiant *la fonction suivante est en fait une expression constante connue à la compilation*.

Même un constructeur peut être qualifié de `constexpr` si ses paramètres sont en fait des constantes statiques. L'objet résultant est alors véritablement constant!

Par exemple, examinez le code ci-dessous :

```
class X {
   int val;
public:
   constexpr X(bool petit) : val{petit? 10 : 100} {
   }
   constexpr operator int() const {
      return val;
   }
};
constexpr float facteur() {
   return 0.5f;
}
int main() {
   double vals[X{false} * static_cast<int>(facteur())] { };
}
```

Ce code, qui serait illégal à bien des égards en C++ 03, s'avère légal en C++ 11. Puisque le littéral `false` est une constante statique, parce que `X{bool}` est un constructeur d'expression constante, parce que `facteur()` est une expression constante, parce que `static_cast<int>` sur l'expression retournée par `facteur()` est réalisé à la compilation, et parce qu'un `X` se convertit à la compilation en entier, le tableau `vals` dans `main()` sera de taille `50` et chaque case sera initialisée à `0.0`.

L'attribut d'instance `val` d'une instance de `X`, quant à elle, n'a pas à être `const` dans ce cas-ci. Le compilateur générera une erreur si l'application d'expressions constantes généralisées est incohérente dans le code client.

Sachant cela, revenons à la classe `Note`. Considérant ce que nous venons d'examiner un peu plus haut, il devient évident que dans ce cas-ci, le problème se règle aisément en qualifiant de `constexpr` les trois fonctions qui, au fond, ne font que réaliser des conversions statiques de constantes statiques.

Ainsi, le code client profitera des services de `Note` si cela s'avère possible, et la généricité de la classe `Note` sera préservée dans tous les cas. Nous obtiendrons à la fois la vitesse, l'élégance et la flexibilité.

```
template <class T>
   class Note {
      T val = valeur_min();
   public:
      Note() = default;
      constexpr T valeur() const {
         return val;
      }
      static constexpr T valeur_min() {
         return 0;
      }
      static T constexpr valeur_max() {
         return 100;
      }
      static T constexpr seuil_passage() {
         return 60;
      }
      // ...
   };
```

Des tests réalisés à la compilation – `if constexpr`

Depuis C++ 17, C++ supporte le concept de tests réalisés à la compilation à même une structure de contrôle du langage. Ces tests ont la particularité suivante :

- la condition doit pouvoir être évaluée à la compilation, et exclut donc tout élément qui serait évalué à l'exécution d'un programme;
- la branche « `if` » existera seulement si la condition est vraie;
- la branche « `else` » n'existera quant à elle que si la condition est fausse;
- les deux branches doivent être bien formées au sens du langage – ce mécanisme diffère en ce sens des `#ifdef … #else … #endif` du préprocesseur.

Ce mécanisme, inspiré en partie du `static_if` du langage D mais distinct de ce dernier (qui s'harmonisait plus ou moins bien avec certaines particularités de C++), porte en C++ le nom `if constexpr`.

Il s'agit d'un mécanisme à la fois simple et extrêmement puissant.

> Certains disent aussi `constexpr if` car ce nom a flotté au comité de standardisation en début de parcours, mais `if constexpr` a été privilégié car cette écriture ouvre la porte à d'autres structures statiques (p. ex. : un éventuel `for constexpr`)

Exemple – optimisations mineures

Un cas d'utilisation – et d'optimisation – simple pour ce mécanisme est la résolution à la compilation de conditions qui auraient pu être évaluées (inutilement) à l'exécution. Pour illustrer ceci, prenons un exemple naïf :

```
template <class T>
   void decrire(T obj) {
      enum { SEUIL = 64 };
      if (sizeof(T) < SEUIL)
         cout << "C'est un petit objet" << endl;
      else
         cout << "C'est un gros objet" << endl;
   }
```

Ici, la condition évaluée dépend de deux constantes connues à la compilation soit la valeur de `SEUIL` et la taille en *bytes* du type `T`. Le code sera plus compact (supprimant les branches inappropriées) et plus rapide (supprimant le test) si nous l'exprimons plutôt comme suit :

```
template <class T>
   void decrire(T obj) {
      enum { SEUIL = 64 };
      if constexpr (sizeof(T) < SEUIL)
         cout << "C'est un petit objet" << endl;
      else
         cout << "C'est un gros objet" << endl;
   }
```

Exemple – simplification de sélections statiques

Soit la manœuvre de programmation suivante, qui décrit un calcul de proximité choisi sur la base des types (exact, ou encore nombre à virgule flottante) :

```
#include <type_traits>
// ...
class exact{};
class flottant{};
template <class T>
   constexpr T absolue(T val) {
      return val < 0? -val : val;
   }
template <class T>
   constexpr bool assez_proches(T a, T b, exact) {
      return a == b;
   }
template <class T>
   constexpr bool assez_proches(T a, T b, flottant) {
      return absolue(a - b) <= static_cast<T>(0.000001);
   }
template <class T>
   constexpr bool assez_proches(T a, T b) { // aiguillage
      return assez_proches(a, b, conditional_t<is_floating_point_v<T>, flottant, exact>{});
   }
```

La manœuvre peut sembler étrange, mais elle est plutôt chouette : la fonction d'aiguillage (celle à deux paramètres) examine par un trait standard la nature de `T`, à savoir s'il s'agit d'un nombre à virgule flottante ou pas, puis délègue à la bonne version de la fonction prenant trois paramètres pour appliquer un algorithme choisi à la compilation. Le résultat peut entièrement être évalué à la compilation.

Avec `if constexpr`, la même manœuvre s'exprime plus simplement :

```
#include <type_traits>
// ...
template <class T>
   constexpr T absolue(T val) {
      return val < 0? -val : val;
   }
template <class T>
   constexpr bool assez_proches(T a, T b) { // aiguillage
      if constexpr (<is_floating_point_v<T>)
         return absolue(a - b) <= static_cast<T>(0.000001);
      else
         return a == b;
   }
```

Exemple – fonctions aux multiples types de retour

Supposons une fonction devant retourner un tampon de `T`, et privilégiant un `array<T,N>` si `N*sizeof(T)` est suffisamment petit, mais préférant `vector<T>` dans le cas contraire.

Ce genre de manœuvre est plutôt complexe avec C++, du moins jusqu'à C++ `17`. Cependant, avec `if constexpr` et `auto`, c'est assez simple :

```
template <class T, int N>
   auto creer_tampon() {
      if constexpr(sizeof(T) * N <= SEUIL)
         return array<T,N>{};
      else
         return vector<T>(N);
   }
```

Comme vous pouvez le constater, on parle d'un mécanisme étonamment polyvalent, et fort agréable à avoir sous la main.

Manières d'appliquer l'approche OO

Nous allons, dans cette section, illustrer une application plus complexe et plus complète de la POO que celles ayant des visées académiques qui sont parsemées ici et là dans ce document et dans les documents précédents.

Notre objectif sera le suivant :

- nous considérerons un écran console comme étant une arène dans laquelle évolue un personnage. Si le personnage devait quitter l'arène, il lui arriverait des choses innommables;
- un usager sera appelé à entrer des commandes pour déplacer ce personnage. Il pourra le faire avancer, le faire tourner vers la gauche (en sens antihoraire) ou le faire tourner vers la droite (en sens horaire);
- dès que le personnage mettra un pied hors de l'arène, une exception sera levée.

Il existe plusieurs solutions à un tel problème. Certaines sont très simples mais peu réutilisables. De manière générale, accroître la capacité qu'a un objet d'être réutilisé tend à accroître le niveau d'abstraction requis pour le représenter.

Nous irons délibérément vers l'autre extrême, et tenterons d'obtenir une solution qui sera très abstraite, très réutilisable et, ce qui souvent est une conséquence directe de code conçu de manière générique[32], très rapide à l'exécution.

Chaque fois que ce sera possible, nous essaierons de produire des types valeurs et de nous fier sur la Sainte-Trinité.

Le programme principal visé

Nous travaillerons en vue d'obtenir le programme principal proposé à droite; on y créera un Personnage (Bill), un Commentateur et un LecteurCommandes (brad), puis on y décortiquera des commandes entrées par l'usager jusqu'à ce que bill quitte le terrain.

Visiblement, bill sera considéré comme ayant quitté le terrain lorsque brad signalera, à l'aide d'une exception, l'atteinte par bill d'un point illégal – la présomption ici est que ce sera quelque chose d'atypique… d'exceptionnel. Ce n'est évidemment pas la seule approche possible (qu'en pensez-vous?), mais c'est ce que nous chercherons à faire.

```
#include "Personnage.h"
#include "Commentateur.h"
#include "LecteurCommandes.h"
#include <iostream>
// ...using...
int main() {
   Personnage bill{"Bill"};
   Commentateur fred{cout};
   LecteurCommandes
      brad{bill, cin, cout};
   try {
      while (brad.analyser())
         ;
   } catch (Point::Illegal&) {
      cerr << bill.nom()
           << " quitte l'arène\n";
   }
}
```

[32] Contrairement à certaines croyances populaires, les langages OO contemporains (en particulier C++) offrent des facilités permettant d'atteindre un niveau d'abstraction extrêmement élevé tout en générant du code d'une efficacité considérable. L'époque où le mot « général » signifiait « lent » est une époque révolue, du moins pour qui sait penser à la fois de manière générique et pratique. Cette section se veut exemplaire en ce sens.

Représenter la direction

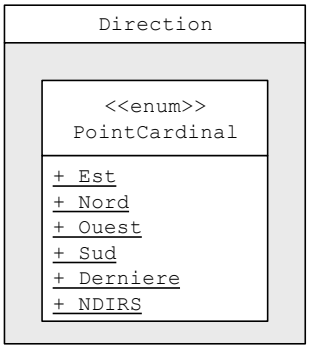
Le personnage pourra se déplacer dans l'arène. Il nous faudra donc représenter le concept de lieu et le concept de direction. Nous commencerons par l'idée de direction (ou d'orientation, si vous préférez).

> ***Note sur les diagrammes UML***
>
> Dans la plupart des diagrammes qui suivent, nous omettrons les constructeurs et les destructeurs par souci d'économie.

Dans un souci de simplicité, nous dirons qu'un personnage avancera dans la direction vers laquelle il regarde et qu'il pourra se déplacer dans cette direction seulement. Nous accepterons les demandes pour faire pivoter le personnage vers la gauche ou vers la droite de `90°`. Sachant cela, il semble raisonnable de considérer suffisants les quatre points cardinaux pour représenter les directions possibles.

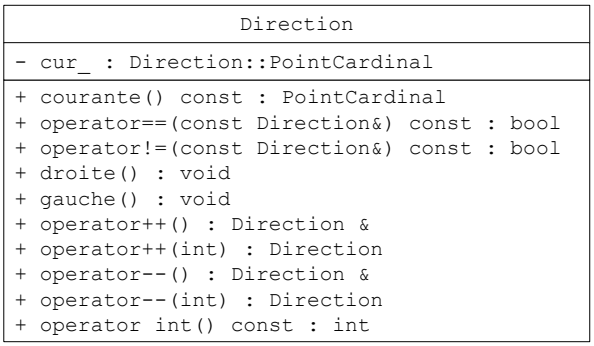
Imaginons donc la classe `Direction` dans laquelle se trouvera un type énuméré `PointCardinal`. Ce type interne servira à distinguer de manière efficace les quatre points principaux apparaissant sur la rose des vents.

La notation `UML` pour une énumération est d'indiquer la mention `<<enum>>` dans la section du haut de la case représentant le type énuméré. Nous utiliserons ici des énumérations fortes.



La classe `Direction` en tant que telle contiendra un attribut indiquant le point cardinal correspondant.

Les opérateurs `++` et `--` (à prendre au sens de *suivante* et *précédente*, un peu comme avec les itérateurs bidirectionnels) permettront respectivement un virage à droite et à gauche. Il en ira de même pour les méthodes `droite()` et `gauche()`.



Le code suggéré suit (fichier d'en-tête seulement; un fichier source serait superflu) :

```
#ifndef DIRECTION_H
#define DIRECTION_H
class Direction {
public:
   enum PointCardinal : short {
      Est, Nord, Ouest, Sud, Derniere = Sud, NDIRS
   };
private:
   PointCardinal cur = Nord;
public:
   Direction() = default;
   constexpr Direction(PointCardinal p) noexcept : cur{ p } {
   }
   constexpr PointCardinal courante() const noexcept {
      return cur;
   }
   constexpr bool operator==(const Direction &p) const noexcept {
      return courante() == p.courante();
   }
   constexpr bool operator!=(const Direction &p) const noexcept {
      return !(*this == p);
   }
   void droite() noexcept { // horaire
      cur = PointCardinal( (static_cast<int>(*this) + 1) % NDIRS);
   }
   void gauche() noexcept { // antihoraire
      cur = PointCardinal((static_cast<int>(*this) + NDIRS - 1) % NDIRS);
   }
   Direction& operator++() noexcept {
      droite();
      return *this;
   }
   Direction operator++(int) noexcept {
      auto temp{ *this };
      operator++();
      return temp;
   }
   Direction& operator--() noexcept {
      gauche();
      return *this;
   }
   Direction operator--(int) noexcept {
      auto temp{ *this };
      operator--();
      return temp;
   }
```

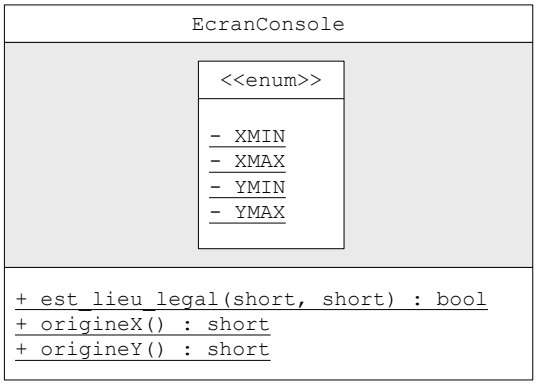
```
   constexpr operator int() const noexcept {
      return static_cast<int>(courante());
   }
};
#endif
```

Représenter un espace

Représenter un espace efficacement et élégamment est une chose plus complexe qu'il n'y paraît. Évidemment, plusieurs solutions sont possibles, et la suivante est particulièrement abstraite, mais aussi particulièrement efficace à plusieurs points de vue. Notez que nous définirons un espace `2D` simple, mais que la stratégie employée ici pourrait s'étendre à des dimensions plus élevées.

Tout d'abord, présumons que tout lieu doive exprimer ce que signifie être à l'origine et que tout lieu doive signifier à l'aide de quel type s'expriment les distances pour lui.

Un `EcranConsole` sera un lieu sur lequel la distance sera représentée sur un `short`, et qui ajoutera le concept de ce que constitue une paire de coordonnées légales (sera considérée légale une paire de coordonnées située dans l'arène).



Nous verrons mieux avec la classe `Point`, représentant un lieu dans un espace et dont nous discuterons plus loin, à quel point la stratégie utilisant des traits est intéressante.

Nous aurons recours à un petit fichier `OutilsGeneriques.h` qui contiendra :

```
#ifndef OUTILS_GENERIQUES_H
#define OUTILS_GENERIQUES_H
template <class T>
   bool est_entre_inclusif(const T &val, const T &borneInf, const T &borneSup) {
      return borneInf <= val && val <= borneSup;
   }
template <class T>
   bool est_entre_exclusif(const T &val, const T &borneInf, const T &borneSup) {
      return borneInf < val && val < borneSup;
   }
#endif
```

Le code d'`EcranConsole` suit :

```
#ifndef ECRAN_CONSOLE_H
#define ECRAN_CONSOLE_H
#include "OutilsGeneriques.h"
class EcranConsole {
public:
   using position_type = short;
   using distance_type = short;
private:
   static constexpr const position_type
      XMIN = 0, XMAX = 79, YMIN = 0, YMAX = 24;
public:
   static bool est_lieu_legal(position_type X, position_type Y) noexcept {
      return est_entre_inclusif(X, XMIN, XMAX) && est_entre_inclusif(Y, YMIN, YMAX);
   }
   static constexpr position_type origineX() noexcept {
      return XMIN;
   }
   static constexpr position_type origineY() noexcept {
      return YMIN;
   }
};
#endif
```

Vous remarquerez des opportunités d'optimisation importantes si le compilateur utilisé supporte C++ 11, en particulier dans l'application possible d'expressions constantes généralisées (voir ***Expressions constantes généralisées, ou `constexpr`***) dans plusieurs cas.

Représenter un lieu dans un espace

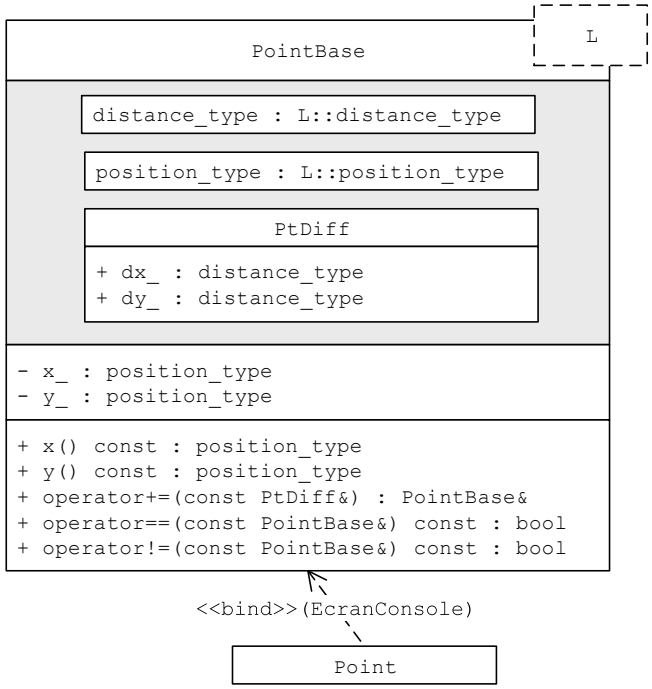
Pour représenter un lieu (un *point*) dans un espace, j'ai choisi de concevoir une classe `PointBase`. Un `PointBase` sera une classe générique dont le rôle est de représenter ce que signifie être quelque part dans un espace `2D`.

La beauté de la généricité ici est que définir `PointBase` comme une classe générique (reposant sur le choix du type d'espace) permet de lui appliquer les caractéristiques d'un lieu quelconque. *Le concept de lieu devient cohérent parce que l'espace dans lequel il réside l'est*.

Les idées de position et de distance entre deux points sont définies à partir de ces caractéristiques. Il est important de distinguer *différence entre points* et *point* : un point légal doit se situer dans l'arène, mais une différence entre points peut avoir des composants `dx` et `dy` négatifs pour représenter un déplacement particulier.

Un `PointBase` connaîtra sa position. Certaines méthodes et certains opérateurs relativement évidents y sont définis.

Le type `Point`, utilisé dans notre programme pour représenter un lieu, est une instanciation de `PointBase` à partir des caractéristiques de la classe `EcranConsole`.



Joindre la généricité à l'approche `OO` dans la classe `Point` permet, sans jamais indiquer le type des coordonnées d'un point, de déterminer *dans l'abstrait* les idées de position et de distance d'un lieu à partir de paramètres de l'espace auquel ce lieu est appliqué. Le tout se fait en pleine et entière généralité, à la compilation, sans la moindre perte de vitesse à l'exécution et sans réel coût supplémentaire en espace.

Le code de `Point.h` suit :

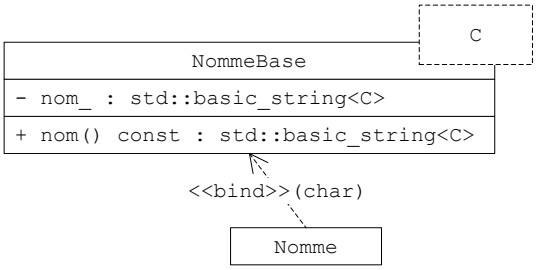
```
#ifndef POINT_H
#define POINT_H
template <class L>
   class PointBase {
   public:
      using Lieu = L;
      using distance_type = typename Lieu::distance_type;
      using position_type = typename Lieu::position_type;
      struct PtDiff {
         distance_type dx, dy;
         constepxr PtDiff(const distance_type &dx, const distance_type &dy): dx{dx}, dy{dy}{
         }
      };
   private:
      position_type x_ = Lieu::origineX(),
                    y_ = Lieu::origineY();
      bool est_legal() const {
         return Lieu::est_lieu_legal(x(), y());
      }
   public:
      class Illegal {};
      constexpr PointBase() = default;
      constexpr PointBase(const position_type &x, const position_type &y) : x_{x}, y_{y} {
         if (!est_legal()) throw Illegal{};
      }
      constexpr position_type x() const {
         return x_;
      }
      constexpr position_type y() const {
         return y_;
      }
      PointBase& operator+=(const PtDiff &p) {
         x_ += p.dx_;
         y_ += p.dy_;
         if (!est_legal()) throw Illegal{};
         return *this;
      }
      constexpr bool operator==(const PointBase &p) const {
         return x() == p.x() && y() == p.y();
      }
      constexpr bool operator!=(const PointBase &p) const {
         return !(*this == p);
      }
   };
#include "EcranConsole.h"
using Point = PointBase<EcranConsole>;
```

```
#include <iosfwd>
template <class L>
   std::ostream& operator<<(std::ostream &os, const PointBase<L> &p) {
      return os << "(" << p.x() << "," << p.y() << ")";
   }
#endif
```

Une abstraction pour tout ce qui porte un nom

Notre petit système permettra de déplacer un personnage dans une arène, mais rien ne dit que nous ne voudrons pas un jour y déplacer un char d'assaut ou un cloporte.

Si nous voulons être capables de commenter les déplacements de quelque chose, il peut être pratique de découpler les concepts de *déplaçable* et de *nommé*.



Nous dirons d'un objet nommé qu'il porte un nom, fixé dès le moment de sa construction[33], qu'il sera en mesure, sur demande, de révéler. Tout objet nommé (toute instance d'une classe dérivée de `Nomme`) portera nécessairement un nom, ce qui facilitera la rédaction d'objets commentateurs relativement génériques.

Le code du fichier `NommeFwd.h` sera tel que proposé à droite. Il définira entre autres la structure générale de la classe `Nomme` comme étant une spécialisation d'un *template* général à partir d'un `char`, pour éviter à qui le souhaite d'inclure la déclaration complète de la classe.

```
#ifndef NOMME_FWD_H
#define NOMME_FWD_H
template <class> class NommeBase;
using Nomme = NommeBase<char>;
using wNomme = NommeBase<wchar_t>;
#endif
```

Il s'agit de la même technique que celle utilisée dans le cas de la bibliothèque standard pour `<iosfwd>`. C'est d'ailleurs la raison pour laquelle le code est proposé ici sous cette forme même pour une implémentation aussi simple.

La classe `Nomme` sera une spécialisation du cas générique `NommeBase`. Le fichier `Nomme.h` inclura `NommeFwd.h` pour assurer la cohérence des deux déclarations.

Dans un cas comme dans l'autre, tous les algorithmes pensés pour agir sur des `std::basic_string` fonctionneront de manière homogène sur le nom de l'objet.

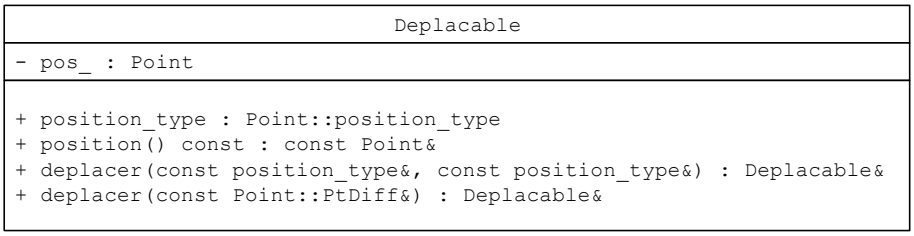
```
#ifndef NOMME_H
#define NOMME_H
#include "NommeFwd.h"
#include <string>
template <class C>
   class NommeBase {
   public:
      using str_type std::basic_string<C>;
   private:
      str_type nom_;
   public:
      NommeBase(const str_type &nom) : nom_{nom} {
      }
      str_type nom() const {
         return nom_;
      }
};
#endif
```

[33] Dans ce cas, l'attribut `nom_` aurait pu être une constante d'instance, mais cela aurait bloqué la capacité des dérivés d'implémenter la copie par affectation.

Le concept de déplaçable

Plusieurs stratégies sont envisageables pour représenter la capacité de se déplacer. L'une d'entre elles est de définir ce que signifie être déplaçable à l'aide d'une classe puis d'en dériver toutes les classes qui sont déplaçables.

Notre classe `Deplacable` aura comme attribut une position courante (un `Point`), des mécanismes de première ligne pour y accéder, et des méthodes de déplacement reposant sur une différence de points.



L'idée ici est de faire reposer les mécanismes de la classe `Deplacable` sur les abstractions de position et de différence entre points de la classe `Point`, elle-même conçue à travers les traits du lieu pour lequel elle a été conçue. Ceci permettra de maximiser l'abstraction (donc l'applicabilité) de l'idée de déplaçable sans entraîner de perte de performance à l'exécution.

Le code suit.

```
#ifndef DEPLACABLE_H
#define DEPLACABLE_H
#include "Point.h"
class Deplacable {
   Point pos_ {};
public:
   using ptdiff_t = Point::PtDiff;
   using position_type = Point::position_type;
   Deplacable() = default;
   Deplacable(const Point &pos) : pos_{pos} {
   }
   Point position() const {
      return pos_;
   }
   Deplacable& deplacer(const position_type &dx, const position_type &dy) {
     return deplacer(ptdiff_t{dx, dy});
   }
   Deplacable& deplacer(const ptdiff_t &dPos) {
      pos_ += dPos;
      return *this;
   }
};
#endif
```

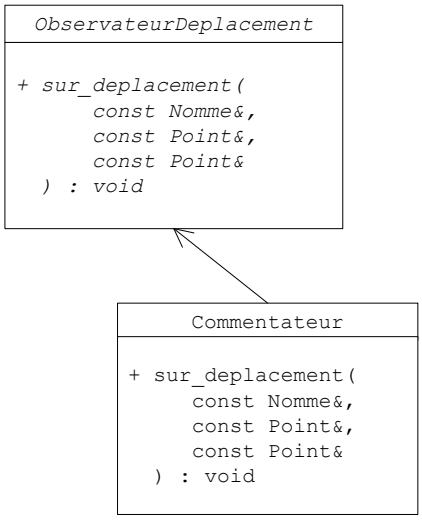
Remarquez au passage qu'un `Deplacable` par défaut sera situé à la construction à une position correspondant au `Point` qu'est l'origine de l'espace dans lequel il existe. `Deplacable` peut être écrit en totalité dans l'abstrait tout en restant aussi rapide que possible.

Il y a un contrat sémantique à l'idée qu'un `Point` par défaut doive se situer à l'origine de son lieu, et ce contrat doit être explicité, au moins dans la documentation de la classe, ou mieux encore : à même le code.

Appliquer le schéma de conception observateur aux déplacements

Pour être en mesure de réagir aux déplacements d'un déplaçable, il peut être utile d'avoir recours à un ***observateur***.

⇒ Un **observateur** est un schéma de conception par lequel un objet s'abonne à un fournisseur pour être tenu au courant de l'occurrence de certains événements.



Ce schéma de conception se présente habituellement comme suit :

- chacun des observateurs implémente une même interface;
- cette interface expose une ou plusieurs méthodes;
- chacune des méthodes de l'interface représente un événement;

- l'observateur s'abonne chez un fournisseur. Le fournisseur tient à jour une liste de ses abonnés, habituellement avec des conteneurs dont la taille est dynamique comme une liste ou un vecteur;
- ce fournisseur peut être tout objet susceptible de rencontrer des événements: un bouton qu'on pourrait presser, un port de communication sur lequel pourraient arriver des données, un système de sécurité détectant des tentatives d'intrusion sur un lieu, un personnage se déplaçant, *etc*.;

Les observateurs constituent un schéma de conception fort répandu dans les systèmes événementiels. Les interfaces personne/ machine sont d'ailleurs un exemple probant d'utilisation de cette stratégie.

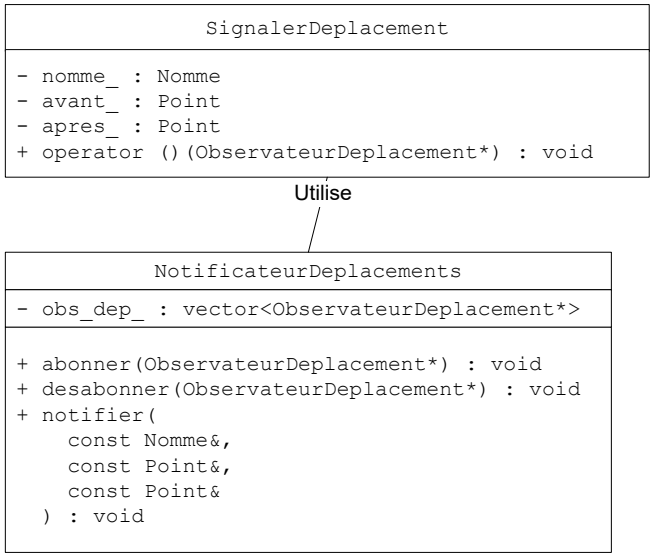
Il faut par contre éviter que les abonnés ne provoquent, de par leur réaction à un événement, un autre événement du même acabit, ce qui entraînerait une récurrence potentiellement infinie et entraînerait une dégradation ou un plantage du système.

Habituellement, le code d'un observateur sera aussi linéaire que possible. En effet, lors d'un événement, le fournisseur ne pourra informer un observateur que si l'observateur précédent a terminé de réagir. Tout observateur réagissant lentement empêche les autres de bien faire leur travail.

- le fournisseur, lorsque l'un des événements sous sa gouverne se produit, itère à travers la liste de ses abonnés, qui sont les observateurs de cet événement, et appelle la méthode appropriée pour chacun d'entre eux;

- l'abonné, quand le fournisseur le rappelle, réagit à l'événement à l'aide du code de la méthode en question.

Dans notre petit système, l'interface de rappel des observateurs, nommée `ObservateurDeplacements`, exposera une méthode nommée `sur_deplacement()`, qui prendra en paramètre un nommé et deux points (avant et après).



La classe `NotificateurDeplacements` servira de de base aux fournisseurs et utilisera (en arrière-plan) un foncteur nommé `SignalerDeplacement` pour avertir les observateurs lorsqu'un déplacement se sera produit chez le fournisseur.

On s'en doutera, les personnages de notre système dériveront de `NotificateurDeplacements`.

Le code suit. Pour le fichier `SuiviDeplacements.h`, on aura :

```
#ifndef SUIVI_DEPLACEMENTS_H
#define SUIVI_DEPLACEMENTS_H
#include "Point.h"
#include "NommeFwd.h"
struct ObservateurDeplacement {
   virtual void sur_deplacement
      (const Nomme&, const Point &avant, const Point &apres) = 0;
   virtual ~ObservateurDeplacement() = default;
};
#include <vector>
class NotificateurDeplacements {
   std::vector<ObservateurDeplacement *> obs_dep;
public:
   class PasAbonne {};
   void abonner(ObservateurDeplacement *p) {
      obs_dep.push_back(p);
   }
   void desabonner(ObservateurDeplacement *);
   void notifier(const Nomme&, const Point &avant, const Point &apres);
};
#endif
```

Pour le fichier `SuiviDeplacements.cpp`, on aura :

```
#include "SuiviDeplacements.h"
#include "Nomme.h"
#include <algorithm>
#include <vector>
// ...using...
void NotificateurDeplacements::desabonner(ObservateurDeplacement *p) {
   if (auto it = find(begin(obs_dep), end(obs_dep), p); it == end(obs_dep))
      throw PasAbonne{};
   else
      obs_dep.erase(it);
}
void NotificateurDeplacements::notifier
   (const Nomme &nomme, const Point &avant, const Point &apres) {
   for (auto &p : obs_dep)
      p->sur_deplacement(nomme, avant, apres);
}
```

L'application du schéma de conception observateur proposée ici est relativement classique : la méthode `notifier()` d'une instance de `NotificateurDeplacements` signale à tous les observateurs qui y sont abonnés l'occurrence du déplacement de l'objet nommé `nomme` ayant bougé du point `avant` au point `apres`. C'est à chaque observateur de faire comme bon lui semble avec cette information.

Exemple d'observateur de déplacements : un commentateur

L'un des observateurs possibles pour les objets déplaçables serait un commentateur de déplacements, représenté par une instance de la classe `Commentateur`. Celui-ci, lorsqu'il se fera notifier de l'occurrence d'un déplacement, réagira en envoyant un message décrivant cet événement sur un flux.

Dans le cas du système que nous concevrons ici, nous appliquerons une stratégie où un seul commentateur s'abonnera pour commenter tous les déplacements.

Cette stratégie est une stratégie possible parmi plusieurs : on aurait pu apposer des observateurs de déplacements distincts à plusieurs objets différents sur une base individuelle, par exemple, ou utiliser un même observateur pour tous les personnages et un autre pour tous les véhicules.

```
#ifndef COMMENTATEUR_H
#define COMMENTATEUR_H
#include "SuiviDeplacements.h"
#include "Population.h"
#include "Point.h"
#include "NommeFwd.h"
#include <iosfwd>
class Commentateur : public ObservateurDeplacement {
   std::ostream &os_;
public:
   Commentateur(ostream &os) : os_{os} {
      Population::get().ajouter_surveillant(this);
   }
   void sur_deplacement
      (const Nomme&, const Point&, const Point&);
};
#endif
```

Dans le but de garder le modèle simple, notre système inscrira tous les personnages dans un singleton de la classe `Population` (voir plus bas), et le commentateur proposé ici s'abonnera aux déplacements de tous les personnages en passant par ce singleton.

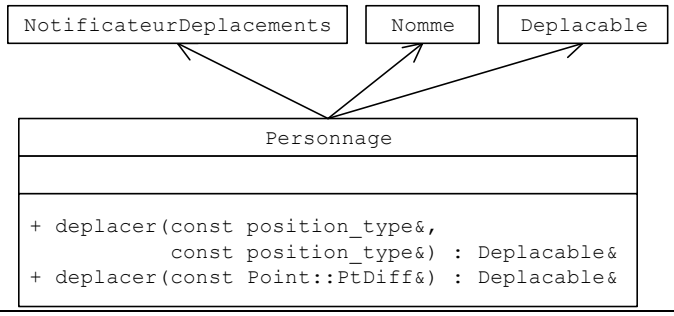
Le code de `Commentateur.h` est visible ci-dessus, à droite, alors que le code de `Commentateur.cpp` suit ci-dessous.

```
#include "Commentateur.h"
#include "Nomme.h"
#include "Point.h"
#include <iostream>
using std::endl;
void Commentateur::sur_deplacement
   (const Nomme &nomme, const Point &avant, const Point &apres) {
   os_ << nomme.nom() << " passe de " << avant << " à " << apres << endl;
}
```

Un personnage

Un personnage, dans notre système, sera une instance de la classe `Personnage`. Cette classe sera simple, profitant de la mécanique de l'héritage multiple et dérivant à la fois de `Nomme`, de `NotificateurDeplacements` et de `Deplacable`.

Dans `Personnage`, pour les besoins de notre système, nous raffinerons les méthodes `deplacer()` de la classe `Deplacable` pour y harmoniser le concept *se déplace* et le concept *notifie de ses déplacements*.



Le code de `Personnage.h` suit.

```
#ifndef PERSONNAGE_H
#define PERSONNAGE_H
#include "Population.h"
#include "SuiviDeplacements.h"
#include "Nomme.h"
#include "Deplacable.h"
#include <string>
struct Personnage : Nomme, Deplacable, NotificateurDeplacements {
   using ptdiff_t = Point::PtDiff;
   Personnage(const std::string &nom) : Nomme{nom} {
      Population::get().inscrire(this); // envisager une fabrique
   }
   ~Personnage() {
      Population::get().desinscrire(this);
   }
   Deplacable& deplacer(const position_type &dx, const position_type &dy) {
      return deplacer(ptdiff_t{dx, dy});
   }
   Deplacable& deplacer(const ptdiff_t &dPos) {
      const Point avant{position()};
      Deplacable::deplacer(dPos);
      notifier(*this, avant, position());
      return *this;
   }
};
#endif
```

Aucun `Personnage.cpp` n'est requis, l'essentiel de la mécanique étant codée de manière générale dans les classes parent.

Remarquez au passage que le constructeur et le destructeur de `Personnage` procèdent respectivement à l'inscription et à la désinscription du personnage en question dans le singleton de la classe `Population`.

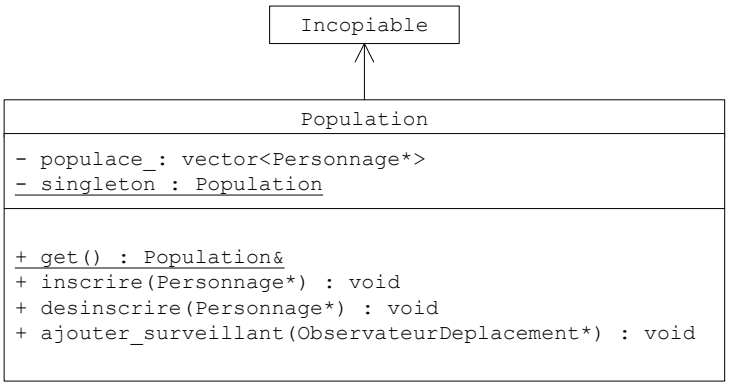
Ce détail technique aurait pu être pensé différemment (voir les exercices en fin de section) mais a pour but de faciliter la tâche de répertorier, au besoin, tous les personnages du système, par exemple pour permettre à un observateur comme notre commentateur de s'abonner à leur service de notification.

Dans un programme semblable mais qui serait soumis à des contraintes de multiprogrammation, cette façon de faire aurait été dangereuse puisque, avant la fin de la construction, l'objet n'est pas encore pleinement construit et puisqu'il s'agit d'un objet polymorphique.

Nous couvrirons ce problème et ses solutions dans un document ultérieur, mais mentionnons qu'il s'agirait ici d'un bon cas d'application du schéma de conception Fabrique.

Un singleton pour représenter la population

Pour faciliter la mécanique de notre système, tel que mentionné à quelques reprises un peu plus haut, nous utiliserons un singleton de la classe `Population` pour répertorier tous les personnages. Ces derniers s'inscriront dans la population lors de leur création et s'en désinscriront lors de leur destruction.



Un observateur de déplacements pourra s'abonner, en passant par la méthode `ajouter_surveillant()` du singleton, aux services de fournisseurs d'événements de déplacement de tous les personnages.

Le code de `Population.h` suit.

```
#ifndef POPULATION_H
#define POPULATION_H
class Personnage;
struct ObservateurDeplacement;
#include "Incopiable.h"
#include <vector>
class Population : Incopiable {
   std::vector<Personnage*> populace;
   static Population singleton;
   Population() = default;
public:
   static Population &get() noexcept {
      return singleton;
   }
   class DejaInscrit {};
   class PasInscrit {};
   void inscrire(Personnage*);
   void desinscrire(Personnage*);
   void ajouter_surveillant(ObservateurDeplacement*);
};
#endif
```

Le code de `Population.cpp` sera:

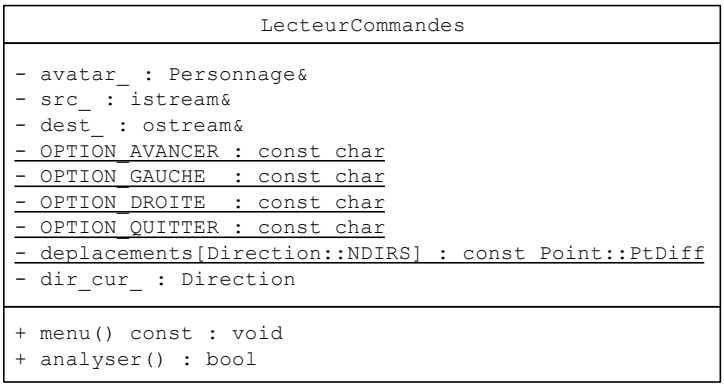
```
#include "Population.h"
#include "SuiviDeplacements.h"
#include "Personnage.h"
#include <algorithm>
using std::find;
Population Population::singleton;
void Population::inscrire(Personnage *p) {
   if (find(begin(populace), end(populace), p) != end(populace))
      throw DejaInscrit{};
   populace.push_back(p);
}
void Population::desinscrire(Personnage *p) {
   if (auto it = find(begin(populace), end(populace), p); it == end(populace))
      throw PasInscrit{};
   else
      populace.erase(it);
}
void Population::ajouter_surveillant(ObservateurDeplacement *p) {
   for(auto &q : populace)
      p->abonner(q);
}
```

Contrôler un personnage : un lecteur de commandes

Enfin, il nous faudra un mécanisme pour contrôler cet avatar que sera le personnage. Dans notre système, ce contrôleur sera une instance de la classe `LecteurCommandes`. Dans notre système, un `LecteurCommandes` contrôlera un seul personnage : l'avatar.

Ce contrôleur sera chargé de se souvenir de la direction vers laquelle pointe le personnage sous sa gouverne (un choix sujet à débats; voir les exercices de fin de section).

Le contrôleur offrira aussi une mécanique de menu pour l'usager et d'analyse des commandes entrées par ce dernier. Ce contrôleur est simple (simpliste!), mais nous suffira.



Le code de `LecteurCommandes.h` suit.

```
#ifndef LECTEUR_COMMANDES_H
#define LECTEUR_COMMANDES_H
#include "Point.h"
#include "Personnage.h"
#include "Direction.h"
#include <iosfwd>
class LecteurCommandes {
   Personnage &avatar;
   static const char OPTION_AVANCER, OPTION_GAUCHE, OPTION_DROITE, OPTION_QUITTER;
   static const Point::PtDiff deplacements[Direction::NDIRS];
   Direction dir_cur;
   std::istream &src;
   std::ostream &dest;
public:
   LecteurCommandes(Personnage &avatar, std::istream &src, std::ostream &dest)
      : avatar{avatar}, dir_cur{Direction::Sud}, src{src}, dest{dest} {
   }
   void menu() const;
   bool analyser();
};
#endif
```

Le code de `LecteurCommandes.cpp` sera :

```
#include "LecteurCommandes.h"
#include "Point.h"
#include <locale>
#include <iostream>
// ...using...
const char
   LecteurCommandes::OPTION_AVANCER = 'A', LecteurCommandes::OPTION_DROITE = 'D',
   LecteurCommandes::OPTION_GAUCHE = 'G', LecteurCommandes::OPTION_QUITTER = 'Q';
const Point::PtDiff LecteurCommandes::deplacements[Direction::NDIRS] {
   // Est, Nord, Ouest, Sud
   { 1, 0 }, { 0, -1 }, { -1, 0 }, { 0, 1 }
};
void LecteurCommandes::menu() const {
   dest << "Options: Avancer (" << OPTION_AVANCER << "), Gauche (" << OPTION_GAUCHE
        << "), Droite (" << OPTION_DROITE << "), Quitter (" << OPTION_QUITTER << "): ";
}
bool LecteurCommandes::analyser() {
   bool poursuivre = true;
   menu();
   char c;
   src >> c;
   switch(toupper(c, locale{""})) {
   case OPTION_AVANCER:
      avatar.deplacer(deplacements[static_cast<int>(dir_cur)]); break;
   case OPTION_GAUCHE:
      --dir_cur; break;
   case OPTION_DROITE:
      ++dir_cur; break;
   case OPTION_QUITTER:
      poursuivre = false; break;
   default:
      cerr << "Option invalide\n";
   }
   return poursuivre;
}
```

Remarquez au passage l'emploi d'un tableau en tant qu'attribut de classe constant, initialisé avant même la création de la première instance de `LecteurCommandes`. L'indice de chaque élément du tableau correspond à une direction possible (voir la classe `Direction`, plus haut) et dont chaque élément correspond à la valeur de `Point::PtDiff` d'un déplacement dans cette direction dans un espace comme l'écran console.

Cette technique de programmation permet, à un coût très faible, de trouver en temps constant le déplacement correspondant à une touche donnée dans l'espace où se situe le personnage. On aurait aussi pu l'appliquer à d'autres endroits (je vous laisse y réfléchir).

Notez que le recours aux opérateurs `++` et `--` sur des instances de `Direction` réalise une rotation en fonction de la rose des vents. Ceci allège l'écriture par laquelle nous faisons passer l'avatar d'une `Direction` à l'autre lors d'un pivot.

Si vous estimez que cette application des opérateurs n'est pas suffisamment claire, alors utilisez plutôt les méthodes `gauche()` et `droite()` d'une `Direction`.

Suivre la mécanique

Au début de l'exécution du programme, le personnage est à l'origine de l'espace dans lequel il navigue. Conceptuellement, pour nous, ceci correspondra au coin en haut et à gauche de l'écran console. Le personnage pointera initialement vers le sud (vers le bas), ce qui lui permet d'avancer sans sortir de l'espace.

Chaque fois qu'un usager appuiera sur une touche valide, le contrôleur indiquera au personnage qu'il doit se déplacer et de combien. Le personnage en prendra immédiatement acte et notifiera tous ses observateurs.

Le personnage se déplacera. Si ce déplacement le place hors de l'espace, une exception sera levée par la méthode de validation d'un point dans l'espace et le programme se terminera.

Exercices – Série 00

EX00 – Il est un peu agaçant que le personnage ne soit pas conscient de sa propre direction. Serait-il possible de concevoir une classe `Dirigeable` qui encapsulerait le concept d'*aller dans une direction*? Quel serait l'impact de cette décision sur les autres classes du système, par exemple `Direction`, `Deplacable`, `Personnage`, `Point` et `LecteurCommandes`?

EX01 – Pouvez-vous modifier `Direction` pour que cette classe permette de représenter la direction avec une granularité plus fine? Quel sera l'impact de cette modification sur le couplage entre `Direction` et `LecteurCommandes`? Pourrait-on remodeler ces deux classes pour réduire le niveau de couplage entre elles et faire en sorte qu'un tel changement devienne local à la classe `Direction`? Si oui, expliquez comment faire pour y arriver (ou mieux : faites-le!). Sinon, expliquez pourquoi.

EX02 – Peut-on concevoir une classe `Individu` dont tous les dérivés s'inscriraient à la construction dans une population donnée et se désinscriraient à la destruction de cette population?

EX03 – Remplacez `Commentateur` par un observateur qui affiche en tout temps le personnage à l'écran à l'endroit où il se trouve. Quelles sont les ramifications de votre implémentation sur les autres classes du système?

EX04 – La classe `LecteurCommandes`, lorsqu'elle constate une erreur, projette présentement un message sur un flux standard prévu à cette fin (`std::cerr`). Modifiez-la pour que le code client puisse indiquer une stratégie de traitement d'erreurs qui lui convienne. Quel est l'impact de votre choix d'implémentation sur le couplage entre classes dans votre système? À quel point votre solution est-elle flexible?

EX05 – Modifiez `Deplacable` pour que cette classe soit générique sur la base du type de point, et analysez les conséquences de votre choix d'implémentation.

EX06 – Selon vous, quelle est le meilleur design pour faire en sorte qu'un `Point` par défaut soit à l'origine de son lieu?

EX07 – Le fichier `SuiviDeplacements.h` montre qu'il aurait pu être utile d'avoir une déclaration *a priori* pour `Point`. Faites en sorte qu'un fichier léger, `PointFwd.h`, puisse être inclus par le code client qui ne souhaite pas inclure toute la mécanique de `Point.h`.

Comprendre les traits

Il arrive qu'on souhaite intégrer objets et types primitifs en utilisant des techniques OO comme par exemple le recours à des types internes et publics ou le recours à des méthodes.

Par exemple, imaginons qu'un programme souhaite déterminer si un type donné est un type entier non signé. Il serait intéressant de pouvoir invoquer une méthode sur une instance de ce type (ou une méthode de classe appropriée), par exemple `est_entier_non_signe()`, mais :

- les types primitifs n'ont pas de méthodes (règle générale : en C#, par exemple, les types primitifs sont des alias pour des `struct` qui sont pourvus de méthodes); et
- il est impossible de prévoir *a priori* tous les services potentiellement envisageables pour un type donné, quel qu'il soit.

Les traits, que nous explorerons ici, sont une manière non intrusive d'injecter, à l'aide de programmation générique, de l'information et des services sur la base de types. Une forme de documentation active injectée *a posteriori* sur la base des types impliqués. Ici, nous pourrions présenter `est_entier_non_signe` de la manière suivante :

- par défaut, la réponse est *non*; mais
- dans quelques cas, connus à la compilation, la réponse est *oui*.

Dans le cas à droite, le cas général est proposé tout en haut, et les quelques cas spécifiques suivent un peu plus bas. Remarquez que pour deux types, soit `char` et `wchar_t`, nous déduisons le caractère signé ou non signé du type à partir de calculs statiques (dans le cas de `char`, le caractère signé ou non du type dépend de la plateforme; dans le cas de `wchar_t`, je ne m'en souviens tout simplement jamais).

```
//
// Tous ces traits existent de manière standard
// dans <type_traits>, bien que sous d'autres
// noms. Utilisez-les!
//
template <class T, T val>
   struct constante_entiere {
      using type = T;
      static const type value = val;
   };
template<bool B> constante_boolenne
   : contante_entiere<bool, B> { };
using vrai = constante_boolenne<true>;
using faux = constante_boolenne<false>;
template <class>
   struct est_entier_non_signe : faux { };
template <>
   struct est_entier_non_signe<bool> : vrai {};
template <>
   struct est_entier_non_signe<unsigned char>
      : vrai { };
template <>
   struct est_entier_non_signe<unsigned short>
      : vrai { };
template <>
   struct est_entier_non_signe<unsigned int>
      : vrai { };
template <>
   struct est_entier_non_signe<unsigned long>
      : vrai { };
template <>
   struct est_entier_non_signe<char>
      : constante_booleenne<
           (static_cast<char>(-1) >= 0)
        > {
   };
template <>
   struct est_entier_non_signe<wchar_t>
      : constante_booleenne<
           (static_cast<wchar_t>(-1) >= 0)
        >
   {
   };
```

Exemple : `std::char_traits`

Le programme proposé à droite utilise les traits pour afficher un message explicatif sur la base des types auxquels il est appliqué, et montre quelques exemples d'appels.

Les traits, que nous explorerons ici, sont une manière non intrusive d'injecter, par programmation générique, de l'information et des services sur la base de types. De la documentation active injectée *a posteriori* sur la base des types impliqués.

À l'aide de traits, il est possible :

- d'écrire du code applicable à une gamme variée de types, primitifs et objets;
- d'intégrer des hiérarchies de classes disjointes sans les modifier;
- de documenter des types à même le code; *etc*.

Cette technique est précieuse.

```
#include <string>
#include <iostream>
// ...using...
template <class T>
   void affichage(ostream &os, const string &nom) {
      os << nom;
      if constexpr (est_entier_non_signe<T>::value)
         os << " est un entier non signé" << endl;
      else
         os << " n'est pas un entier non signé" << endl;
   }
int main() {
   affichage<int>(cout, "int");
   affichage<unsigned int>(cout, "unsigned int");
   affichage<char>(cout, "char");
   affichage<signed char>(cout, "signed char");
   affichage<unsigned char>(cout, "unsigned char");
   affichage<wchar_t>(cout, "wchar_t");
   affichage<float>(cout, "float");
   affichage<string>(cout, "string");
}
```

Le concept de trait est extrêmement abstrait, alors allons-y d'un exemple dont l'utilisation est très répandue et pour lequel il existe beaucoup de documentation. Les types que nous utiliserons pour cette illustration sont tous soit primitifs, soit placés dans l'espace nommé `std`.

La classe `string` est un `basic_string` appliqué à un `char`, et la classe `wstring` est un `basic_string` appliqué à un `wchar_t`.

Ce qui fait fonctionner élégamment la mécanique de `basic_string` avec ces deux types et une infinité potentielle d'autres types est que *le concept de caractère* est défini à un niveau plus élevé que celui des types primitifs `char` et `wchar_t`. La classe `char_traits`[34] exprime, sous forme `OO`, des caractéristiques d'un type comme `char` ou `wchar_t` qui n'est pas nécessairement un objet.

Ce que contient un `char_traits`

Que juge-t-on nécessaire de connaître sur un caractère pour être en mesure d'opérer correctement sur lui? Pour répondre à cette question, il suffit d'examiner la classe `char_traits`.

On y constate qu'il faut savoir :

- à partir de types internes et publics, quel est le type associé au caractère (`char_type`), le type associé à la conversion d'un caractère en entier (`int_type`), le type associé à la position d'un caractère dans un flux (`pos_type`), le type associé à la différence entre deux positions de caractères dans un flux (`off_type`) et le type associé à l'état d'un caractère, surtout pour la gestion d'encodages sophistiqués comme Unicode (`state_type` que plusieurs négligent);
- comment affecter un caractère à un autre, vérifier l'égalité de deux caractères ou établir une relation d'ordre entre deux caractères;
- comment réaliser une série d'opération de base sur une séquence de caractères (copier, comparer, trouver la longueur, *etc*.); et
- comment représenter la fin de fichier à l'aide d'un caractère.

On pourra donc définir ce que signifie `basic_string` appliqué à un type `T` donné dans la mesure où il existe une définition correcte pour `char_traits<T>`.

Les traits sont un idiome de support, près des métadonnées mais pensé de manière à maximiser l'efficacité dans le but de faciliter la conception d'autres types.

[34] Pour en lire plus sur ce sujet, voir [StlChTr0] et (à un niveau d'abstraction plus élevé) [StlChTr1].

Exemple détaillé : une moyenne optimale

Imaginons un cas tout simple : un programme souhaite offrir une fonction capable d'évaluer la moyenne des éléments d'une séquence de valeurs d'un type donné. Intuitivement, en prenant un tableau pour conteneur, on obtiendrait un code semblable à celui proposé à droite.

Notez que l'expression de notre version de l'algorithme `moyenne()` manque de généricité puisqu'elle ne s'applique qu'à des tableaux (un itérateur sur une séquence de `T` y est un `T*`, qualifié constant dans ce cas-ci).

Cette implémentation naïve comporte quelques problèmes. Celui de la généricité, susmentionné, mais aussi le caractère occasionnellement erroné de son résultat : ici, la moyenne des valeurs insérées dans le tableau `Tab` devrait être `5,5`, or le programme affichera `5` tout simplement.

```
template <class T>
   T moyenne(const T *debut, const T *fin) {
      T somme = {};
      int n = 0;
      for (; debut != fin; ++debut) {
         somme += *debut;
         ++n;
      }
      return somme / n;
   }
#include <iostream>
#include <algorithm>
#include <numeric>
// ...using...
int main() {
   const int N = 10;
   int tab[N];
   iota(begin(tab), end(tab), 1);
   cout << moyenne(begin(tab),end(tab));
}
```

La raison de cette erreur de calcul est que le type `T` sert de type pour le cumul et que le nombre d'éléments comptabilisés par l'algorithme est représenté par un entier.

Un raffinement intrusif de cette stratégie serait de mettre au point un algorithme dont la généricité repose sur deux types plutôt qu'un seul. Ce n'est malheureusement pas une stratégie recommandable puisqu'elle alourdit le code client, qui se trouve alors forcé de compliquer son invocation, ce qui provoque une forme d'intrusion dans la mécanique de l'algorithme de calcul d'une moyenne.

On souhaiterait à la fois du code client plus simple, plus élégant, et un algorithme dont la démarche est plus autonome.

```
template <class Res, class T>
   Res moyenne(const T *debut, const T *fin) {
      Res somme = {};
      int n = 0;
      for (; debut != fin; ++debut) {
         somme += *debut;
         ++n;
      }
      return somme / n;
   }
#include <iostream>
#include <algorithm>
#include <numeric>
// ...using...
int main() {
   const int N = 10;
   int tab[N];
   iota(begin(tab), end(tab), 1);
   cout << moyenne<float>(begin(tab), end(tab));
}
```

De plus, le problème de la généricité demeure. Nous souhaitons un algorithme qui puisse fonctionner avec tout type d'itérateur, qu'il s'agisse d'un pointeur dans un tableau ou d'un objet capable d'itérer à travers une séquence générique.

Si nous modifions l'algorithme initial pour qu'il s'exprime en termes d'itérateur et qu'il déduise le type d'un élément de la séquence à travers les types internes et publics d'un itérateur (son type `value_type`, par convention), nous obtenons alors un algorithme applicable à tous les itérateurs exposant un type public nommé `value_type`, or, il se trouve que les pointeurs servant d'itérateurs sur des tableaux bruts n'exposent pas de types internes et publics, ce qui ne fait que déplacer le problème de la généricité insuffisante.

De plus, nous voilà revenus au problème initial qui confond type d'un élément et type de la moyenne. Encore une fois, nous pourrions ajouter un type supplémentaire à l'algorithme `moyenne()`, avec les mêmes réserves.

Nous avons donc un sérieux problème :

```
template <class It>
   typename It::value_type
      moyenne(It debut, It fin) {
      typename It::value_type somme = {};
      int n = 0;
      for (; debut != fin; ++debut) {
         somme += *debut;
         ++n;
      }
      return somme / n;
   }
#include <iostream>
#include <vector>
#include <algorithm>
#include <numeric>
// ...using...
int main() {
   const int N = 10;
   vector<int> v(N);
   iota(begin(v), end(v), 1);
   cout << moyenne(begin(v), end(v));
}
```

- si les itérateurs sont des objets, alors ils peuvent inclure des définitions de types internes et publics susceptibles de nous aider à déterminer les bons types à utiliser pour une valeur, un cumul ou d'autres manœuvres (par contre, si les itérateurs sont des types primitifs, nous n'avons pas ce luxe);
- de toute manière, une classe étant une entité fermée pour la modification, on ne peut ajouter nos propres types à l'intérieur de types existants. Si un itérateur n'offre pas l'une des définitions de types sur lesquelles nous comptons, nous voilà bien mal pris;

```
// moyenne() tel que que ci-dessus
int main() {
   const int N = 10;
   int tab[N];
   iota(begin(tab), end(tab), 1);
   cout << moyenne(begin(tab), end(tab));
}
```

- s'ajoute à cela le fait que les caractéristiques descriptives d'un type (p. ex. : quel est le bon type pour évaluer la moyenne d'une séquence à partir du type de la valeur d'un élément de la séquence?) sont en nombre potentiellement infini. Il est impensable de couvrir tous les cas possibles *a priori*.

La maxime à coût zéro (MC0)

Nous viserons, chaque fois où ce sera possible, une solution à coût zéro en espace et en temps d'exécution, quitte à ce que le temps de compilation soit plus long. Rien n'est gratuit.

La technique des traits

Nous sommes donc à la recherche d'une stratégie nous permettant d'*étendre la capacité descriptive d'un type sans avoir recours à une stratégie intrusive ou à la modification de ce type*.

L'objectif sous-jacent est de nous doter de la capacité d'écrire des algorithmes et des types génériques aussi souples et aussi performants que possibles sans modifier les types auxquels ces algorithmes et ces types génériques s'appliquent.

Un **trait**, pour donner une définition à la fois opérationnelle et un peu simple (pour ne pas dire simpliste), est **une entité générique capable d'offrir de l'information statique sur un type**. Cette définition nous servira amplement ici : avec elle, nous ferons de petits miracles. Les traits font d'ailleurs partie des techniques et concepts de base de la métaprogrammation, dont nous reparlerons.

Simplifions le cas qui nous intéresse ici et présumons que l'information complémentaire pertinente à l'algorithme `moyenne()` applicable à un type d'itérateur donné soit de savoir le type à virgule flottante correspondant le mieux au type valeur d'un itérateur donné (que nous nommerons son `reel_t`).

Une des applications les plus importantes des traits est de documenter des types et de les accompagner pour guider l'exécution des programmes.

Cette approche, développée par **Nathan Myers**, était autrefois nommée *approche par bagage* du fait qu'elle ajoutait de l'information sémantique (le bagage en question) à des types existants.

La bibliothèque standard de C++ et les bibliothèques à très haute performance comme *Blitz++*, *Boost* et *Loki* utilisent massivement cette technique. Ce n'est pas un accident.

Présumons aussi que nous ne nous intéressions (pour le moment) qu'aux moyennes applicables à des types primitifs ou à des entités susceptibles d'être converties en un type primitif. Pour nos fins, le `reel_t` d'un type `T` devra être équivalent à `float` sauf dans les cas des types `long` et `double` pour lesquels nous privilégierons le type `double`[35].

[35] Rappel : il s'agit d'une simplification à titre d'illustration, pas d'une prise de position mathématique. Nous cherchons ici un point de départ simple pour la réflexion et la présentation du concept, sans plus.

Nous nommerons `TraitsCalcul` la classe[36] décrivant la caractéristique `reel_t` de type à virgule flottante à utiliser quand on parle d'un type `T` donné.

Une version préliminaire pourrait aller comme suit :

- le cas par défaut (le *cas général*) pour un type `T` est de définir `reel_t` comme équivalent à `float`; puis
- des spécialisations partielles du modèle générique, appliquées respectivement aux types `long` et `double`, indiquant que dans ces cas, le type `reel_t` correspondra au type `double`.

```
template <class T>
   struct TraitsCalcul {
      using reel_t = float;
   };
template <>
   struct TraitsCalcul<double> {
      using reel_t = double;
   };
template <>
   struct TraitsCalcul<long> {
      using reel_t = double;
   };
```

Remarquez tout de suite que `TraitsCalcul<std::string>::reel_t` équivaut à `float` selon cette définition, ce qui est peu recommandable. Nous verrons comment éviter cette situation dans ***Provoquer des erreurs***, plus bas.

Cette technique est très légère : nous n'avons fait que nommer des choses, sans instancier quoi que ce soit et sans invoquer quelque sous-programme que ce soit. Nous n'avons rien fait qui ait le moindre coût en espace ou en temps d'exécution. Muni de ce trait, voyons comment il nous serait possible de raffiner notre algorithme de calcul de la moyenne des éléments d'une séquence.

Le type de la fonction générique `moyenne()` sera celui du trait `reel_t` applicable au type `T` d'un élément de la séquence.

J'ai choisi ici de cumuler la somme des éléments de la séquence dans une variable cumulative dont le type est `T`. Je limite le recours au type `TraitsCalcul<T>::reel_t` pour réduire les problèmes de précision. Cela dit, cette technique est perfectible (nous y reviendrons).

Remarquez que le programme de test peut déduire à lui seul le type générique `It` mais pas le type `T` qui doit (pour le moment) être suppléé par le code client.

Nous n'avons fait qu'insérer des types (donc des noms) dans le programme. Celui-ci n'a ni grossi, ni ralenti. Les traits constituent une technique statique, prise en charge à la compilation, et ont un coût d'exécution nul. Ils respectent donc la maxime `MC0`.

```
// ...
template <class T, class It>
   typename TraitsCalcul<T>::reel_t
      moyenne(It debut, It fin) {
      using reel_t = typename
         TraitsCalcul<T>::reel_t;
      T somme = {};
      int n = 0;
      for (; debut != fin; ++debut) {
         somme += *debut;
         ++n;
      }
      return static_cast<reel_t>(somme)/ n;
   }
// ...inclusions et using...
int main() {
   const int N = 10;
   vector<int> v(N);
   iota(begin(v), end(v), 1);
   cout << moyenne<int>(begin(v), end(v));
}
```

Le code de `moyenne()` plus haut pourrait encore être raffiné, mais nous serons mieux équipés pour y arriver un peu plus loin, dans la section ***L'algorithme `moyenne()` revisité***.

[36] En fait, on a souvent (pas toujours) recours à un `struct` pour définir des traits puisque la plupart d'entre eux sont des types publics ou des constantes publiques de classe.

Provoquer des erreurs

Notons ici que nous avons défini que tout type `T` a un trait `TraitsCalcul<T>::reel_t` susceptible de servir (entre autres) au calcul d'une moyenne sur une séquence de `T`.

Notons aussi que cela implique qu'une programmeuse ou un programmeur pourrait, accidentellement, chercher à calculer la moyenne d'éléments d'un type pour lequel cela n'a pas vraiment de sens (le type `std::string` par exemple) et obtiendrait probablement une erreur de compilation étrange ayant trait à la conversion d'une `std::string` en `float`.

Raffinons donc `TraitsCalcul` pour que :

- seuls les types pour lesquels nous estimons raisonnable qu'il existe ce trait soient supportés; et que
- toute tentative d'utiliser un type pour lequel ce groupe de traits ne semble pas s'appliquer génère une erreur à la compilation (donc aucun coût à l'exécution).

S'il advient qu'un type pour lequel un `TraitsCalcul` serait raisonnable et pour lequel ce trait n'est pas défini, il suffit alors d'ajouter une spécialisation pour couvrir ce cas oublié. Les traits sont, par définition, découplés et extensibles.

Bien qu'il soit probable que plusieurs trouvent cet énoncé dérangeant, il se trouve que **les erreurs de compilation sont une bonne chose**. Elles permettent de dépister les problèmes avant la livraison du code et coûtent ainsi *énormément* moins cher que les erreurs d'exécution.

Pour les fins de la démonstration, nous présumerons que les types entiers utilisés traditionnellement à des fins non numériques (`char`, `signed char`, `bool`, `wchar_t`, les types énumérés et ainsi de suite) ne doivent pas avoir de `TraitsCalcul`.

Il existe des traits standards sur les types numériques, décrits par la classe standard `std::numeric_limits` de la bibliothèque `<limits>`, sur laquelle nous reviendrons dans la section ***Applications des traits*** (plus loin), à partir desquels nous pourrions aussi travailler.

Définissons donc cette mouture de `TraitsCalcul` de la manière suivante :

- sur le plan générique (tout en haut, en gras), `TraitsCalcul` est un `struct` applicable à un type. Remarquez que la version générique ne fait qu'exister : on n'en connaît pas la nature ou la structure. Il s'agit d'un type incomplet, comme le sont les déclarations *a priori*. S'en servir en tant qu'elle-même est donc une erreur;
- par la suite, sur le plan spécifique, nous offrons des cas particuliers pour les types qui nous intéressent (et oui, on peut faire plus simple, mais il faut pour ce faire pousser plus à fond la métaprogrammation, ce que nous ne ferons pas ici);
- qu'il y ait plusieurs cas particulier importe peu puisque le programme ne grossira pas. Ce ne sont que des noms, des *alias*… de pures idées.

Armés de ces définitions, utiliser notre petit programme de test sur une séquence de `int` fonctionnera toujours correctement (en utilisant `float` comme type de retour pour `moyenne()`), mais appliquer le même programme à une séquence de `bool` ou de `std::string` échouera à la compilation.

```
template <class T>
   struct TraitsCalcul;
template <>
   struct TraitsCalcul<short> {
      using reel_t = float;
   };
template <>
   struct TraitsCalcul<unsigned short> {
      using reel_t = float;
   };
template <>
   struct TraitsCalcul<int> {
      using reel_t = float;
   };
template <>
   struct TraitsCalcul<unsigned int> {
      using reel_t = float;
   };
template <>
   struct TraitsCalcul<float> {
      using reel_t = float;
   };
//
// ... et ainsi de suite pour long,
// unsigned long, long long,
// unsigned long long, double et
// long double
//
```

Enrichir les traits

Le choix de cumuler la somme de valeurs de type `T` sur une variable de type `T` est contestable puisqu'il invite aux débordements de capacité.

Une possibilité pour adopter un comportement plus raisonnable est de déterminer un autre trait dans le groupe `TraitsCalcul`, que nous nommerons `cumul_t`, pour représenter le type à utiliser lors d'un cumul de valeurs de type `T`.

Les choix de `cumul_t` pour un type `T` donné peuvent varier (par exemple, on pourrait décider d'utiliser `long double` pour cumuler des double et d'utiliser un type non portable pour cumuler des `long`). Cela importe peu puisque les algorithmes devraient fonctionner quand même (présumant que les sémantiques opératoires des types utilisés soient conformes aux attentes).

On pourrait dans certains cas vouloir utiliser des constructions complexes comme le type `LARGE_INTEGER` de *Win32* pour cumuler certains types `T`. Si ces types n'offrent pas la sémantique opératoire d'un type primitif (par exemple dans le cas où le type désiré ne supporte pas le cumul par l'opérateur `+=`), il suffit de les enrober dans un type de notre cru qui, lui, offre cette sémantique.

De tels enrobages constituent une application pertinente du bien connu schéma de conception Décorateur [hdDeco].

Les traits peuvent aussi décrire des constantes connues à la compilation et exposer des méthodes de classes. Nous en verrons quelques exemples dans la section ***Applications des traits***, plus bas.

```
template <class T>
   struct TraitsCalcul;
template <> struct TraitsCalcul<short> {
   using reel_t = float;
   using cumul_t = int;
};
template <>
   struct TraitsCalcul<unsigned short> {
   using reel_t = float;
   using cumul_t = unsigned int;
};
template <> struct TraitsCalcul<int> {
   using reel_t = float;
   using cumul_t = long;
};
template <>
   struct TraitsCalcul<unsigned int> {
   using reel_t = float;
   using cumul_t = unsigned long;
};
template <> struct TraitsCalcul<float> {
   using reel_t = float;
   using cumul_t = double;
};
//
// ... et ainsi de suite pour long,
// unsigned long, long long,
// unsigned long long, double et
// long double
//
```

Sachant cela, l'algorithme `moyenne()` pourrait maintenant s'écrire comme proposé à droite. Remarquez qu'encore une fois, nous n'avons aucun coût en espace ou en temps de calcul à encaisser. Le gain en généralité se fait à coût zéro.

Malgré les qualités de cette solution, il nous reste un certain nombre d'irritants à adresser avant qu'elle ne puisse être considérée satisfaisante. En particulier, le type d'une valeur (le type `T`) doit pouvoir être déduit du type d'itérateur pour que le code client soit simple et naturel, et cette déduction doit se faire (surprise!) elle aussi à coût zéro.

Le fait que nous comptions les éléments considérés dans le calcul de la moyenne à l'aide d'une variable de type `int` est aussi, ce qui peut surprendre, un irritant important. Nous y reviendrons sous peu quand nous examinerons le concept de *distance entre deux itérateurs*.

Pour voir comment arriver au niveau de qualité de code attendu, il nous faudra maintenant examiner certaines applications des traits dans la bibliothèque standard.

```
//
// TraitsCalcul (omis par économie)
//
template <class T, class It>
   typename TraitsCalcul<T>::reel_t
      moyenne(It debut, It fin) {
      // alias locaux, pour alléger l'écriture
      using cumul_t = typename
         TraitsCalcul<T>::cumul_t;
      using reel_t = typename
         TraitsCalcul<T>::reel_t;
      cumul_t somme = {};
      int n = 0;
      for (; debut != fin; ++debut) {
         somme += *debut;
         ++n;
      }
      return static_cast<reel_t>(somme) / n;
   }
#include <iostream>
#include <vector>
#include <algorithm>
#include <numeric>
// ...using...
int main() {
   const int N = 10;
   vector<int> v(N);
   iota(begin(v), end(v), 1);
   cout << moyenne<int>(begin(v), end(v));
}
```

Usages contemporains

Nos pratiques avec les traits se sont raffinées au fil du temps, et il est de moins en moins fréquent que l'on trouve des traits regroupant plusieurs types, privilégiant en général des traits plus unitaires. Ainsi, plutôt que ceci :

```
// ...
template <class>
   struct TraitsCalcul;
template <>
   struct TraitsCalcul<int> {
      using cumul_t = long;
      using reel_t = float;
   };
// ...
```

…on trouvera habituellement cela :

```
// ...
template <class>
   struct cumul_traits;
template <>
   struct cumul_traits<int> {
      using type = long;
   };
// ...
template <class>
   struct reel_traits;
template <>
   struct reel_traits<int> {
      using type = float;
   };
// ...
```

Cette pratique est avantageuse à plusieurs égards. Entre autres, elle standardise la nomenclature des types en utilisant systématiquement le mot `type` pour les identifier.

De même, elle s'accompagne d'un enrichissement permis par C++ 11 et permettant de généraliser certaines pratiques de nommage :

```
// ...
template <class T>
   using cumul_t = typename cumul_traits<T>::type;
template <class T>
   using reel_t = typename reel_traits<T>::type;
// ...
```

Cette formulation d'un alias paramétrique n'était pas possible avec les `typedef` traditionnels du langage C, mais est possible depuis C++ 11 grâce à la syntaxe plus riche des alias avec `using`. Ainsi, pour l'algorithme général de moyenne, on pourrait remplacer ceci :

```
template <class T, class It>
   typename TraitsCalcul<T>::reel_t moyenne(It debut, It fin) {
      using cumul_t = typename TraitsCalcul<T>::cumul_t;
      using reel_t = typename TraitsCalcul<T>::reel_t;
      cumul_t somme = {};
      int n = 0;
      for (; debut != fin; ++debut) {
         somme += *debut;
         ++n;
      }
      return static_cast<reel_t>(somme) / n;
   }
```

…par cela :

```
template <class T, class It>
   reel_t<T> moyenne(It debut, It fin) {
      cumul_t<T> somme = {};
      int n = 0;
      for (; debut != fin; ++debut) {
         somme += *debut;
         ++n;
      }
      return static_cast<reel_t<T>>(somme) / n;
   }
```

L'allègement syntaxique est manifeste.

Depuis C++ 14, la vie est encore plus simple puisqu'il est possible de déduire le type de retour d'une fonction du type du premier return que l'on y trouvera (s'il y a plusieurs points de sortie, alors ils doivent tous retourner une valeur du même type).

Le code devient donc simplement :

```
template <class T, class It>
   auto moyenne(It debut, It fin) {
      cumul_t<T> somme = {};
      int n = 0;
      for (; debut != fin; ++debut) {
         somme += *debut;
         ++n;
      }
      return static_cast<reel_t<T>>(somme) / n;
   }
```

Bref coup d'œil sur le type `std::iterator_traits`

Plusieurs questions peuvent se poser quant à la nature d'un itérateur :

- quel est le type vers lequel cet itérateur pointe?
- si nous décrivions des catégories d'itérateurs en fonction de leur capacité opératoire, pour considérer par exemple d'une même famille les itérateurs qui permettent à la fois d'avancer et de reculer dans une séquence, dans quelle catégorie un itérateur donné apparaîtrait-il?
- quel est le type d'une distance entre deux itérateurs? Cette question est moins banale qu'il n'y paraît : à titre d'exemple, la distance entre deux itérateurs permettant de parcourir une liste chaînée serait évaluée en nombre de nœuds à traverser alors que la distance entre deux pointeurs est une différence d'adresses (type `std::ptrdiff_t`); *etc*.

Pour permettre de décrire efficacement un type d'itérateur donné, la technique des traits est des plus appropriées. Les traits attendus d'un type d'itérateur `It` sont d'ailleurs décrits dans la classe `iterator_traits<It>`, définie dans le fichier d'en-tête `<iterator>`, pour les classes les plus conventionnelles et pour les pointeurs.

Les algorithmes utilisant des itérateurs utilisent *beaucoup* ces traits. Par exemple, l'algorithme (simpliste) à droite prend deux itérateurs de même type décrivant une séquence, et retourne la valeur de l'élément de cette séquence ayant la plus petite valeur (au sens de l'opérateur `<` appliqué aux éléments pointés).

Cet algorithme plantera si la séquence est vide, donc si `debut==fin` au tout début. Voyez-vous pourquoi?

Le programme principal montre deux appels conformes à notre algorithme : l'un utilise des pointeurs (les extrémités logiques d'un petit tableau) et l'autre utilise les extrémités d'une `std::list`. La beauté de cet exemple est qu'il met de l'avant que l'algorithme déduit le type pointé à partir des traits des itérateurs, qu'il s'agisse d'un pointeur brut ou d'une classe relativement complexe.

```
#include <iterator>
// ...using...
template <class It>
   // depuis C++14, on écrirait auto ici
   typename iterator_traits<It>::value_type
      plus_petit_element(It debut, It fin) {
      auto val = *debut;
      for (++debut; debut != fin; ++debut)
         if (*debut < val) val = *debut;
      return val;
   }
#include <list>
#include <iostream>
int main() {
   int tab[] = { 3, -1, 2 };
   list<int> lst(begin(tab), end(tab));
   cout << plus_petit_element(
               begin(tab), end(tab)
           )
        << '\n'
        << plus_petit_element
             (begin(lst), end(lst))
        << endl;
}
```

Examinons maintenant une ébauche[37] de ce que contient la classe `std::iterator_traits` tel qu'on la retrouve dans la bibliothèque standard.

Remarquez trois catégories fondamentales, en commençant par la deuxième :

- si le type `T` auquel on applique un `iterator_traits` est un pointeur sur un `T` (donc un `T*`), alors le type d'une valeur est `T` et le type d'une différence est celui d'une distance entre pointeurs;
- la version pour un pointeur constant exprime le même concept à la constance des pointeurs et des références près; et
- si le type `It` n'est ni un pointeur, ni un pointeur constant, alors on utilise plutôt ses types internes et publics (qu'il est alors sage d'avoir défini).

```
template<class It>
   struct iterator_traits {
      // ...
      using value_type = typename
         It::value_type;
      using difference_type = typename
         It::difference_type;
      using pointer = typename
         It::pointer;
      using reference = typename
         It::reference;
   };
template<class T>
   struct iterator_traits<T *> {
      // ...
      using value_type = T;
      using difference_type = ptrdiff_t;
      using pointer = T*;
      using reference = T&;
   };
template<class T>
   struct iterator_traits<const T *> {
      // ...
      using value_type = T;
      using difference_type = ptrdiff_t;
      using pointer = const T*;
      using reference = const T&;
   };
```

Évidemment, les noms de types `T` et `It` ne sont distincts que parce que nous les voulons descriptifs. Nous aurions pu utiliser le nom `T` dans la première déclinaison de la classe standard `iterator_traits` comme dans les deux autres.

Si vous souhaitez définir vos propres itérateurs à l'aide de classes, vous avez trois options :

- définir les types internes et publics attendus dans votre classe et laisser s'appliquer les traits généraux d'`iterator_traits`;
- dériver votre classe d'une classe définissant les traits pour vous (il en existe une : la classe `std::iterator`, du fichier d'en-tête `<iterator>`); ou
- définir la classe à votre manière et lui adjoindre votre propre version spécialisée d'`iterator_traits` pour ce nouveau type.

Dans chaque cas, à qualité de code égale, le résultat sera le même.

[37] Il manque ici le concept de catégorie sur lequel nous reviendrons sous peu.

L'algorithme `moyenne()` peut utiliser `iterator_traits` pour ne dépendre que du type des itérateurs impliqués.

Puisque que les noms de types génériques deviennent vite complexes, on a habituellement recours à des *alias* locaux pour alléger l'écriture du code. Vous remarquerez donc la présence de trois instructions `typedef` dans la fonction générique `moyenne()`.

Le type de la fonction `moyenne()` étant défini à l'extérieur de la fonction, nous ne pouvons pas avoir recours à un *alias* local, ce qui explique le caractère verbeux du type de retour (en français : *le type à virgule flottante correspondant au type de la valeur pointée par un itérateur de type `It`*).

```
#include <iterator>
using std::iterator_traits;
template <class It>
   reel_t <
      typename std::iterator_traits<It>::value_type
   > moyenne(It debut, It fin) {
      using val_t = typename
         std::iterator_traits<It>::value_type;
      cumul_t<val_t> somme = {};
      int n = 0;
      for (; debut != fin; ++debut) {
         somme += *debut;
         ++n;
      }
      return static_cast<reel_t<val_t>>(somme)/n;
   }
```

Comme mentionné plus haut, depuis C++ 14, nous pouvons alléger l'écriture en utilisant `auto` à titre de type de retour. Cette nouvelle option ramène l'écriture d'algorithmes génériques à des dimensions plus… humaines.

```
#include <iterator>
template <class It>
   auto moyenne(It debut, It fin) {
      using val_t = typename
         std::iterator_traits<It>::value_type;
      cumul_t<val_t> somme = {};
      int n = 0;
      for (; debut != fin; ++debut) {
         somme += *debut;
         ++n;
      }
      return static_cast<reel_t<val_t>>(somme)/n;
   }
```

Tout cet effort de notation se trouve côté serveur (côté implémentation de l'algorithme).

Le code client, lui, se trouve fortement allégé : à partir du type des itérateurs utilisés à l'appel de la fonction `moyenne()`, tous les types jugés optimaux pour générer le code de `moyenne()` peuvent être déduits *dès la compilation du code client*.

```
// ...inclusions et using...
int main() {
   const int N = 10;
   vector<int> v(N)
   iota(begin(v), end(v), 1);
   cout << moyenne(begin(v), end(v));
}
```

Nous avons donc maintenant une version de `moyenne()` applicable à une séquence de d'éléments de n'importe quel type en fonction duquel nous estimons raisonnable d'exposer les traits `cumul_t` et `reel_t`, que ces éléments soient contenus dans un tableau, un vecteur, une liste chaînée ou tout autre conteneur susceptible d'être traversé à l'aide d'itérateurs.

Compter les éléments : le problème de la distance

Ce qui suit est fortement inspiré des textes de **Nicolai M. Josuttis**.

Notre calcul de moyenne peut encore être raffiné. En effet, nous comptons encore le nombre d'éléments en utilisant un compteur entier, ce qui implique une initialisation et plusieurs incrémentations alors que dans certains cas une simple soustraction entre la fin et le début aurait pu faire le travail.

Une version `moyenne()` qui se voudrait aussi rapide que possible du point de vue du calcul du nombre d'éléments dans la séquence pourrait être telle que proposé à droite.

Portez attention à ces détails :

- on y utilise un itérateur temporaire nommé `p`, pour garder `debut` intact; et
- on n'y compte plus les éléments avec un compteur; car
- le nombre d'éléments y est obtenu par simple arithmétique sur les itérateurs.

```
#include <iterator>
template <class It>
   auto moyenne(It debut, It fin) {
      using val_t = typename
         std::iterator_traits<It>::value_type;
      cumul_t<val_t> somme = {};
      for (auto p = debut; p != fin; ++p)
         somme += *p;
      return static_cast<reel_t<val_t>>(somme)/
                (fin - debut);
   }
```

Cet algorithme est applicable à toute une gamme d'itérateurs, soit ceux qui permettent de calculer efficacement la distance entre deux itérateurs à l'aide d'une soustraction. Cette catégorie inclut tous les itérateurs dits **à accès directs** (en anglais : les *Random Access Iterators*).

Pour d'autres catégories d'itérateurs, cet algorithme serait déraisonnablement long. Pour vous en convaincre. pensez à implémenter la distance en terme de soustraction pour un itérateur permettant de traverser une liste simplement chaînée : la complexité serait $O(n^2)$ où `n` est la distance entre les itérateurs.

Ainsi, pour les deux exemples ci-dessous, celui de gauche resterait légal mais celui de droite ne le serait pas, les itérateurs d'une `std::list` n'exposant pas un opérateur `-` conforme. Ce choix est technique : on pourrait l'implémenter mais il serait tellement lent que cela mènerait à du code inutilisable, alors vaut mieux ne pas l'offrir.

```
int main() {
   const int N = 10;
   vector<int> v;
   for (int i = 0; i < N; ++i)
      v.push_back(i + 1);
   // ok: les itérateurs supportent -
   cout << moyenne(begin(v), end(v));
}
```

```
int main() {
   const int N = 10;
   list<int> ls;
   for (int i = 0; i < N; ++i)
      ls.push_back(i + 1);
   // incorrect!
   cout << moyenne(begin(ls),end(ls));
}
```

La solution à ce problème est la fonction `std::distance()`. Cette fonction calcule la distance entre deux itérateurs *de la manière la plus efficace possible pour la catégorie d'itérateurs à laquelle ces itérateurs appartiennent*.

La fonction `std ::distance()` fait partie de la bibliothèque standard et est livrée dans `<iterator>`, mais son implémentation est des plus instructives. Nous verrons maintenant comment nous pourrions implémenter nous-mêmes cette fonction, et quelles applications des traits et de la `POO` cela nous révélera.

À titre illustratif, la version de `moyenne()` proposée à droite fonctionnera pour tous les types d'itérateurs conformes au standard et, bien entendu, pour lesquels les traits appropriés existent.

```
#include <iterator>
template <class It>
   auto moyenne(It debut, It fin) {
      using namespace std;
      using val_t = typename
         iterator_traits<It>::value_type;
      cumul_t<val_t> somme = {};
      for (auto p = debut; p != fin; ++p)
         somme += *p;
      return static_cast<reel_t<val_t>>(somme) /
             distance(debut, fin);
   }
```

Plus forte affirmation encore : toute chose étant égale par ailleurs, le code généré pour `moyenne()` sera le plus efficace possible pour les types impliqués.

Implémenter la fonction `std::distance()`

Pour bien comprendre les techniques impliqués, nous mettrons en place une version simplifiée de la fonction `std::distance()`, que nous nommerons `distance_entre()`. Pour cette section du document, présumez que la fonction `moyenne()` ci-dessus est telle quelle à ceci près qu'elle a recours à `distance_entre()` plutôt qu'à `std::distance()` pour calculer le nombre d'éléments dans la séquence.

Une version naïve et incorrecte (mais rapide) de `distance_entre()` serait celle proposée à droite.

```
// le type de retour est
// typename std::iterator_traits<It>::difference_type
// alors écrivons auto :)
template <class It>
   auto distance_entre(It debut, It fin) {
      return fin - debut;
   }
```

Cette version est incorrecte du fait qu'elle ne fonctionne pas pour toutes les catégories d'itérateurs. En retour, quand elle fonctionne, elle est très rapide.

Il est possible d'écrire une version générale de notre fonction `distance_entre()`, mais il faut pour cela aller au plus lent commun dénominateur, soit déplacer l'itérateur de début d'une position vers la fin jusqu'à ce que la fin soit atteinte en comptant le nombre de déplacements.

Cette version, beaucoup plus lente que la version précédente, s'écrit de la manière proposée à droite.

Ce que nous cherchons à faire ici est donc de faire en sorte que la version la plus efficace possible soit prise pour tout type d'itérateur et que ce choix soit fait à la compilation.

```
template <class It>
   auto distance_entre(It debut, It fin) {
      typename std::iterator_traits<
         It
      >::difference_type n = {};
      for (; debut != fin; ++debut)
         ++n;
      return n;
   }
```

Les catégories d'itérateurs

Pour comprendre comment y arriver, revenons un peu sur la classe de traits `std::iterator_traits`.

Lorsque nous avons examiné ces regroupements de traits un peu plus haut, nous avions pris soin d'escamoter la question de la catégorie d'itérateurs. Au point où nous en sommes maintenant, nous aurons besoin de cette idée pour atteindre nos objectifs.

Le trait `iterator_category` d'un `iterator_traits` est un type descriptif de ce qu'un itérateur donné offre comme capacités; un type servant en quelque sorte de marqueur ou de balise.

Chaque catégorie d'itérateur est un type vide, qui n'a pour utilité que d'exister et de montrer une relation conceptuelle envers d'autres types vides, exprimés par héritage. Ceci permet d'exprimer qu'une catégorie d'itérateurs fait *tout ce qu'une autre catégorie d'itérateurs fait… et même plus!* sans toutefois devoir dire exactement ce que fait une famille d'itérateurs.

Dans la classe `iterator_traits` standard, on peut voir qu'un itérateur capable d'accès direct aura comme le sont les pointeurs aura comme type interne `iterator_category` le type `random_access_iterator_tag`.

```
template<class It>
   struct iterator_traits {
      using iterator_category = typename
         It::iterator_category;
      using value_type = typename
         It::value_type;
      using difference_type = typename
         It::difference_type;
      using pointer = typename It::pointer;
      using reference = typename It::reference;
      // ...
   };
template<class T>
   struct iterator_traits<T *> {
      using iterator_category =
         random_access_iterator_tag;
      using value_type = T;
      using difference_type = ptrdiff_t;
      using pointer = T*;
      using reference = T&;
      // ...
   };
template<class T>
   struct iterator_traits<const T *> {
      using iterator_category =
         random_access_iterator_tag;
      using value_type = T;
      using difference_type = ptrdiff_t;
      using pointer = const T*;
      using reference = const T&;
      // ...
   };
```

D'autres catégories d'itérateurs sont possibles : par exemple, un itérateur ne permettant qu'un parcours unidirectionnel aura comme type interne `iterator_category` le type `forward_iterator_tag`.

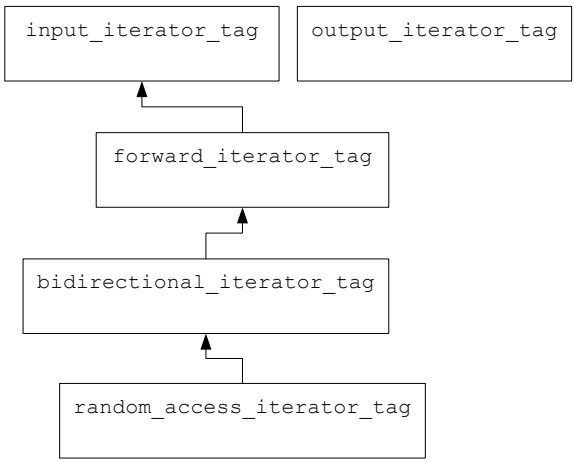
Hiérarchie descriptives disjointes

Les catégories d'itérateurs [StlItTag] nous enseignent une technique très intéressante. En effet, elles montrent qu'il est possible de concevoir une hiérarchie de classes pour décrire les relations opératoires entre types *a priori* disjoints.

Pour décrire des relations opératoires entre des objets et des types primitifs comme des pointeurs, voilà quelque chose de fort utile.

Deux types d'itérateurs n'ont pas nécessairement de relation d'héritage au sens `OO` mais leur relation au sens de la sémantique opératoire peut quand même s'exprimer par une hiérarchie de catégories. C'est une idée subtile et novatrice.

La hiérarchie des catégories standards d'itérateurs pour C++ se présente comme suit. Toutes les relations d'héritage sont publiques et ces cinq classes sont vides (donc aucun attribut, aucune méthode). Le suffixe `_tag` est, par convention, apposé au nom d'une classe de catégorie.



Il faut lire ce schéma de la manière suivante : si un itérateur a comme trait `iterator_category` une catégorie donnée dans cette hiérarchie, alors le code peut s'attendre à ce qu'il expose toutes les fonctionnalités des itérateurs de la même catégorie de même que celles des itérateurs appartenant aux catégories dont sa propre catégorie dérive.

Les catégories `input_iterator` et `output_iterator` servent strictement pour lire ou pour avancer et écrire (pensez aux instances de `istream_iterator` et aux instances de `ostream_iterator` pour des exemples connus).

> Un `output_iterator` est un oiseau étrange. Certains mentionnent, à juste titre [StlItTag], que ces objets ne sont pas vraiment des itérateurs : ils n'ont pas de position dans une séquence (ils sont toujours au point d'écriture, peu importe ce que cela signifie) et ne peuvent donc pas être soumis à une fonction comme `std::distance()`. Cela explique la position à tout le moins unique de cette catégorie dans la hiérarchie des catégories d'itérateurs.

Un `forward_iterator` permet de lire et (s'il n'est pas constant) d'écrire mais de progresser dans une seule direction. Un `bidirectional_iterator` permet de lire et (s'il n'est pas constant) d'écrire et de progresser d'une position vers la fin comme vers le début d'une séquence. C'est la catégorie minimale pour construire un `reverse_iterator`. Enfin, un `random_access_iterator` se comporte comme un pointeur[38].

[38] Cette catégorie permet par exemple d'avancer l'itérateur `itt` de n positions, n étant entier, avec `itt+=n`, et d'utiliser la notation `itt[n]` pour accéder à l'élément n positions devait `itt`.

Spécialiser `distance_entre()` à la compilation

Comment combiner catégories d'itérateurs et algorithmes génériques sur des itérateurs pour sélectionner, dès à la compilation, la version optimale d'un algorithme en fonction des possibilités offertes par un itérateur donné?

Voici comment procéder. Exprimons tout d'abord une version de la fonction `distance_entre()` prenant *trois* paramètres soit le début et la fin d'une séquence et un paramètre anonyme dont le type est celui d'une balise d'itérateur à accès direct. Cette version sera utilisée pour les itérateurs à accès direct;

```
template <class It>
   auto distance_entre(
      It debut, It fin,
      bidirectional_iterator_tag) {
      return fin - debut;
   }
```

Exprimons ensuite une autre version de la fonction `distance_entre()` prenant aussi *trois* paramètres soit le début et la fin d'une séquence et un paramètre anonyme dont le type est celui d'une balise d'itérateur unidirectionnel.

Cette version sera utilisée dans le cas d'itérateurs capables d'avancer mais incapables d'accès directs (réexaminez la hiérarchie des catégories pour comprendre cette manœuvre[39]).

```
template <class It>
   auto distance_entre(
      It debut, It fin,
      forward_iterator_tag) {
      using dist_t = typename
         iterator_traits<It>::difference_type;
      dist_t dist = {};
      for (; debut != fin; ++debut)
         ++dist;
      return dist;
   }
```

Enfin, exprimons la version à *deux* paramètres, destinée à être utilisée par le code client. Son rôle sera de déléguer à l'une des deux fonctions à trois paramètres mentionnées ci-dessus.

La sélection de la bonne version de `distance_entre()` à trois paramètres reposera sur le type du troisième paramètre, et sera donc statique.

```
template <class It>
   auto distance_entre(It debut, It fin) {
      using cat_t = typename
         iterator_traits <
            It
         >::iterator_category;
      return distance_entre(debut, fin, cat_t{});
   }
```

Depuis C++ 17, il est possible de simplifier ceci avec `if constexpr`. Voir ***Des tests réalisés à la compilation – `if constexpr`*** pour plus d'informations à ce sujet.

[39] Quelques remarques s'imposent. Tout d'abord, bien que les types des catégories soient des `struct`, il n'y a pas lieu de les passer par référence puisque les instances de ces `struct` sont vides : le coût de la copie d'une instance d'une catégorie est aussi faible que possible. Si un itérateur de catégorie `bidirectional_iterator_tag` est utilisé avec `distance_entre()`, alors la version pour un itérateur de catégorie `forward_iterator_tag` sera utilisée (ce qui est l'effet souhaité) puisqu'il s'agit d'une spécialisation d'un itérateur unidirectionnel mais pas d'un itérateur à accès direct.

Il s'agit d'une manœuvre de haut calibre :

- dans la version `distance_entre()` à deux paramètres, qui est celle à laquelle nous voulons que le code client ait recours, notre travail se limite à repérer le trait décrivant à catégorie à laquelle appartiennent les itérateurs décrivant la séquence;
- pour alléger la syntaxe, nous en faisons un *alias* local nommé `categ`. Le type `categ` est un type, pas un objet, et est déterminé à la compilation à partir des traits du type `It`;
- la délégation de la version à deux paramètres de `distance_entre()` vers une version à trois paramètres est donc un simple relais, pleinement visible au compilateur et d'une banalité inouïe à optimiser par du *inlining*;
- le troisième paramètre permet au compilateur de savoir laquelle des versions de `distance_entre()` à trois paramètres il devra utiliser, d'où l'instanciation d'un `categ` par défaut anonyme[40];
- le coût de cette instanciation est microscopique, l'objet ainsi créé étant vide (toutes les catégories sont des classes vides). Plusieurs compilateurs le font complètement disparaître du code optimisé (les anglophones diraient *It is Optimized Away*);
- cela dit, la simple présence de ce paramètre vide et anonyme permet de distinguer les deux versions par leur signature, de manière statique, et d'invoquer correctement la meilleure version de la fonction `distance_entre()` considérant les traits du type `It`.

Qu'obtenons-nous donc ici à l'aide des traits et de l'idée de catégorie? Ce que plusieurs nommeront du **polymorphisme statique** : un choix de la meilleure version d'un sous-programme ou d'un type à partir de ses caractéristiques descriptives, le tout sans que cela n'entraîne le moindre coût en temps ou en espace lors de l'exécution du programme.

L'algorithme `moyenne()` revisité

Sachant ce que nous avons maintenant, réécrivons l'algorithme `moyenne()` en utilisant des algorithmes standards dans la mesure du possible. Nous poursuivrons le travail là où nous nous étions arrêtés (le code est proposé à droite).

Notez que `moyenne()`, selon l'acception la plus répandue, s'exprime sous la forme d'un rapport entre la somme des valeurs et le nombre de valeurs. Nous savons déjà que `distance()` nous permet de déduire, de manière optimale, le nombre d'éléments. Voyons maintenant comment utiliser les algorithmes standards pour réaliser un cumul de valeur optimal.

```
#include <iterator>
// ...using...
template <class It>
   auto moyenne(It debut, It fin) {
      using val_t = typename
         iterator_traits<It>::value_type;
      cumul_t<val_t> somme = {};
      for (auto p = debut; p != fin; ++p)
         somme += *p;
      return static_cast<reel_t<val_t>>(somme) /
             distance(debut, fin);
   }
```

[40] L'anonymat de cette variable en facilite l'optimisation par élimination de la part du compilateur.

Il existe en effet un algorithme très utile, `std::accumulate()`, logé dans la bibliothèque standard `<numeric>`, et dont nous pouvons nous servir pour réaliser un calcul optimal d'un cumul de valeurs.

Cet algorithme se décline sous deux saveurs. La plus commune est une version à trois paramètres, soit (dans l'ordre) :

- le début d'une séquence standard;
- la fin de la séquence; et
- la valeur initiale du cumul. Ici, nous utiliserons un zéro du bon type, ce qui convient dans le cas d'une somme.

L'algorithme déduira le type du cumul à partir du type du troisième paramètre, ce qui en allégera la tâche et permettra de dissocier le type des valeurs dans la séquence du type du cumul.

Évidemment, `accumulate()` retournera la valeur du cumul.

```
#include <numeric>
#include <iterator>
// ...using...
template <class It>
   auto moyenne(It debut, It fin, T init) {
      return static_cast<reel_t<val_t>>(
         accumulate(debut, fin, init)
      ) / distance(debut, fin);
   }
template <class It>
   auto moyenne(It debut, It fin) {
      using val_t = typename
         iterator_traits<It>::value_type;
      return moyenne(debut, fin, cumul_t<val_t>{});
   }
```

Cette version de `moyenne()`, en plus d'être compacte, sera optimale. Au mieux, nous pourrions faire aussi bien qu'elle, pas mieux[41].

[41] Un petit bémol : si nous le souhaitions, nous pourrions utiliser la même technique que pour `distance_entre()` et éviter deux parcours dans le cas où la séquence `(debut..fin(` n'est pas basée sur des itérateurs à accès direct, ce qui pourrait être plus rapide mais représenterai plus de travail.

La fonction `accumulate()` est sous-utilisée, car trop peu de gens la connaissent.

Si nous souhaitions calculer le produit des valeurs d'une séquence dans un `double`, par exemple, nous pourrions l'exprimer comme proposé à droite :

```
#include <numeric>
using std::accumulate;
template <class T>
   struct produit {
      template <class U>
         T operator()(const U &a, const U &b) {
            return a * b;
         }
   };
template <class It>
   double produit_sequence(It debut, It fin) {
      return accumulate(
         debut, fin, 1.0, produit<double>{}
      );
   }
```

- un foncteur `produit`, générique sur la base du résultat d'une multiplication, avec un opérateur `()` binaire (à deux opérandes) déduisant à la compilation les types de ses opérandes[42]; et
- une fonction `produit_sequence()`, générique sur la base du type des itérateurs qui lui sont passés, qui invoque `accumulate()` avec une base de `1.0` (un `double`) pour le cumul et une instance de `produit<double>` comme quatrième paramètre.

Lorsque le quatrième paramètre à `accumulate()` est spécifié, ce paramètre doit être une opération à deux opérandes qui sera sollicitée pour le cumul. La version à trois paramètres de l'algorithme `accumulate()` équivaut à une invocation à quatre paramètre qui utiliserait le foncteur `std::plus`, lui aussi dans `<functional>`, comme quatrième paramètre.

[42] Ceci évite d'avoir à identifier le `value_type` des itérateurs dans `produit_sequence()` : le compilateur le fera pour nous en générant le code!

Le foncteur produit ci-dessus aurait d'ailleurs avantageusement pu être remplacé par le foncteur standard `multiplies`, de la bibliothèque `<functional>`, qui fait exactement la même chose.

En ce sens, le programme à droite cumule rapidement le produit des valeurs dans le tableau brut `vals`.

```
#include <numeric>
#include <functional>
#include <algorithm>
#include <iostream>
// ...using...
template <class It>
   double produit_sequence(It debut, It fin) {
      return accumulate(
         debut, fin, 1.0, multiplies<double>{}
      );
   }
int main() {
   int vals[]{ 1, 2, 3, 4, 5 };
   cout << produit_sequence(begin(vals), end(vals));
}
```

Évidemment, aujourd'hui, les λ sont l'outil privilégié pour compléter les algorithmes standards. La version à droite est donc plus idiomatique que les précédentes.

```
template <class It>
   double produit_sequence(It debut, It fin) {
      return accumulate(
         debut, fin, 1.0, [](double so_far, double x) {
            return so_far * x;
         }
      );
   }
int main() {
   int vals[]{ 1, 2, 3, 4, 5 };
   cout << produit_sequence(begin(vals), end(vals));
}
```

Exercices – Série 01

EX00 – Le type `LARGE_INTEGER` de *Win32* sert, sur cette plateforme, à certaines fonctions spécialisées. Définissez un type `GrosEntier` capable d'enrober un `LARGE_INTEGER` avec des opérations arithmétiques et relationnelles et définissez le groupe de traits `TraitsCalcul` pour un `GrosEntier`.

EX01 – Pour concevoir une catégorie d'itérateurs, il importe de mettre en place plusieurs opérateurs, parmi lesquels ont compte le déréférencement (opérateur `*` unaire), l'opérateur ++ préfixé et postfixé, l'affectation (si l'itérateur est non `const`) et l'opérateur `--` (pour les itérateurs bidirectionnels). Il se trouve qu'un itérateur à accès direct (*Random Access*) doit offrir l'opérateur `[]` permettant d'accéder directement au *i*ème élément dans la séquence à partir de la position qu'il y représente. Expliquez pourquoi cet opérateur est important.

EX02 – Rédigez une fonction générique `ecart_type()` prenant en paramètre une séquence standard et retournant l'écart type de cette séquence. Quelles sont les contraintes imposées sur cette séquence (car il y en a; examinez l'équation dans l'encadré à droite)? Pouvez-vous en profiter pour réutiliser votre algorithme `moyenne()` (et, si oui, pouvez-vous le faire efficacement)? Pouvez-vous être pleinement génériques? Pouvez-vous utiliser efficacement l'algorithme `accumulate()`? Expliquez vos choix d'implémentation.

L'écart type d'une séquence s'exprimer mathématiquement sous la forme

$$\sqrt{\frac{\sum(x-\bar{x})^2}{n-1}}$$

où x est une valeur de la séquence, $\bar{x}$ est la moyenne de la séquence, et n est le nombre d'éléments dans la séquence.

Applications des traits

Rencontrer une technique un peu inhabituelle est une chose; l'appliquer en est une autre. Examinons donc une application des traits, question de nous faire une idée sur ce que cette technique ouvre comme possibilités.

Les descriptifs de types numériques

Certaines caractéristiques descriptives d'un type numérique sont utiles sur une base presque universelle. Par exemple :

- le type supporte-t-il les valeurs négatives?
- le type offre-t-il une représentation naturelle de l'infini?
- le type offre-t-il une représentation naturelle de `NaN` (*Not a Number*)?
- le type supporte-t-il l'opération modulo?

Certaines de ces caractéristiques sont propres à tous les types numériques alors que d'autres, comme par exemple la stratégie appliquée pour arrondir une valeur, sont propres aux nombres à virgule flottante.

Le regroupement de traits `std::numeric_limits`, de la bibliothèque `<limits>`, offre ces descriptifs. Un type pour lequel `numeric_limits` est défini devrait permettre qu'on lui applique des algorithmes génériques numériques. Évidemment, ces traits sont définis pour les types numériques primitifs, mais vous pouvez les spécialiser pour vos propres types.

Traits et méthodes

Les traits de `numeric_limits` incluent certaines méthodes. Nous n'avons que très peu exploré de cas où un regroupement de traits offrait des services alors profitons de cette opportunité pour examiner les ramifications de cette stratégie.

Un regroupement de traits n'est pas destiné à être instancié (bien qu'il soit possible de le faire; ce n'est simplement pas très utile dans la majorité des cas). Habituellement, son rôle est d'offrir un soutien descriptif statique (à la compilation seulement) pour la construction d'algorithmes génériques et efficaces. C'est pourquoi, dans la majorité des cas, les méthodes d'un regroupement de traits seront des méthodes de classe plutôt que des méthodes d'instance.

> Cette remarque quant à la faible probabilité de l'utilité de l'instanciation d'un regroupement de traits est une observation, pas un dogme. Il se peut qu'on pense éventuellement à une application d'un regroupement de traits en tant qu'entité instanciée.
>
> Tout en étant conscient(e)s des pratiques en vigueur, gardez l'esprit ouvert.

Dans plusieurs cas, les méthodes d'un regroupement de traits expriment quelque chose qui se rapporte à une constante (par exemple, `numeric_limits<T>::max()` permet d'obtenir la valeur maximale d'un type numérique `T` donné). Le passage à une méthode plutôt qu'à une constante tient du fait que certaines caractéristiques d'un type pourraient devoir être calculées plutôt qu'être connues *a priori*.

Spécialiser `numeric_limits`

Imaginons un type qui est susceptible de se gérer comme un nombre : à tout hasard, une note (au sens de résultat scolaire).

Une version simple et naïve d'un programme lisant et commentant une note légale au Québec (disons entre `0` et `100` inclusivement) pourrait ressembler à l'exemple proposé à droite.

Cet exemple a plusieurs défauts, en particulier le fait qu'il soit peu descriptif : il n'y a pas de concept de note en soi, pas de validation hors du code client, et le tout repose sur des abstractions trop détachées du domaine du problème (des `int`) pour être utiles dans le cas spécifique de la gestion des résultats scolaires.

Nous voudrions, en fait, opérer sur des notes, au sens d'instances d'une classe `Note`, sans perte de généralité ou de performance.

```
template <class T>
   constexpr bool est_entre_inclusif
      (T val, T borne_min, T borne_max) {
      return borne_min <= val && val <= borne_max;
   }
#include <iostream>
// ...using...
int main() {
   const int MIN = 0, MAX = 100;
   int val;
   cout << "Entrez une valeur entre " << MIN
        << " et " << MAX << " inclusivement: ";
   cin >> val;
   if (est_entre_inclusif(val, MIN, MAX))
      cout << "Bravo!" << endl;
   else
      cout << "Pas fort..." << endl;
}
```

Exprimer l'idée de `Note` en tant que `Note` permettra de joindre en un tout cohérent la valeur d'une note et son volet opérationnel (méthodes, validation, encapsulation, garanties, bornes, opérateurs de toute sorte, *etc.*).

Une version correcte et simple d'une classe `Note` est proposée à droite. Cette version exprime son propre lot d'abstractions et contient, entre autres :

- une représentation des bornes minimum et maximum de validité pour une valeur;
- l'encapsulation de la valeur;
- la Sainte-Trinité (implicitement);
- une mécanique d'échange de valeurs;
- quelques opérateurs d'ordonnancement; et
- des opérateurs pour extraire une `Note` d'un flux et pour projeter une `Note` sur un flux.

Vous remarquerez que la méthode de classe `valider()` a recours à un algorithme générique simple, `est_entre_inclusif()` utilisée plus haut. C'est raisonnable et sain de réutiliser du code existant, à plus forte partie s'il est efficace.

Lever une exception à la construction lorsqu'un invariant de l'objet en cours de construction n'est pas respecté est une sage pratique.

```
#ifndef NOTE_H
#define NOTE_H
class Note {
public:
   class Illegal {};
   using value_type = short;
private:
   static const value_type MAX = 100, MIN = 0;
   value_type val = MIN;
   static constexpr value_type valider(value_type val){
      return est_entre_inclusif(val, MIN, MAX)?
         val : throw Illegal{};
   }
public:
   constexpr Note() = default;
   constexpr Note(value_type val) : val{valider(val)} {
   }
   constexpr value_type valeur() const {
      return val;
   }
   constexpr bool operator<(const Note &n) const {
      return valeur() < n.valeur();
   }
   constexpr bool operator<=(const Note &n) const {
      return !(n < *this);
   }
};
#include <iosfwd>
std::ostream& operator<<
   (std::ostream&, const Note&);
std::istream& operator>>(std::istream&, Note&);
#endif
```

Le code de l'opérateur d'extraction d'un flux pour une `Note` est relativement simple. Il repose sur un constructeur paramétrique, un opérateur d'affectation et la capacité de lire une donnée de type `Note::value_type` d'un flux.

L'opérateur de projection sur un flux est, quant à lui, banal.

```
#include "Note.h"
#include <iostream>
// ...using...
ostream& operator<<(ostream &os, const Note &n) {
   return os << n.valeur();
}
istream& operator>>(istream &is, Note &n) {
   if (!is) return is;
   if (Note::value_type val; is >> val)
      n = Note{val};
   return is;
}
```

Le programme de test que nous utiliserons pour le moment s'exprime entièrement en termes d'instances de la classe `Note`.

```
#include "Note.h"
#include <iostream>
int main() {
   using namespace std;
   cout << "Entrez une note légale au Québec: ";
   try {
      if(Note n;cin >> n)
         cout << "Bravo!" << endl;
      else
         cerr << "Pas une note\n";
   } catch (Note::Illegal&) {
      cout << "Pas fort..." << endl;
   }
}
```

Imaginons maintenant que nous souhaitions offrir des services généraux et homogènes pour que des algorithmes applicables à des primitifs puissent aussi s'appliquer à des instances de `Note`.

Par exemple, exprimons une fonction `est_legal()` en termes généraux de respect des bornes de validité pour un type donné et exprimons le programme de test en ces termes. Pour les fins de l'exemple, imaginons que le programme ait pour mission de faire un diagnostic préalable de validité avant de chercher à instancier une note (peu importe la raison). Le type `T` servira de barème pour la plage de valeurs acceptables alors que le type `U` sera une valeur candidate à être exprimée en termes de `T`.

Ici, l'algorithme `est_legal()` repose sur les traits numériques des types sur lesquels il opère. De manière sous-jacente, il utilisera une version à deux types de `est_entre_inclusif()` et les traits des types impliqués.

```
template <class T>
   constexpr bool est_entre_inclusif
      (T val, T borne_min, T borne_max) {
      return borne_min <= val && val <= borne_max;
   }
template <class T, class U>
   constexpr bool est_entre_inclusif
      (U val, T borne_min, T borne_max) {
      return borne_min <= val && val <= borne_max;
   }
#include <limits>
#include <iostream>
// ...using ...
template <class T, class U>
   constexpr bool est_legal(U val) {
      return est_entre_inclusif(
         val, numeric_limits<T>::min(),
              numeric_limits<T>::max()
      );
   }
int main() {
   cout << "Entrez une note légale au Québec: ";
   if (Note::value_type val;
       cin >> val && est_legal<Note>(val))
      cout << "Bravo!" << endl;
   else
      cout << "Pas fort..." << endl;
}
```

Remarquez que pour les types primitifs, dans bien des cas, `est_legal()` est un algorithme banal qui retournera toujours vrai (par définition, tout `short` est un `int` valide, bien que le contraire soit faux). En retour, nous venons de nous doter d'un outil efficace et général pour valider la légalité de valeurs en fonction d'intervalles auxquelles elles sont supposées appartenir.

Si nous souhaitons exprimer `est_legal()` de cette manière, nous devrons donc déterminer le sens à donner à `numeric_limits<Note>`. Remarquez que notre tâche devrait, normalement, être simple : mis à part les règles propres à la gestion des bornes de validité des valeurs possibles pour une `Note`, le reste des traits d'une `Note` devrait s'inspirer des traits du type utilisé pour en représenter la valeur (le type `Note::value_type`)[43].

Nous avons jusqu'ici encapsulé les bornes de validité d'une `Note` à l'intérieur de la classe `Note`; poursuivre dans cette veine est une bonne idée puisque cela réduit le couplage dans les programmes qui utilisent des instances de `Note` (moins le code client voit de noms et de types superflus à son propre travail et mieux ce sera).

[43] Je simplifie ici : si nous voulons que `Note` soit un type numérique à part entière, il faudra mettre en place une gamme plus complète d'opérateurs que les quelques-uns utilisés dans cette petite illustration.

Pour maintenir ce niveau d'encapsulation, nous spécifierons que `numeric_limits<Note>` est ami de `Note`. De même, nous livrerons les traits numériques d'une `Note` *avec* la classe `Note`, dans un seul et même module, comme il se doit.

```
#include <limits>
using std::numeric_limits;
// ...
class Note {
   // ...
   friend class numeric_limits<Note>;
   // ...
public:
   // ...
};
// ...
```

En effet, si une classe `X` est amie d'une autre classe `Y`, au sens `OO`, alors *`X` fait partie de l'interface de `Y`* et les deux devaient aller de pair.

Le reste de la classe `Note` demeure inchangé.

Par la suite, spécialiser `numeric_limits` de manière à ce que les traits qui y sont définis soient applicables à une `Note` est un jeu d'enfant.

En effet, il est possible d'écrire cette classe en entier, mais ce n'est pas nécessaire puisqu'il il suffit d'en faire un dérivé (donc une spécialisation) d'un regroupement de traits correct dans les circonstances, à tout hasard `numeric_limits` appliqué à `Note::value_type`, puis de spécialiser les méthodes de classe qui nous intéressent (`min()` et `max()`) de manière à offrir des bornes conformes aux attentes.

Il est interdit d'ajouter des éléments à l'espace nommé `std` dans le code client, mais il est permis de spécialiser des éléments existants en fonction de nos propres types. Le code à droite est donc légal.

```
#include <limits>
using std::numeric_limits;
// ...la classe Note...
namespace std {
   template <>
      struct numeric_limits<Note>
         : numeric_limits<Note::value_type> {
         static constexpr Note max() {
            return { Note::MAX };
         }
         static constexpr Note min() {
            return { Note:: MIN };
         }
      };
}
```

Notre stratégie de tests de légalité est généralisée pour l'ensemble des types numériques, quelle que soit leur nature. Elle est aussi fortement découplée des types eux-mêmes puisque les traits sont détaillés par une structure descriptive adjointe aux types décrits.

Est-ce du polymorphisme?

Est-ce que la spécialisation de `numeric_limits<T>::max()` pour le type `Note` est une forme de polymorphisme? Techniquement, **non**. En fait, chaque type `T` pour lequel on définit une classe générique `numeric_limits<T>` entraîne la génération d'un nouveau type sans lien hiérarchique avec d'autres types pour lesquels il existe une version de `numeric_limits`. La généricité est un complément statique au polymorphisme mais ne le remplace pas – ce sont deux mécaniques différentes mais permettant un effet semblable. Cela dit, voir plus loin…

Documenter fonctions et foncteurs

Imaginons que nous souhaitions minuter le temps requis pour invoquer une fonction ou un foncteur dans un programme. Il serait agréable d'avoir une seule stratégie qui puisse être applicable dans les deux situations. Pour illustrer notre propos, nous chercherons à enrober l'invocation d'une opération, fonction ou foncteur, sans paramètres.

Nous voudrons éventuellement gérer correctement le cas où l'opération est de type `void` et le cas où son type est différent : si l'opération retourne une valeur, nous voudrons la relayer à l'appelant. Cependant, nous commencerons avec un cas plus simple : notre première approche sera de minuter le temps d'exécution d'une opération (retournant dans ce cas-ci un simple `int`).

La tactique employée sera banale : prendre le temps avant, invoquer l'opération, prendre le temps après, afficher le temps écoulé entre les deux lectures de temps et retourner l'entier résultant de l'opération.

```
#include <chrono>
#include <iostream>
template <class Op>
   auto Minuter(Op oper) -> decltype(oper()) {
      // ...using...
      auto avant = system_clock::now();
      auto resultat = oper();
      auto total = system_clock::now() - avant;
      cout << duration_cast<seconds>(total).count()
           << " sec." << endl;
      return resultat;
   }
```

Vous trouverez à droite un programme montrant comment il serait possible d'utiliser la fonction générique `Minuter()` pour afficher le temps requis pour une invocation de la fonction `f()` tout en récupérant son résultat, et comment il serait possible de faire de même avec le résultat de l'exécution de l'opérateur `()` d'un foncteur de type `X`.

Le comportement générique de `Minuter()` lui permet d'opérer sur la base de la signature de son paramètre `oper` de type `Op`, peu importe le sens à donner au type `Op`.

À la fin de notre démarche, nous souhaiterons conserver la même simplicité d'utilisation que celle propre à ce petit programme de démonstration, mais en faisant en sorte que le type de `Minuter()` s'adapte au type de l'opération à minuter. Nous y arriverons à l'aide de diverses techniques vues jusqu'ici, en particulier celle des traits.

Comme c'est souvent le cas, nous arriverons à maintenir la simplicité à l'utilisation, simplicité du code client, en la transportant du côté serveur.

```
// ...
int f() {
   // ... calculs
   return 3;
}
struct X {
   int operator()() const {
      // ... calculs
      return 7;
   }
};
int main() {
   auto resultat = Minuter(f);
   cout << resultat << endl;
   resultat = Minuter(X{});
   cout << resultat << endl;
}
```

Exercices – Série 02

EX00 – Plutôt que d'afficher le temps écoulé dans `Minuter()`, faites en sorte que ces opérations retournent une paire faite du résultat du calcul et du temps écoulé. Ne considérez pas le cas des opérations `void`.

EX01 – **Difficile** : reprenez le problème posé par `EX00` mais considérez cette fois les opérations `void`. Il vous faudra un peu d'imagination ici.

EX02 – Faites en sorte que le code client invoque une fonction `Minuter()` du même nom peu importe que l'opération à minuter soit `void` ou non. Évidemment, la version `void` ne retournera rien, ce qui influencera la forme que prendra son invocation. Quelles sont les options qui s'offrent à vous? Justifiez vos choix d'implémentation.

EX03 – Les opérations de minuterie utilisées ici manquent un peu de flexibilité. Elle affichent toujours sur la sortie standard (avec `std::cout`) et utilisent toutes la fonction `std::clock()` de la bibliothèque standard pour relever le temps écoulé. Cette fonction, traditionnellement, retourne une valeur précise à peu près au millième de seconde, ce qui est généralement inférieur aux mécanismes propres à la plateforme. Repensez la stratégie de minuterie pour qu'elle puisse tenir compte d'un service de saisie de temps choisi par le code client, tout en utilisant `std::clock()` par défaut. *Ce travail, sans être très difficile, demande plus de réflexion qu'il n'y paraît sur le plan du design*.

Les *templates* variadiques

L'un des ajouts conceptuels importants à C++ depuis l'avènement de C++ 11 est la possibilité d'exprimer, directement, à l'aide de mécanismes du langage, des types et des algorithmes génériques sur la base d'un nombre arbitrairement grand de paramètres.

Il faut comprendre tout d'abord que l'expression de tels types et algorithmes est possible depuis longtemps, par exemple grâce aux listes de types mises de l'avant par **Andrei Alexandrescu**. Cependant, les *templates* variadiques, mécanisme par lequel s'exprime cette nouvelle facilité du langage, simplifient et structurent cette pratique pour en faire un citoyen de première classe du langage.

Soupçon d'histoire – les fonctions variadiques de C

Le terme fonction variadique est utilisé en langage C pour décrire une fonction prenant un nombre arbitrairement grand de paramètres. Les plus connues de ces fonctions sont celles des familles `printf()` et `scanf()`, que nous examinerons brièvement ici.

Le code à droite donne un exemple d'utilisation des fonctions `printf()` et `scanf()`. Vous constaterez que ces fonctions sont flexibles, au sens où :

```
#include <cstdio>
int main() {
   using namespace std;
   enum { TAILLE_MAX_NOM = 64 };
   int age;
   char nom[TAILLE_MAX_NOM];
   printf("Entrez un nom (sans blancs) suivi "
          "d'un blanc et d'un age (entier): ");
   scanf("%s %d", nom, &age);
   printf("Vous avez entré: \"%s\" d'âge %d",
          nom, age);
}
```

- elles prennent en paramètre une chaîne de caractères (un `const char*`) décrivant des règles de formatage, indiquées par des marqueurs tels que `%s` (chaîne de caractères) ou `%d` (entier); et
- elles prennent par la suite un nombre de paramètres aussi grand que le souhaite le code client, et opère sur ceux-ci sur la base des règles décrites par la chaîne de formatage.

Le hic est que, bien que flexible, cette approche est extrêmement dangereuse. Ce code, par exemple, n'empêche pas un usager pervers (ou simplement inconscient) d'entrer un nom plus grand que `TAILLE_MAX_NOM` caractères, causant un débordement de capacité sur la variable nom (question de formatage insuffisant dans ce cas-ci). Pire encore, c'est à l'exécution que la mise en correspondance de la chaîne de formatage et des paramètres subséquents est réalisée, ce qui fait qu'un programmeur pourrait ne pas utiliser le bon nombre de paramètres pour une chaîne de formatage donnée, ce qui peut constituer un sérieux bris de sécurité (surtout avec les fonctions de la famille `scanf()`).

Certains compilateurs accordent un traitement spécial à ces fonctions (pas toutes les fonctions variadiques, évidemment, mais celles qui sont connues et fortement usitées), et font une validation au préalable sur la correspondance apparente entre la chaîne de formatage et les paramètres supplémentaires. Il faut comprendre ici que cette pratique se veut utile et conviviale, mais dépend pleinement de l'outil; rien ne garantit un tel support *a priori*, et ces fonctions demeurent fondamentalement dangereuses.

Le langage C a un système de types simple; les fonctions variadiques n'y supportent que les types primitifs, mais cela n'y pose pas vraiment problème puisqu'il n'y a pas autre chose que des primitifs (incluant pointeurs sur des données et pointeurs sur des fonctions), des tableaux et des enregistrements dans ce langage.

La plupart des problèmes associés aux fonctions variadiques de C disparaissent avec C++, qui utilise une métaphore à deux opérandes :

- chaque écriture prend un flux, un opérande et retourne une référence sur le flux suite à l'écriture, ce qui permet d'enchaîner les écritures en aussi grand nombre que souhaité;
- la mécanique est la même pour la lecture;
- un flux est testable comme un booléen, ce qui permet de tester le succès ou l'échec d'une opération. Le code à droite montre comment valider une lecture à l'aide d'une alternative (un if), même s'il ne traite pas le cas d'une erreur de lecture;
- enfin, la possibilité d'utiliser des objets tels qu'une `std::string` plutôt que des types primitifs comme des tableaux de char permet d'écrire du code évitant tout débordement de capacité.

```
#include <iostream>
#include <string>
int main() {
   using namespace std;
   int age;
   string nom;
   cout << "Entrez un nom (sans blancs) suivi d'un blanc et d'un age (entier): ";
   if (cin >> nom >> age)
      cout << "Vous avez entré: \"" nom << "\" d'âge " << age;
}
```

Les fonctions variadiques de C ne sont pas une solution généralement applicable en C++, pour les raisons indiquées ci-dessus. Cependant, nombreuses sont celles et nombreux sont qui aiment cette manière d'exprimer des entrées et – surtout – des sorties sous cette forme.

Java a pris une approche simple mais lente en appliquant un traitement particulier au type `String` et en faisant en sorte que tout type soit convertible en `String` par une méthode `toString()` souvent implicite.

Sans surprises, chaque conversion en `String` prend du temps, mais la métaphore a le mérite d'être simple et accessible :

```
int âge;
String nom;
// ... lecture de l'âge et du nom...
// omis pour fins de simplicité...
System.out.println("Vous avez entré \"" + nom + "\" d'âge " + âge);
```

Les langages `.NET`, par exemple C#, utilisent une métaphore très proche de celle des fonctions du langage C, mais l'expriment en transformant les paramètres supplémentaires en un tableau de paramètres, susceptible d'être traité itérativement par la fonction appelée.

Nous avons ici un cas clair où, même si les langages `.NET` sont des outils contemporains, la métaphore préconisée par C, avec ses risques et ses tares, demeure en vogue et appréciée de plusieurs.

```
int âge;
string nom;
// ... lecture de l'âge et du nom...
// omis pour fins de simplicité...
Console.WriteLine("Vous avez entré \"{0}\" d'âge {1}", nom, âge);
```

Peut-on envisager une métaphore semblable à celle-ci, visiblement populaire, tout en demeurant sécuritaire et en évitant les coûts rencontrés en Java ou dans les langages `.NET`?

Exemple simple – remplacer l'infâme `std::printf()`

Les *templates* variadiques permettent d'implémenter un mécanisme pour réaliser l'impression d'autant de paramètres que souhaité dans un seul et même appel de fonction. Pour les besoins de l'exemple, nous ne ferons pas de formatage particulier, mais il ne serait pas difficile d'enrichir notre approche pour tenir compte de choses telles que représenter un nombre en format octal ou hexadécimal par exemple.

Voici une implémentation opérationnelle, sous laquelle j'utilise les flux d'entrée/ sortie standards de C++. Cet exemple a été écrit de manière à mettre en relief quelques caractéristiques des templates variadiques. Des explications suivent.

```
#include <iostream>
#include <string>
using namespace std;
template <class T>
   void print_(T &&arg) {
      cout << arg << ' ';
   }
template <class T, class ... Args>
   void print_(T && val, Args &&... args) {
      print_(std::forward<T>(val));
      print_(std::forward<Args>(args)...);
   }
template <class ... Args>
   void print(Args &&... args) {
      cout << "Inmpression de " << sizeof...(Args) << " parametres: ";
      print_(std::forward<Args>(args)...);
   }
int main() {
   string s = "Yo";
   print("J'aime mon prof!", 3, s);
}
```

J'ai fait le choix délibéré ici d'insérer un blanc entre chaque élément affiché, pour les fins de cet exemple de démonstration, mais en pratique ce n'est pas ce que je privilégierais; il ne faut pas chercher à imposer de tels choix au code client, à mon avis.

Examinons les éléments clés, prenant le programme de bas en haut, du concret vers l'abstrait.

Programme principal – appel à `print()`

Le programme principal met en relief l'appel à la fonction `print()` recevant trois paramètres, soit un `const char*`, un `int` et une `std::string`.

Ce qu'il faut noter dans cet exemple est que nous n'avons pas écrit une fonction `print()` à trois paramètres; nous avons plutôt écrit une fonction `print()` générale, acceptant autant de paramètres que souhaité.

Fonction générale `print(Args && ...)`

Cette fonction a été écrite pour mettre en relief :

- la capacité de relayer des paramètres sans perte d'information de type (voir ***Relais parfait (Perfect Forwarding)***), sur lequel nous reviendrons un peu plus bas, et
- l'opérateur `sizeof...`, ci-dessous.

Remarquez la syntaxe pour les paramètres variadiques :

- la signature dans la déclaration des paramètres du *template* est `class ... Args` (les espaces sont optionnels), plaçant l'ellipse entre `class` (ou `typename`, au choix) et le nom du paramètre du *template*, soit `Args` dans ce cas-ci (le nom `Args` ici est un choix personnel, pas une obligation syntaxique);
- dans la signature de la fonction, l'écriture est `Args ... args`, donc l'ellipse est alors placée entre le nom du « type » et le nom de la « variable ». J'ai placé les mots « type » et « variable » entre guillemets français du fait que ce sont en fait des marqueurs génériques représentant une liste de types et la liste de variables correspondante. Le compilateur, bien sûr, connaît le nombre de paramètres, le type de chaque paramètre, et est en mesure de donner des noms à chacun de ces paramètres, le tout à l'insu des programmeuses et des programmeurs;
- enfin, à l'utilisation, deux écritures sont possibles, soit :
  - `f(args...)` pour une fonction `f` prenant les paramètres représentés par `args` (dans l'ordre), ce qui permet à une fonction variadique de déléguer un bloc de `n` paramètres à une autre fonction prenant exactement les mêmes paramètres; et
  - `f(args)...` pour appliquer la fonction `f` à tous les paramètres, un à la fois. C'est cette écriture que nous utilisons ici avec l'appel de fonction `std::forward<Args>(args)...`, car cette fonction ne prend qu'un seul paramètre et a pour rôle d'assurer leur relais sans perte de sémantique du point de vue des types

Décrite en mots, nous pourrions dire que `print(args...)` affiche le nombre de paramètres dans `args...`, et appelle `print_(args...)` en s'assurant que chaque paramètre passé à `print()` soit par la suite passé à `print_()`, dans le même ordre et sans perte d'information quant aux types.

Opérateur `sizeof...(Args)`

Alors que l'opérateur statique `sizeof(T)` s'applique à un type `T` (ou à une instance d'un type `T`) permet de connaître dès la compilation le nombre de *bytes* qu'occupe son paramètre en mémoire, l'opérateur `sizeof...(Args)` s'applique à un groupe de paramètres et permet de connaître dès la compilation le nombre d'éléments du groupe de paramètres `Args`.

Fonction `print_(T&&, Args&&...)`

Pour traiter correctement chaque paramètre d'un groupe de paramètres, la manière la plus simple est de les prendre un par un, comme si nous traitions une liste dans un langage fonctionnel : traiter l'élément en tête de liste, puis traiter le reste des éléments (qui constituent eux-mêmes une liste) de manière récursive.

Les *templates* variadiques se distinguent des listes traditionnelles des langages fonctionnels au sens où ici, la récursivité se fait par la génération, dès la compilation, de fonctions spécialisées pour traiter les types impliqués lors de l'appel. Ainsi :

- appeler `print_(int,float,const char*)` génère une fonction `print_(int,Args...)` qui appellera `print_(int)` et `print_(Args...)`, donc `print_(float,const char*);`
- cet appel à `print_(float,const char*)` se décomposera en deux appels, soit l'un à `print_(float)` et l'autre à `print_(const char*);`
- puisque les fonctions générées sont déterminées à la compilation, le compilateur écrit en quelque sorte pour nous les fonctions dont nous avons besoin, plutôt que d'appeler une même fonction de manière récursive;
- enfin, puisque ces fonctions sont connues du compilateur, ce dernier est en mesure de réaliser des optimisations, par exemple du *function inlining*, ce qui a pour conséquence de rendre les divers appels de fonctions à essentiellement gratuits

*Relais parfait (*Perfect Forwarding*)*

Vous avez probablement remarqué, au passage, des appels à une fonction `std::forward()`. Qu'est-ce donc que cet étrange animal?

Deux saveurs de `&&`

Le langage distingue le cas où un type suffixé par `&&` est fixé par le programme, dans quel cas il s'agit d'une référence sur un *rvalue* (passage par mouvement) et celui où il est découvert par le compilateur (passage par relais, *Forwarding References*).

Ainsi :

Référence sur des *rvalues*	Références de relais

`int && r = 3;`
`template <class T>`
`  void f(vector<T> &&v); // v par mouvement`
`template <class T>`
`struct X {`
`  X(X&&); // X<T> passé par mouvement`
`  X(T&&); // T passé par mouvement`
`};`
	`auto && r = 3;`
	`template <class T>`
	`  void f(T &&v);`
	`template <class T>`
	`  struct X {`
	`  template <class U>`
	`  X(U&&);`
	`  };`

Dans le cas d'un passage par relais, le compilateur fait de son mieux pour respecter la sémantique du paramètre au point d'appel. En indiquant `T&&`, il passera `T` par référence ou par mouvement (en fonction de la nature du paramètre au point d'appel), et en indiquant `Args&&...`, il passera chaque paramètre de `Args...` selon sa sémantique au point d'appel.

Les paramètres en mouvement sont typiquement anonymes au point d'appel; en les accueillant dans une fonction et en les nommant, ils perdent cette qualité. La fonction `std::forward()` intervient pour préserver la sémantique identifiée au point d'appel.

Utilité du Perfect Forwarding

Les appels à la fonction `std::forward()` dans le code ne sont pas décoratifs. À titre d'exemple, supposons la fonction `print_size()` imprimant la taille des types des paramètres plutôt que leur valeur, et supposons-là écrite comme suit :

```
template <class T>
   void print_size(T &&) {
      cout << sizeof(T) << ' ';
   }
template <class T, class ... Args>
   void print_size(T && val, Args && ... args) {
      print_size(val);
      print_size(args...);
   }
```

Si nous appelons cette fonction comme suit :

```
print_size("J'aime mon prof", 3, 3.14159);
```

... alors nous verrons probablement s'afficher quelque chose comme `16 4 8`. Par contre, si nous appelons cette fonction comme suit :

```
print_size(3, "J'aime mon prof", 3.14159);
```

... alors nous verrons probablement s'afficher quelque chose comme `4 4 8`.

La différence tient à la position de la chaîne de caractères dans la séquence, et à un phénomène de décrépitude de pointeurs : lorsque la chaîne est placée au début de la séquence de paramètres, le compilateur la perçoit comme un `const char(&)[16]` (donc la taille en *bytes* est considérée `16`) alors que lorsque ce tableau est passé par copie à une fonction, la sémantique associée au pointeur se perd et le compilateur ne voit plus qu'un `const char*` (donc pour le compilateur, la taille devient celle d'un pointeur).

Cette perte de sémantique est évitée par le recours à `std::forward()`, qui a pour rôle de réaliser le *Perfect Forwarding*, donc de convertir un type de manière à en éviter les pertes de sémantique. Écrire ceci :

```
template <class T>
   void print_size(T &&) {
      cout << sizeof(T) << ' ';
   }
template <class T, class ... Args>
   void print_size(T && val, Args && ... args) {
      print_size(forward<T>(val));
      print_size(forward<Args>(args)...);
   }
```

... a pour conséquence que les deux appels ci-dessous afficheront respectivement `16 4 8` et `4 16 8`, donc que nous éviterons les pertes de sens.

```
print_size("J'aime mon prof", 3, 3.14159);
```

```
print_size(3, "J'aime mon prof", 3.14159);
```

Fonction `print_(T&&)`

Enfin, la version de `print_()` à un seul paramètre fait l'affichage à proprement dit.

Comme pour toute fonction du genre, il est possible de la spécialiser pour différents types, tout en gardant la version générique proposée ici comme cas général. Ceci explique que `print_(T,Args...)` appelle successivement `print_(T)` et `print_(Args...)` plutôt que d'afficher le `T` directement : en passant par la fonction à un seul paramètre dans tous les cas, la spécialisation par type devient possible, et nous gagnons en flexibilité sans pertes du point de vue de la vitesse d'exécution.

Définir un petit conteneur maison

Pour mieux comprendre comment sont faits les conteneurs standards et les itérateurs[44], il peut être utile de définir notre propre (petit) conteneur maison.

Pour nos fins pédagogiques, nous définirons une classe `liste_simple`[45] qui représentera une liste simplement chaînée. Cette classe sera générique et n'offrira que quelques services minimalistes, soit `push_back()`, `size()`, `begin()`, `end()`, l'affectation, `clear()`, `swap()`, un destructeur et quelques constructeurs.

Un programme de test simple pour cette classe pourrait être celui proposé à droite.

On remarquera tout de suite quelques éléments clés pour un conteneur qui se veut à la fois conforme au standard et très utilisable :

```
// ... inclusions requises pour notre
// liste_simple (omises pour le moment)
#include <iostream>
#include <algorithm>
#include <iterator>
int main() {
   // ...using...
   int tab []{ 1, 2, 3, 4, 5 };
   liste_simple<int> lst{begin(tab), end(tab)};
   for(auto & val : lst)
      cout << val << ' ';
   liste_simple<short> lst2{lst};
   cout << endl << "Taille de lst2: "
        << lst2.size() << endl;
}
```

- construire un conteneur à partir d'une séquence existante est une opération naturelle. Ici, `lst` est construite à partir des éléments de `tab`. Nous voudrons donc que `liste_simple` expose au moins un constructeur de séquence;
- construire un conteneur par copie est aussi une opération naturelle, même si les éléments des conteneurs ne sont pas du même type (il en va de même pour l'affectation, même si cette opération n'est pas illustrée ici). Nous voudrons donc que `liste_simple` expose au moins un constructeur de type apparenté;
- un itérateur maison doit se comporter comme un itérateur standard de la même catégorie pour faciliter son utilisation à l'aide d'algorithmes standards; enfin
- il est sous-entendu que chaque `liste_simple` sera responsable de ses propres données. La mémoire vive ne doit pas fuir dans un programme utilisant des instances de `liste_simple` comme celui proposé ici.

[44] Nous éviterons la question des allocateurs pour le moment, mais sentez-vous libres d'examiner la section ***Gestion avancée de la mémoire***, plus loin, si vous avez envie de compléter notre conteneur de manière à ce qu'il supporte aussi des allocateurs déterminés par le code client.

[45] Cette section a été écrite avant l'insertion, dans C++ 11, du type `std::forward_list`, alors ne trichez pas!

Types internes et publics

Un conteneur standard doit, pour faciliter l'opération de certains algorithmes, par exemple l'algorithme `std::copy()` dont se servira notre programme de test, définir certains types internes et publics.

Les plus utiles en pratique sont ceux proposés à droite : `value_type` pour le type des éléments du conteneur, `reference` pour le type d'une référence sur un élément, `pointer` pour le type d'un pointeur sur un élément, `const_reference` pour le type d'une référence constante sur un élément de même que `size_type` pour le type sur lequel est représenté le nombre d'éléments du conteneur.

```
// dans ce qui suit, les using sont
// omis pour alléger le code
#include <iterator>
#include <utility>
template <class T>
   class liste_simple {
   public:
      using value_type = T;
      using reference = T&;
      using pointer = T*;
      using const_reference = const T&;
      using size_type = std::size_t;
```

Représentation interne

Notre conteneur sera une liste simplement chaînée inefficace (mais qui peut être *fortement* améliorée) et ses éléments seront insérés dans des instances de la classe `Noeud`.

Un `Noeud` sera une classe interne et privée à `liste_simple`, qui exposera une valeur et un successeur, représenté par un pointeur de `Noeud`. J'ai utilisé un type aux attributs publics par souci de simplicité, mais on pourrait évidemment faire bien mieux.

Ne confondez pas `Noeud` et itérateur dans la liste : le `Noeud` est un outil *organisationnel* interne alors que l'itérateur sera un outil *de parcours* et une interface d'accès aux éléments du conteneur.

```
   private:
      struct Noeud {
         using value_type = T;
         value_type val {};
         Noeud *succ = nullptr;
         Noeud() = default;
         Noeud(const value_type &val) : val{val} {
         }
         template <class U>
            Noeud(const U &val) : val{val} {
            }
         Noeud& operator=(const Noeud &n) {
            val = n.val;
            return *this;
         };
         template <class U>
            Noeud& operator=(const Noeud<U> &n) {
               val = n.val;
               return *this;
            }
         ~Noeud() = default;
      };
```

Un `Noeud` n'est pas responsable de la gestion de la vie de son successeur. Cette tâche revient au conteneur. Une erreur trop souvent commise est de confier trop de responsabilités à une unité primitive telle que `Nœud`; ne tombez pas dans le piège!

Remarquez que l'affectation d'un `Noeud` à un autre réalise une copie de valeurs, pas une copie de successeurs. La classe `Noeud` se limite à son travail, sans plus.

Une instance de `liste_simple` connaîtra sa tête (un pointeur sur son premier élément) et le nombre d'éléments qu'elle contient.

Pour trouver le dernier élément de la liste, une méthode privée de complexité $O(n)$ sera implémentée. C'est une stratégie *très* inefficace, mais ça fonctionne.

```
Noeud *tete = nullptr;
size_type nelems {};
// Complexité O(n)
Noeud *trouver_dernier() const noexcept {
   Noeud *q = {};
   for (auto p = tete; p; p = p->succ)
      q = p;
   return q;
}
```

Définir un itérateur

Un itérateur pour une instance de `liste_simple` sera un objet conscient de la structure interne de la liste, et permettant de la parcourir et d'accéder de manière naturelle et homogène à ses éléments.

L'itérateur étant intégré à la liste, il connaîtra sa structure interne et saura donc ce qu'est un `Noeud`. En retour, son rôle sera de donner accès aux valeurs dans chaque `Noeud`, et non pas au `Noeud` lui-même.

Notre itérateur sera un itérateur de catégorie *Forward Only*, ce qui signifie qu'il ne permettra que d'avancer dans la séquence, pas de reculer. Pour une liste chaînée, c'est sûrement raisonnable.

Depuis C++ 17, `std::iterator` est déprécié, et il est désormais recommandé de définir les traits d'itérateurs un à un…

Son opérateur `++` dans chacune des déclinaisons habituelles permettra de passer au prochain élément de la séquence et sera exprimé dans le respect des idiomes du langage.

```
class iterator
   : public std::iterator <
        std::forward_iterator_tag, value_type
     >
{
   Noeud *cur = nullptr;
public:
   iterator() = default;
   iterator(Noeud *p) noexcept : cur{p} {
   }
   // la Sainte-Trinité s'applique

   iterator& operator++() noexcept {
      cur = cur->succ;
      return *this;
   }
   iterator operator++(int) noexcept {
      aurt temp{*this};
      operator++();
      return temp;
   }
```

```
        bool operator==
            (const iterator &it) const noexcept {
            return cur == it.cur;
        }
        bool operator!=
            (const iterator &it) const noexcept {
            return !(*this == it);
        }
        value_type& operator*() noexcept {
            return cur->val;
        }
        const value_type& operator*() const noexcept {
            return cur->val;
        }
        pointer operator->() noexcept {
            return &(cur->val);
        }
        const pointer operator->() const noexcept {
            return &(cur->val);
        }
    };
```

Les comparaisons se feront sur les instances de `Noeud` plutôt que sur les valeurs puisqu'un itérateur est d'abord et avant tout un lieu logique, comme le sont les pointeurs.

Les indirections, passant par les opérateurs `*` et `->`, agiront quant à elles sur les valeurs des diverses instances de `Noeud` plutôt que sur les instances de `Noeud` elles-mêmes.

```
    public:
        using const_iterator = const iterator;
        iterator begin() noexcept {
            return tete;
        }
        const_iterator begin() const noexcept {
            return tete;
        }
        iterator end() noexcept {
            return nullptr;
        }
        const_iterator end() const noexcept {
            return nullptr;
        }
```

Toute instance de `liste_simple` exposera les types internes et publics `const_iterator` et `iterator`, de même que les habituelles méthodes `begin()` et `end()`, qui exploiteront le constructeur paramétrique de l'itérateur pour faciliter la mise en place d'une logique simple : un itérateur sur un pointeur de `Noeud` nul représentera conceptuellement une fin de séquence, et pointera au même endroit que le dernier élément d'une liste-simple donnée.

Ce choix un peu simpliste d'implémentation poserait problème si nous souhaitions mettre en place un itérateur bidirectionnel et implémenter un `reverse_iterator`, capable de parcourir la liste à rebours, puisqu'il est plus simple de terminer une séquence avec un pointeur nul que de débuter une séquence avec un tel pointeur (après tout, qu'y aurait-il, conceptuellement, à la suite de l'adresse `0`?). Si vous souhaitez extrapoler à partir de cette proposition de conteneur pour en arriver à une solution bidirectionnelle, envisagez terminer la séquence non pas par un pointeur nul mais bien par un nœud bidon, représentant la fin de séquence.

Les constructeurs seront très simples et reposeront, autant que faire se peut, sur des algorithmes standards.

Une `liste_simple` vide aura une tête nulle et n'aura, conséquemment, aucun élément. Les copies d'éléments reposeront sur la méthode `push_back()`, elle-même invoquée par l'instance de `std::back_inserter`, qui réalisera (nous y reviendrons) une insertion à partir de la valeur de l'original, implémentant *de facto* une sémantique de copie de valeurs plutôt qu'une copie de pointeurs.

La similitude entre le constructeur de copie, le constructeur de séquence et le constructeur de type apparenté n'est pas accidentelle.

Réflexion 03.0 : ce code est-il sécuritaire face aux exceptions? Si oui, pourquoi? Sinon, quels ajustements devraient y être apportés? Réponse dans ***Réflexion 03.0 : construction sécuritaire?***.

```
template <class It>
   void ajouter_fin(It debut, It fin) {
      copy(debut, fin, back_inserter(*this));
   }
liste_simple() = default;
liste_simple(const liste_simple &lst)
   : liste_simple() {
   ajouter_fin(lst.begin(), lst.end());
}
template <class U>
   liste_simple(const liste_simple<U> &lst)
      : liste_simple() {
      ajouter_fin(lst.begin(), lst.end());
   }
template <class It>
   liste_simple(It debut, It fin)
      : liste_simple(){
      ajouter_fin(debut, fin);
   }
```

Certaines méthodes sont très simples à implémenter. On pense entre autres au destructeur (qui doit vider la liste), de complexité $O(n)$; à la méthode `empty()`, qui doit indiquer si la liste est vide (complexité $O(1)$), et à la méthode `size()`, de complexité $O(1)$.

La méthode `clear()`, dont le rôle est de vider le conteneur, s'exprime simplement : tant qu'il restera au moins un élément, détruire cet élément.

```
      ~liste_simple(){
         clear();
      }
      size_type size() const noexcept {
         return nelems;
      }
      bool empty() const noexcept {
         return !tete;
      }
      void clear() noexcept {
         while (!empty()) {
            auto p = tete->succ;
            delete tete;
            tete = p;
         }
         nelems = 0;
      }
```

La méthode `swap()` sera de complexité $O(1)$ et offrira la garantie `noexcept`, comme il se doit pour cette méthode fondamentale.

```
   void swap(liste_simple &lst) noexcept {
      using std::swap;
      swap(tete, lst.tete);
      swap(nelems, lst.nelems);
   }
```

L'affectation sous sa déclinaison usuelle et sous sa déclinaison de type apparenté s'exprimera en toute simplicité et en temps linéaire (car la copie et la conversion, respectivement, devront copier les éléments un à un) grâce à l'idiome d'affectation sécuritaire.

```
liste_simple& operator=(const liste_simple &lst) {
   liste_simple{lst}.swap(*this);
   return *this;
}
template <class U>
   liste_simple& operator=(const liste_simple<U> &lst) {
      liste_simple{lst}.swap(*this);
      return *this;
   }
```

La méthode `push_back()` sera de complexité linéaire à cause des choix d'implémentation pour le moins perfectibles qui ont été faits ici.

Enfin, comme pour toute classe dans laquelle l'opération `swap()` aura été raffinée, nous exposerons une spécialisation optimisée de l'algorithme standard `std::swap()` pour notre type `liste_simple`.

```
      void push_back(const value_type &val) {
         auto q = new Noeud{val};
         if (auto p = trouver_dernier(); p)
            p->succ = q;
         else
            tete = q;
         ++nelems;
      }
   };
namespace std {
   template <class T>
      void swap(liste_simple<T> &a,
                liste_simple<T> &b) noexcept {
         s.swap(b);
      }
}
```

Exercices – Série 03

EX00 – Implémentez la méthode `push_front()` de `liste_simple` qui insère un élément au début de la séquence et validez qu'elle fonctionne en tout temps. Documentez la complexité algorithmique de cette méthode.

EX01 – Implémentez la méthode `insert()` de `liste_simple` qui insérera un élément tout juste après l'élément pointé par un itérateur valide de la liste, et validez qu'elle fonctionne en tout temps (incluant l'insertion en début de liste, en fin de liste, en un point arbitraire d'une liste et dans une liste vide). Documentez la complexité algorithmique de cette méthode.

EX02 – Implémentez la méthode `insert()` de `liste_simple` qui insère une séquence d'éléments tout juste après l'élément pointé par un itérateur valide de la liste, et validez qu'elle fonctionne en tout temps (incluant encore une fois l'insertion en début de liste, en fin de liste, en un point arbitraire d'une liste et dans une liste vide). Documentez la complexité algorithmique de cette méthode.

EX03 – Implémentez la méthode `erase()` de `liste_simple` qui supprimera l'élément pointé par un itérateur valide de la liste, et validez qu'elle fonctionne en tout temps (incluant la suppression en début de liste, en fin de liste et en un point arbitraire d'une liste). Quel devrait être son comportement si la liste est vide? Documentez la complexité algorithmique de cette méthode.

EX04 – Implémentez la méthode `erase()` de `liste_simple` qui supprimera une séquence d'éléments déterminée par paire valide d'itérateurs dans la liste, et validez qu'elle fonctionne en tout temps (incluant la suppression en début de liste, en fin de liste et en un point arbitraire d'une liste). Quel devrait être son comportement si la liste est vide? Documentez la complexité algorithmique de cette méthode.

EX05 – Repensez l'implémentation de `liste_simple` sans changer son visage public pour faire en sorte que ses opérations primitives soient plus efficaces. Êtes-vous en mesure de procéder à ce raffinement à coût faible (ou nul) en espace mémoire?

EX06 – Implémentez le constructeur de mouvement et l'affectation de mouvement pour `liste_simple`.

Itérer sur un intervalle de valeurs

Il existe des algorithmes standards pour un tas d'opérations récurrentes en programmation, peu importe le volet de spécialisation des informaticien(ne)s. Par exemple, dans l'espace nommé `std`, il existe un algorithme rodé et efficace pour :

- appliquer une opération aux éléments d'une séquence (`for_each()`);
- faire d'une séquence une version modifiée d'une autre (`transform()`);
- initialiser les éléments d'une séquence (`generate()` et `generate_n()`);
- chercher un élément dans une séquence (`find()`,`find_if()`, `binary_search()`, …); et j'en passe.

Il existe aussi diverses versions du concept de séquence, concept représenté de manière abstraite mais opérationnelle par la théorie des itérateurs sur des conteneurs. Les concepts d'algorithme, d'itérateur et de conteneur sont à la fois puissants et polyvalents.

En mêlant conteneurs, itérateurs et foncteurs, il est possible d'exprimer un algorithme projetant les entiers de `0` à `99` sur un flux en sortie de la manière proposée à droite.

L'algorithme standard `iota()` initialise une séquence à l'aide de valeurs séquentielles obtenues à partir d'une valeur initiale (un peu plus simple qu'un appel à `generate()`), alors que `copy()` projette une séquence de valeurs sur une séquence en sortie (ici, le flux associé à la console).

```
#include <iterator>
#include <algorithm>
#include <numeric>
#include <iostream>
int main() {
   // ...using...
   const int N = 100;
   int tab[N];
   iota(begin(tab), end(tab), int{});
   copy(begin(tab), end(tab),
        ostream_iterator<int>{cout, " "});
}
```

Un irritant : ce programme consomme beaucoup de mémoire pour rien, le tableau étant au fond un peu superflu. Une répétitive à compteur ferait le même travail à l'aide d'un seul entier plutôt qu'à partir d'un tableau d'entiers (et le ferait en une seule itération plutôt que deux).

Il serait donc utile de pouvoir exprimer le concept d'*itérer à travers une séquence de valeurs* sans nécessairement devoir remplir un conteneur avec les valeurs puis itérer à travers ce conteneur. Cela économiserait à la fois temps (une itération, pas deux) et espace (pas besoin de tableau).

Définir un itérateur de valeurs

Notre souhait est de pouvoir exprimer un programme opérant sur une séquence de valeurs (comme le programme affichant les entiers de `0` à `99` inclusivement ci-dessus) mais de manière telle que la séquence de valeurs en question soit déterminée strictement par des itérateurs. Un bon exemple serait le programme suivant :

```
#include "value_iterator.h"
#include <iostream>
#include <iterator>
#include <algorithm>
//
// ...vos propres inclusions...
//
int main() {
   // ...using...
   const int N = 100;
   copy(value_iterator<int>{0}, value_iterator<int>{N},
         ostream_iterator<int>{cout, " "});
}
```

On remarque tout de suite la disparition du tableau de valeurs et de l'étape d'initialisation à l'aide d'`iota()`. Notez que `value_iterator<T>` n'est pas un type standard, mais que nous pouvons le définir nous-mêmes. Dans cette section, nous allons donc nous attarder à la définition d'un type `value_iterator<T>`.

Conceptualiser `value_iterator`

Le type `value_iterator` représentera une valeur *itérable*, donc sur laquelle il est possible d'itérer Il offrira les opérations suivantes :

- construction par défaut, paramétrique et par copie;
- comparaisons (les opérateurs relationnels), qui reposeront sur la valeur représentée (p. ex. : deux `value_iterator` de même type seront égaux s'ils représentent la même valeur);
- successeur et prédécesseur (opérateurs `++` et `--`) de même qu'avancer et reculer de `n` positions (de `n` valeurs);
- affectation;
- permutation de valeurs; et
- déréférencement (opérateur `*` unaire préfixé).

On pourrait aussi envisager d'autres opérations (par exemple l'opérateur `->` pour obtenir un `value_iterator` sur des pointeurs) mais ce serait probablement superflu (un pointeur est déjà un itérateur; il n'y aurait pas de valeur ajoutée).

Je n'implémenterai pas tous les opérateurs pertinents dans cette section, par souci d'économie. Parmi ceux opérateurs qu'il serait utile d'ajouter (et qui sont banals à implémenter), on trouve les opérateurs `+`, `-`, `+=` et `-=` prenant un entier en tant qu'opérande de droite. Idéalement, on ajouterait aussi l'opérateur `-` entre deux instances de `value_iterator<T>`.

Une solution possible à ce problème est celle proposée à droite. Notez que la Sainte-Trinité s'applique dans le cas d'un `value_iterator`, qui est (c'est le cas de le dire) un type valeur.

On pourrait lui ajouter plusieurs opérateurs, indiqués dans les commentaires, ce que je vous invite à faire en exercice. Notez que pour avoir un véritable itérateur à accès direct (*Random Access*), il faudrait aussi y ajouter `operator[]`, ce qui n'est pas en soi difficile – comment l'exprimeriez-vous?

Ce code suffit pour exprimer le concept d'itérateur sur un intervalle de valeurs. Cependant, il pourrait être plus simple d'utilisation. En effet, écrire quelque chose d'aussi long que l'expression `value_iterator<int>{3}` est en effet un peu lourd : à partir de la valeur `3`, on serait en droit de s'attendre à ce que le mot `int` soit implicite.

C'est là l'apport de la fonction `val_it()`, comme nous le verrons dans le code client. C'est une fonction génératrice permettant, à coût zéro, d'alléger l'écriture du code client.

Notez que `value_iterator` dérive de `std::iterator` de manière à définir les types internes et publics attendu d'une telle classe. Ceci permettra d'utiliser sans peine `std::iterator_traits` sur un `value_iterator`, par exemple.

Si nous ne définissons pas les traits de nos itérateurs, alors certains algorithmes, comme par exemple `std::distance()`, seront soit inefficaces, soit inutilisables avec eux.

```
#ifndef VALUE_ITERATOR_H
#define VALUE_ITERATOR_H
#include <iterator>
template <class T>
   class value_iterator
      : public std::iterator<
           std::random_access_iterator_tag, T, T
        >
   {
      T cur;
   public:
      constexpr value_iterator(const T &val): cur{val}{
      }
      value_iterator& operator++() {
         ++cur;
         return *this;
      }
      value_iterator operator++(int) {
         auto temp = *this;
         operator++();
         return temp;
      }
      // opérateurs --,+=,-= (banals à faire)
      bool operator==(const value_iterator &vi) const {
         return cur == vi.cur;
      }
      bool operator<(const value_iterator &vi) const {
         return cur < vi.cur;
      }
      // opérateurs !=, <=, >, >= (banals à faire)
      T operator*() const {
         return cur;
      }
      friend value_iterator operator+
         (const value_iterator &vi, int n) {
         return { vi.cur + n };
      }
      friend value_iterator operator+
         (int n, const value_iterator &vi) {
          return { vi.cur + n };
      }
      // etc.
   };
template <class T>
   value_iterator<T> val_it(const T &val) {
      return { val };
   }
#endif
```

En pratique, donc, un programme comme celui-ci…

```
// ... inclusions et using...
int main() {
   const int VAL_MIN = 0, VAL_MAX = 100;
   copy(value_iterator<int>{VAL_MIN}, value_iterator<int>{VAL_MAX},
        ostream_iterator<int>{cout, " "});
   cout << endl << "Distance: "
        << distance(value_iterator<int>{VAL_MIN}, value_iterator<int>{VAL_MAX})
        << endl;
}
```

…peut s'exprimer de manière bien plus simple et bien plus légère :

```
// ... inclusions et using...
int main() {
   const int VAL_MIN = 0, VAL_MAX = 100;
   copy(val_it(VAL_MIN), val_it(VAL_MAX), ostream_iterator<int>{cout, " "});
   cout << endl << "Distance: " << distance(val_it(VAL_MIN), val_it(VAL_MAX)) << endl;
}
```

Exercices – Série 04

EX00 – Pourrait-on concevoir une classe `Pixel` et utiliser un itérateur pour parcourir tous les pixels d'un écran de manière à faciliter la mise en place d'un effet logiciel (faire tendre les points vers une teinte particulière de rouge, par exemple)? Entendu ici que ce type d'opération serait nettement plus efficace si implémenté de manière matérielle, mais…

EX01 – Pourrait-on concevoir une classe `Sommet` et itérer à travers les sommets d'un maillage pour lui appliquer un effet de déformation (le faire fondre au sol, par exemple)?

EX02 – Complétez `value_iterator` avec les opérateurs `+`, `-`, `+=` et `-=`, chacun prenant un entier comme opérande de droite.

EX03 – Ajoutez à `value_iterator` avec l'opérateur `[]` dans sa déclinaison `const`. Expliquez vos choix d'implémentation.

EX04 – Est-il pertinent d'implémenter une sémantique de mouvement sur `value_iterator`? Expliquez votre point de vue et complétez votre explication par des exemples concrets.

EX05 – Est-il pertinent d'implémenter un constructeur de conversion sur `value_iterator`? Expliquez votre point de vue et complétez votre explication par des exemples concrets.

EX06 – Définissez un type `value_range<T>` exposant `begin()` et `end()` et montrez comment il est possible de faire compiler un exemple tel que :

```
// ... inclusions et using ...
int main() {
   for(auto i : value_range<int>{0,100})
      cout << i << ' ';
   cout << endl;
}
```

Vous trouverez une proposition de solution à un problème connexe dans ***Annexe 03 – Itérer sur des valeurs (écriture alternative)***.

Définir une file circulaire à l'aide d'itérateurs

Une autre application relativement naturelle des itérateurs est la conception d'une file circulaire d'éléments. En soi, une file circulaire n'est pas tant une propriété du conteneur qu'une adaptation des itérateurs d'un conteneur existant, un peu comme les itérateurs inverses (les `reverse_iterator` offerts pour les itérateurs bidirectionnels) sont des adaptateurs sur des itérateurs capables de progresser vers l'avant.

Nous nommerons ***itérateur circulaire*** (en C++, un `circ_iterator`) l'adaptateur d'itérateur qui nous permettra de définir le comportement cyclique attendu sur les itérateurs d'un conteneur.

Nous aurions aussi pu utiliser le terme *itérateur cyclique*; simple question de goût.

On comprendra qu'un itérateur circulaire doit au moins permettre de progresser d'un pas vers l'avant (d'être au moins de catégorie *Forward Iterator*). Les férus de programmation générique parmi vous verront peut-être une opportunité de réaliser une assertion statique ici, mais ce n'est pas vraiment nécessaire, sinon par souci de clarté.

Pour implémenter un itérateur circulaire, nous aurons besoin de connaître quatre éléments, tous définis par des itérateurs traditionnels sur la séquence en fonction de laquelle l'adaptateur sera construit, soit :

- le début de la séquence source (là où l'itérateur circulaire devra reprendre s'il atteint la fin de la séquence originale);
- la fin de la séquence source (pour détecter le besoin de réaliser un mouvement cyclique et de repartir au début);
- le début de la séquence déterminant un cycle (rien ne force le cycle à commencer au début de la séquence source, après tout), pour permettre la détection éventuelle d'une fin de cycle; et
- la position courante dans le cycle, valeur de l'itérateur au sens traditionnel.

Examinons une implémentation opérationnelle. Tel qu'attendu, un itérateur circulaire sera un adaptateur d'itérateur, donc un objet générique sur la base d'une classe d'itérateur existante.

La manière la plus simple de mettre en place les bases d'un tel itérateur est de dériver notre classe d'un descriptif d'itérateur dont les traits s'inspirent des traits de l'itérateur autour duquel se construit l'adaptateur. De cette manière, les définitions seront conformes aux exigences du standard et aux règles propres à l'itérateur d'origine.

```
#ifndef FILE_CIRCULAIRE_H
#define FILE_CIRCULAIRE_H
#include <iterator> // les traits
template <class It>
   class circ_iterator
      : public std::iterator <
           typename It::iterator_category,
           typename It::value_type
        >
   {
```

Les attributs dont nous aurons besoin pour réaliser notre implémentation seront ceux annoncés plus haut.

```
    It debsrc, finsrc, debcirc, cur;
public:
```

C'est sans surprises que nous constaterons que notre constructeur paramétrique sera un peu lourd.

La position courante à la construction d'un itérateur circulaire sera celle du début de cycle. Notons que notre classe sera un type valeur; la Sainte-Trinité s'appliquera.

```
    circ_iterator(It debsrc, It finsrc, It debcirc)
       : debsrc{debsrc},  finsrc{finsrc}, debcirc{debcirc}, cur{debcirc}
    {
    }
```

Une partie importante du travail d'un itérateur est, évidemment, d'itérer à travers une séquence.

L'opérateur ++ préfixé définit les règles de progression (la version postfixée repose sur lui). C'est ici que la logique est la plus lourde puisqu'il faut définir un standard d'arrêt de la progression.

Pour faciliter la logique, nous considérerons que progresser sur la fin de la source nous amène simplement à revenir au début de la source et ne constitue pas une atteinte de la fin de la séquence. Par contre, progresser vers le début de cycle constitue une atteinte de fin de séquence et propulsera l'itérateur courant vers la fin de la séquence source.

Conserver le concept de fin de la source comme fin effective de séquence facilitera entre autres la mise en place d'itérateurs inverses.

```
    circ_iterator& operator++() {
       ++cur;
       if (cur == debcirc)
          cur = finsrc;
       else if (cur == finsrc)
          if (debcirc == debsrc)
             cur = finsrc;
          else
             cur = debutsrc;
       return *this;
    }
    circ_iterator operator++(int) {
       auto temp{*this};
       operator++();
       return temp;
    }
```

Ce code pourrait être spécialisé selon le type d'itérateur sous-jacent.

Le fait d'utiliser la fin de la source comme marqueur de fin de séquence impliquera par contre la nécessité d'implémenter la comparaison à deux niveaux, soit entre deux itérateurs circulaires et entre un itérateur circulaire et un itérateur source.

L'implémentation proposée ici utilise des opérateurs sous forme de fonctions globales, ce qui permet d'utiliser l'itérateur circulaire à gauche ou à droite de comparaisons.

```
      friend bool operator==(const circ_iterator& i0, const circ_iterator& i1) const {
         return i0.cur == i1.cur;
      }
      friend bool operator!=(const circ_iterator& i0, const circ_iterator& i1) const {
         return !(i0== i1);
      }
      friend bool operator==(const circ_iterator &i0, const It &i1) const {
         return i0.cur == i1;
      }
      friend bool operator==(const It &i0, const circ_iterator &i1) const {
         return i0 == i1.cur;
      }
      friend bool operator!=(const circ_iterator &i0, const It &i1) const {
         return !(i0 == i1);
      }
      friend bool operator!=(const It &i0, const circ_iterator &i1) const {
         return !(i0 == i1);
      }
```

Les opérateurs de déréférencement sont banals, du fait qu'ils ne servent que d'indirection vers les valeurs pointées par l'itérateur courant sur la séquence source. Notez tout de même que l'opérateur `->` est un drôle d'oiseau.

```
      value_type* operator->() {
         return *cur;
      }
      const value_type* operator->() const {
         return *cur;
      }
      value_type& operator*() {
         return *cur;
      }
      const value_type& operator*() const {
         return *cur;
      }
```

Enfin, un opérateur `<` simplet est proposé pour faciliter le recours à des algorithmes standard, mais il est clair que la logique proposée ici pourrait (devrait!) être raffinée pour tenir compte du caractère circulaire de la séquence. Le défi est d'en faire une opération efficace malgré la hausse de complexité encourue.

```
      bool operator< (const circ_iterator &it) const {
         return cur < it.cur;
      }
   };
```

Un adaptateur d'itérateur muni d'un constructeur à trois paramètres est trop lourd syntaxiquement pour une utilisation normale. Deux options s'offrent à nous pour alléger son utilisation, soit la mise en place de fonctions génératrices ou l'intégration de l'adaptateur dans un conteneur qui joue lui-même le rôle d'adaptateur sur un conteneur existant.

Nous prendrons ici la deuxième option. La première sera laissée en exercice.

Tout d'abord, notre `file_circulaire` sera un adaptateur sur un conteneur conforme au standard.

Par souci de convivialité, nous utiliserons par défaut le vecteur standard à titre de conteneur, mais nous laisserons le code client faire un autre choix s'il l'estime pertinent.

```
#include <vector>
template <class T,
          class C = std::vector<T> >
   class file_circulaire {
```

Une grande partie du code de notre file circulaire sera faite de définitions de types internes. Il faut comprendre qu'en plus des quatre itérateurs usuels (avant, avant constant, à rebours, à rebours constant), nous en ajoutons ici quatre autres qui seront leurs contreparties circulaires.

Tout cela peut sembler lourd, mais constatons que ce que nous faisons ici est exposer une masse importante de fonctionnalités au code client sans avoir à définir cette fonctionnalité.

Relevez au passage la notation `::circ_iterator` qui permet de distinguer la classe en soi des définitions internes.

```
public:
   using conteneur_type = C;
   using value_type = typename
      conteneur_type::value_type;
   using size_type = typename conteneur_type::size_type;
   using reference = typename conteneur_type::reference;
   using pointer = typename conteneur_type::pointer;
   using iterator = typename conteneur_type::iterator;
   using const_iterator = typename
      conteneur_type::const_iterator;
   using reverse_iterator = typename
      conteneur_type::reverse_iterator;
   using const_reverse_iterator = typename
      conteneur_type::const_reverse_iterator;
   using circ_iterator = ::circ_iterator<iterator>;
   using const_circ_iterator =
      ::circ_iterator<const_iterator>;
   using circular_reverse_iterator =
      ::circ_iterator<reverse_iterator>;
   using const_reverse_circ_iterator =
      ::circ_iterator<const_reverse_iterator>;
```

J'ai choisi de proposer le conteneur d'origine sous forme d'un attribut privé, mais notez que la plupart des adaptateurs de conteneurs standards (tels `std::stack` et `std::queue`) exposent ce conteneur sous la forme d'un attribut d'instance public nommé c pour permettre au code client d'accéder à ses fonctionnalités si ce choix s'avère pragmatique.

```
private:
   conteneur_type donnees;
```

Les constructeurs et le destructeur sont pratiquement évidents. Le destructeur et le constructeur par copie sont omis parce que la version par défaut suffit.

Notez toutefois la présence d'un constructeur de séquence, par souci de conformité au standard et par souci de convivialité.

Quelques méthodes utilitaires typiques sont exposées pour faciliter la vie du code client.

On pourrait sans doute accroître la liste de méthodes utilitaires ici mais il faut être prudent puisque nous ne savons pas quel conteneur le code client a choisi en tant que représentation interne pour la file circulaire.

Les méthodes `begin()` et `end()` sont de simples relais vers les méthodes équivalentes du conteneur sous-jacent, et il en va de même pour les méthodes `rbegin()` et `rend()`. Ces relais sont typiques des adaptateurs.

Grâce à `SFINAE` [POOv02], si le conteneur sous-jacent ne supporte pas ces méthodes, le code de notre itérateur circulaire compilera quand même, dans la mesure où personne n'invoque les méthodes manquantes.

```
public:
   file_circulaire() = default;
   template <class It>
      file_circulaire(It debut, It fin)
         : donnees{debut, fin}
      {
      }

   void push_back(const value_type &val) {
      donnees.push_back(val);
   }
   size_type size() const noexcept {
      return donnees.size();
   }
   bool empty() const noexcept {
      return donnees.empty();
   }
   iterator begin() {
      return donnees.begin();
   }
   const_iterator begin() const {
      return donnees.begin();
   }
   iterator end() {
      return donnees.end();
   }
   const_iterator end() const {
      return donnees.end();
   }
   reverse_iterator rbegin() {
      return donnees.rbegin();
   }
   const_reverse_iterator rbegin() const {
      return donnees.rbegin();
   }
   reverse_iterator rend() {
      return donnees.rend();
   }
   const_reverse_iterator rend() const {
      return donnees.rend();
   }
```

Il nous faut ensuite une métaphore pour exposer les itérateurs circulaires, constants ou non, à partir de ce conteneur.

J'ai choisi de nommer les méthodes offrant ces services `begin_from()` et `end_from()`. Ce sont des noms de mon cru, ces itérateurs ne faisant pas partie du standard.

Leur comportement par défaut sera essentiellement le même que ceux de `begin()` et de `end()`, mais il sera possible de leur suppléer un point de départ dans la séquence source pour obtenir un itérateur circulaire conforme.

En procédant ainsi, la complexité de la construction des itérateurs circulaires se trouve encapsulée dans la file circulaire.

```
        circ_iterator begin_from
           (iterator cur = begin()) {
           return {begin(), end(), cur};
        }
        circ_iterator end_from
           (iterator cur = begin()) {
           return {begin(), end(), end()};
        }
        const_circ_iterator begin_from
           (const_iterator cur = begin()) const {
           return {begin(), end(), cur};
        }
        const_circ_iterator end_from
           (const_iterator cur = begin()) const {
           return {begin(), end(), end()};
        }
```

La logique est la même pour les itérateurs à rebours. J'ai nommé les méthodes `rbegin_from()` et `rend_from()` dans l'optique où il me semblait raisonnable de garder une certaine conformité syntaxique avec les méthodes usuelles d'accès aux itérateurs d'un conteneur.

Notez que, bien que les noms de types d'itérateurs soient lourds, le code client n'aura habituellement pas besoin de les expliciter, ces noms étant la plupart du temps déduits par la mécanique d'instanciation des *templates*.

Les noms de méthodes sont, eux, simples et utilisables.

La question, enfin, est de savoir comment tester tout cela, dans le but de nous donner un certain seuil de confiance.

```
        circular_reverse_iterator rbegin_from
           (reverse_iterator cur = rbegin()) {
           return {rbegin(), rend(), cur};
        }
        circular_reverse_iterator rend_from
           (reverse_iterator cur = rbegin()) {
           return {rbegin(), rend(), rend()};
        }
        const_reverse_circ_iterator rbegin_from
           (const_reverse_iterator cur = rbegin()) const {
           return {rbegin(), rend(), cur};
        }
        const_reverse_circ_iterator end_from
           (const_reverse_iterator cur = rbegin()) const {
           return {rbegin(), rend(), rend()};
        }
    };
#endif
```

Une ébauche de programme de test possible irait comme suit :

- tout d'abord, remplir la file circulaire avec quelques éléments. On aurait pu utiliser le constructeur de séquence ici;
- examiner un parcours conventionnel des éléments de notre file;
- examiner un parcours cyclique à partir d'un point arbitraire de la file circulaire; enfin
- examiner un parcours à rebours cyclique des éléments de la file.

```
// inclusions diverses
int main() {
   file_circulaire<int> f;
   for (int i = 0; i < 5; ++i)
      f.push_back(i);
   ostream_iterator<int> out{cout, " "};
   copy(begin(f), end(f), out);
   cout << endl;
   copy f.begin_from(begin(f)+2),
        f.end_from(begin(f)+2), out);
   cout << endl;
   copy(f.rbegin_from(f.rbegin()+2),
        f.rend_from(f.rbegin()+2), out);
   cout << endl;
}
```

Exercices – Série 05

EX00 – Complétez les comparaisons avec l'opérateur `==` et avec l'opérateur `!=` sur la classe `circ_iterator` pour que ces opérations fonctionnent correctement peu importe l'ordre de leurs opérandes.

EX01 – Ajoutez les méthodes nécessaires à `file_circulaire` pour faciliter une insertion en début ou en fin de séquence circulaire et testez rigoureusement le tout.

EX02 – Implémentez un véritable opérateur d'ordonnancement (opérateur `<`) sur une paire de `circ_iterator` qui soit $O(1)$. Prenez soin de définir au préalable ce que signifie l'opérateur `<` sur deux instances de `circ_iterator` (c'est plus subtil qu'il n'y paraît).

EX03 – Implémentez un opérateur `[]` sur un `circ_iterator` qui soit $O(1)$ si l'itérateur sous-jacent est un *Random Access Iterator*. **Un peu plus difficile** : faites en sorte que cet opérateur ne soit pas disponible pour les autres types d'itérateurs (idéalement, en générant une erreur à la compilation).

EX04 – Implémentez des opérateurs `+=` et `-=` prenant en paramètre un entier, de la manière aussi efficace que possible selon le type d'itérateur sous-jacent un `circ_iterator`.

EX05 – Implémentez des opérateurs `+` et `-`prenant en paramètre un entier, de la manière aussi efficace que possible selon le type d'itérateur sous-jacent un `circ_iterator`.

EX06 – Définissez des fonctions `make_circ_iterator()` prenant en paramètre un conteneur et un point de départ dans la séquence source pour faciliter le travail de gens désireux d'obtenir une séquence circulaire sans avoir recours à `file_circulaire`. Inspirez-vous de fonctions semblables dans le standard (par exemple `std::make_pair()`). Montrez en quoi cela allège le code client.

EX07 – Spécialisez `operator++()` pour un `circ_iterator` en fonction de sa catégorie, et faites-le de la manière la plus efficace possible (en performance et en simplicité. d'entretien).

Traits, *polymorphisme statique* et bases de métaprogrammation

Les traits sont aussi une technique qui permet d'étendre la gamme des comportements génériques de classes existantes ou groupées par catégories.

Imaginons que plusieurs équipes aient mis au point des classes représentant des créatures, que vous et vos collègues nommez des *bibittes*. Imaginons aussi que, peu importe la raison (besoins spécifiques du projet, performance, intégration de classes d'autres projets, …), les diverses classes de bibittes ne dérivent pas toutes d'un parent commun (peut-être dérivent-elles toutes *sauf une* d'un même parent, qui sait?).

Autre possibilité : imaginons qu'une hiérarchie de bibittes existe déjà et que des comportements généraux aient été définis chez une classe parent, puis que survienne le besoin d'ajouter des comportements génériques à des catégories de bibittes sans modifier la structure de la hiérarchie ou les classes qui la composent. Il peut y avoir plusieurs raisons pour refuser de modifier l'existant : par exemple, il est possible que les bibittes aient été écrites de manière à optimiser l'utilisation d'une ressource, par exemple l'antémémoire, et qu'ajouter ne serait-ce qu'un *byte* à une bibitte aurait une incidence néfaste sur la performance d'ensemble du système.

Imaginons maintenant que nous souhaitions quand même un comportement générique spécialisé *par famille* de bibittes :

- si une bibitte et puante, nous pourrions souhaiter qu'elle ait un impact funeste sur la végétation avoisinante;
- si une bibitte est envoûtante, nous pourrions souhaiter que les réactions des tiers envers elle soit influencées de manière positive; enfin
- si une bibitte est terrible, alors elle doit pouvoir rugir alors que si elle n'est pas terrible, elle doit pouvoir murmurer de manière sympathique.

> Il et important de comprendre ici que la technique des traits opère *a priori* au niveau des types, pas au niveau des instances. Elle prend donc son sens lorsque nous exprimons des algorithmes applicables à des familles d'objets.
>
> Le polymorphisme, lui, s'applique sur une base instance. Les deux tactiques sont complémentaires.

Nous chercherons donc dans cette section à montrer comment réaliser une forme de *polymorphisme statique*, résolu à la compilation sur la base de types et de leurs traits. Nous bâtirons notre démonstration à l'aide de trois classes de bibittes : la classe `Orque`, la classe `Troll` et la classe `Elfe`.

Parce que ces trois classes nous seront d'abord utiles en tant que classes, nous ne nous intéresserons pas vraiment à leurs membres d'instances. Notre regard se posera sur le fait qu'elles existent et qu'elles peuvent être décrites à partir de traits.

Pour nos fins, les traits de bibittes pourront être définis à partir de constantes :

- une constante `EST_TERRIBLE` pour indiquer si les bibittes d'une famille donnée sont terribles ou non;
- une constante `EST_PUANT` pour indiquer si les bibittes d'une famille donnée sont puantes ou non; et
- une constante `EST_ENVOUTANT` pour indiquer si les bibittes d'une famille donnée sont envoûtantes ou non.

```
class Orque {
   // ...
};
class Troll {
   // ...
};
class Elfe {
   // ...
};
```

Il nous faut ensuite faire un choix de design quant aux traits que nous définirons : supporterons-nous une gamme de traits par défaut, que nous spécialiserons ensuite de manière à décrire nos diverses familles de bibittes, ou déterminerons-nous qu'un programme tentant d'appliquer un algorithme destiné à des bibittes ne doit tout simplement pas compiler si on cherche à l'appliquer à un type qui n'a pas de traits de bibittes?

L'option utilisant des traits par défaut serait implémentée comme proposé à droite.

La version générale, applicable à un type `T` quelconque, définit les divers traits comme étant faux. L'idée derrière cette manœuvre est de spécifier que la bibitte typique n'a rien de bien particulier.

Les versions spécifiquement appliquées à des types de bibittes données mettent en relief les caractéristiques particulières de chacun de ces types de bibittes.

Dans le cas où on ne voudrait pas de traits par défaut (donc où un algorithme s'appliquant à une bibitte ne devrait pas compiler si `T` n'est pas une bibitte possédant des traits clairs), alors il suffirait de définir le cas général comme suit :

```
typedef <class> struct bibitte_traits;
```

```
template <class T>
   struct bibitte_traits {
      static constexpr const bool
         EST_TERRIBLE = false,
         EST_MALODORANT = false,
         EST_ENVOUTANT = false;
   };
template <>
   struct bibitte_traits<Orque> {
      static constexpr const bool
         EST_TERRIBLE = true,
         EST_MALODORANT = false,
         EST_ENVOUTANT = false;
   };
template <>
   struct bibitte_traits<Troll> {
      static constexpr const bool
         EST_TERRIBLE = true,
         EST_MALODORANT = true,
         EST_ENVOUTANT = false;
   };
template <>
   struct bibitte_traits<Elfe> {
      static constexpr const bool
         EST_TERRIBLE = false,
         EST_MALODORANT = false,
         EST_ENVOUTANT = true;
   };
```

Le code client souhaité

Prenons, pour quelques instants, le problème du point de vue du code client. Un programme de test correct pour ce que nous souhaitons mettre en place comme mécanisme serait celui proposé à droite.

```
int main() {
   Elfe e;
   faire_peur(e);
   Orque Urg;
   faire_peur(Urg);
}
```

Ce petit programme devrait causer une réaction terrible pour `Urg` (disons quelque chose comme hurler `"ARRRRRHHH"` à la console) alors que la réaction pour un `Elfe` comme `e` devrait être beaucoup plus douce (par exemple afficher `"Coucou!"`).

Souvenons-nous que les classes `Elfe` et `Orque` n'ont, ici, aucun parent commun. Il faut permettre au compilateur de réagir correctement selon les traits de bibittes d'un `Orque` ou d'un `Elfe`.

Nous pourrions évidemment surcharger `faire_peur()` pour chaque type de bibitte mais cela ne nous permettrait pas de faire hurler, de manière générique, toute bibitte terrible (à moins qu'elles aient toutes un parent commun, mais encore une fois nous présumons que ce ne soit pas possible ici).

Un sélecteur statique de type

Maintenant, il nous faut mettre au point un mécanisme capable de choisir automatiquement une version appropriée de `faire_peur()` en fonction des traits de chaque type de bibitte.

On pourrait aussi y aller avec des catégories, comme dans le cas des itérateurs. Je vous invite à procéder ainsi si vous voulez vous assurer d'avoir bien compris la manœuvre.

Les traits sont essentiellement faits d'information statique, constituée de types et de constantes, avec quelques méthodes de classes au besoin. Nous profiterons de ce savoir statique pour que le choix de la bonne version de la fonction `faire_peur()` puisse être réalisé à la compilation du programme, réduisant à néant les coûts décisionnels à l'exécution.

Imaginons donc deux types distincts, visibles à droite. Le type `terrible` décrira tout type `T` pour lequel le trait `bibitte_traits<T>::EST_TERRIBLE` est vrai, et le type `pas_terrible`, lui, correspondra aux types `T` pour lesquels le trait `bibitte_traits<T>::EST_TERRIBLE` est faux.

```
class terrible{};
class pas_terrible{};
```

Ces deux classes sont vides. Leur rôle est de représenter chacune un concept distinct et d'être différentes l'une de l'autre.

L'étape suivante est de définir une abstraction permettant de choisir un type à partir d'un critère. Ici, le critère est un booléen parce que le trait qui nous intéresse s'exprime sous cette forme.

Le critère sera le type `selecteur_terreur`, qui sera un type générique dépendant d'un `bool`. Selon la valeur du `bool` en question, `selecteur_terreur` définira un type interne et public qui sera équivalent soit à `terrible`, soit à `pas_terrible`.

```
template <bool>
   struct selecteur_terreur;
template <>
   struct selecteur_terreur<true> {
      using type = terrible;
   };
template <>
   struct selecteur_terreur<false> {
      using type pas_terrible;
   };
```

Ce que nous venons de faire, sans tambour ni trompette, est essentiellement de faire une sorte d'alternative (de `if`) statique. Notre alternative dépend d'une donnée connue à la compilation (une constante booléenne) et son résultat est un type, aussi connu à la compilation.

D'ailleurs, la manière idiomatique d'en arriver au même résultat est d'utiliser le trait standard `std::conditional`, que l'on trouve dans `<type_traits>` :

```
template <bool cond>
   using selecteur_terreur = typename std::conditional<cond, terrible, pas_terrible>::type;
```

Depuis C++ 14, il existe même une écriture plus courte pour obtenir précisément le même résultat, soit :

```
template <bool cond>
   using selecteur_terreur = std::conditional_t<cond, terrible, pas_terrible>;
```

Depuis C++ 17, on pourrait évidemment utiliser `if constexpr`.

Des techniques de programmation statiques, résolues à la compilation et déterminant des données aussi connues à la compilation, sont en fait des formes de programmation dont le résultat est un programme… de la **métaprogrammation**.

Réaliser le polymorphisme statique

Enfin, il ne nous reste plus qu'à nous assurer que `faire_peur()` soit invoqué correctement en fonction des types impliqués.

La technique pour y arriver a déjà été explorée plus haut pour la fonction `std::distance()` : il suffit d'injecter un paramètre anonyme de type distinct (et dépendant des traits) pour chaque version de la fonction `faire_peur()`.

Les versions spécialisées de `faire_peur()` pour les classes `terrible` et `pas_terrible` vont de soi. Leur paramètre sélectif ne fait qu'exister; on ne s'en sert pas, ce qui explique qu'on n'ait pas à le nommer.

La version générale, utilisée par le code client, définit un type `terreur_type` interne à partir du sélecteur statique de type et des traits de bibitte du type `T` puis invoque une version à deux paramètres du sous-programme `faire_peur()` en fonction d'une instance (vide!) de `terreur_type`.

```
template <class T>
   void faire_peur(const T&, terrible) {
      cout << "ARRRRRHHH!";
   }
template <class T>
   void faire_peur(const T&, pas_terrible) {
      cout << "Coucou!";
   }
template <class T>
   void faire_peur(const T &e) {
      selecteur_terreur <
         bibitte_traits<T>::EST_TERRIBLE
      > terreur_type;
      faire_peur(e, terreur_type{});
   }
// ... ou encore ...
template <class T>
   void faire_peur(const T &e) {
      if constexpr(bibitte_traits<T>::EST_TERRIBLE)
         cout << "ARRRRHHH!";
      else
         cout << "Coucou|";
   }
```

Si vous préférez la version « manuelle » de `selecteur_terreur`, celle qui ne repose pas sur `std::conditional`, alors il vous faudra remplacer ceci :

```
selecteur_terreur <bibitte_traits<T>::EST_TERRIBLE> terreur_type;
```

… par cela :

```
template <class T>
   using terreur_type = typename selecteur_terreur<bibitte_traits<T>::EST_TERRIBLE>::type;
```

Alléger l'écriture des traits

Vous aurez peut-être remarqué que les `bibitte_traits` sont, pour le moment, fastidieux à définir. Pour tout type de bibitte, l'attente est que soient définis tous les traits pertinents, même si un seul trait d'une sorte de bibitte donnée diffère de la version générale.

Une stratégie plus propre et plus utile serait d'avoir un profil de bibitte standard et de spécialiser seulement les traits qui en divergent. Cette stratégie a plusieurs avantages. En particulier :

- elle allège l'écriture des traits pour un type donné; et
- elle fait en sorte qu'il n'y ait plus de traits par défaut pour un type qui ne soit pas explicitement une bibitte.

Le truc pour y arriver (sans changer quoi que ce soit dans le reste du programme) va comme suit.

Tout d'abord, définir une classe `bibitte_traits` générique (et générale), qui ne fait qu'exister.

```
template <class T>
   struct bibitte_traits;
```

Ceci permet la spécialisation ultérieure du type et fait en sorte qu'il n'existe aucune version par défaut pour un type `T` quelconque.

Définir une classe vide `BibitteBase` (par exemple) qui servira indirectement à définir les traits par défaut d'une bibitte.

```
class BibitteBase {};
```

Ensuite, définir les traits par défaut à partir du type générique `bibitte_traits` appliqué à la classe `BibitteBase`.

Nous avons ici une classe vide qui ne sert que pour servir d'aiguilleur à un type générique.

Ensuite, les traits des autres types de bibittes seront définis en dérivant leur propre classe `bibitte_traits` de la classe générique `bibitte_traits` appliquée au type `BibitteBase`.

Puisque les enfantes héritent des constantes et des types du parent, il ne leur reste plus qu'à cacher ce qui ne leur convient pas derrière des abstractions de leur propre cru. En termes techniques, seuls les traits différents de la gamme par défaut doivent être définis.

```
template <>
   struct bibitte_traits <BibitteBase> {
      static constexpr bool EST_TERRIBLE = false,
                            EST_MALODORANT = false,
                            EST_ENVOUTANT = false;
   };

template <>
   struct bibitte_traits<Orque>
      : bibitte_traits<BibitteBase> {
      static constexpr bool EST_TERRIBLE = true;
   };
template <>
   struct bibitte_traits<Troll>
      : bibitte_traits<BibitteBase> {
      static constexpr bool EST_TERRIBLE = true,
                            EST_MALODORANT = true;
   };
template <>
   struct bibitte_traits<Elfe>
      : bibitte_traits<BibitteBase> {
      static constexpr bool EST_ENVOUTANT = true;
   };
```

Exercices – Série 06

EX00 – Imaginez une fonction `rencontre()` entre une instance de type `T` (qui doit être une bibitte) et une référence à un `Envoutable`. Tout `Envoutable` doit offrir une méthode polymorphique `apprivoiser()` qui ne devrait être invoquée que si `T` a le trait `EST_ENVOUTANT` de valeur `true`. Écrivez la fonction `rencontre()`.

EX01 – Implémentez `faire_peur()` à l'aide de catégories (comme dans le cas des catégories d'itérateurs) plutôt qu'à l'aide d'un sélecteur de types.

EX02 – Pouvez-vous implémenter un jeu de roche-papier-ciseaux à l'aide de la technique décrite dans cette section? Si oui, faites-le; sinon, expliquez pourquoi.

Composition de fonctions

Une autre application typique des lieurs est la composition de fonctions. Pensons par exemple au concept de « entre `a` et `b` », qui signifie être supérieur ou égal à `a` et inférieur ou égal à `b`. Une stratégie intéressante existe sous `STL` mais ne fait pas partie du standard de C++ au moment d'écrire ces lignes [StlBiComp]. Écrire un lieur capable de réaliser cette tâche est un sympathique exercice de programmation générique.

Un exemple simple illustrant cette manœuvre pour `f(g(x))`, présumant donc que `f()` et `g()` sont des fonctions unaires (à un seul paramètre), serait le suivant.

Le lieur [POOv02] réalisant la composition des fonctions prendra en paramètre dès sa construction les fonctions à composer et les invoquera dans l'ordre lorsqu'on l'utilisera en tant que foncteur.

Question : est-ce que ce foncteur se comportera raisonnablement si `g(x)` lève une exception? Expliquez votre réponse.

```
template <class F, class G>
   class f_g_x_impl {
      F f;
      G g;
   public:
      f_g_x_impl (F f, G g) : f{f}, g{g} {
      }
      // depuis C++14, decltype... est superflu ici
      template <class Arg>
         auto operator()(Arg x) -> decltype(f(g(x))) {
            return f(g(x));
         }
   };
```

Pour alléger la construction du foncteur, nous offrirons une fonction de fabrication. Puisque c'est elle qui sera utilisée par le code client, c'est à elle que nous donnerons le nom le plus convivial (`f_g_x()`).

```
template <class F, class G>
   f_g_x_impl<F,G> f_g_x(F f, G g) {
      return f_g_x_impl<F,G>{f,g};
   }
```

Depuis C++ `14`, il est possible d'exprimer cette fonction de fabrication de manière plus simple encore.

```
template <class F, class G>
   f_g_x_impl<F,G> f_g_x(F f, G g) {
      return {f,g};
   }
```

Le compilateur générera la fonction de fabrication à partir des types des paramètres, ce qui permettra au code client de ne pas se préoccuper des types précis impliqués; cette technique est très répandue en programmation générique (tous les lieurs, plus haut, s'en servent, de `ptr_fun` à `bind2nd`).

Les fonctions à composer pour notre exemple seront celles définies à droite, soit `Cube` et `racine_carree()`.

Pour `Cube`, j'ai utilisé un foncteur mais par simple intérêt pédagogique (pour montrer comment faire) car une fonction aurait suffi.

```
struct Cube {
   template <class T>
      auto operator()(T x) const {
         return x * x * x;
      }
};
#include <cmath>
double racine_carree(double val) {
   return std::sqrt(val);
}
```

La raison pour laquelle `racine_carree()` est utilisée (plutôt que `std::sqrt()` directement) est qu'il existe trois déclinaisons distinctes de `std::sqrt()` et que je souhaitais éliminer l'ambiguïté à l'utilisation.

Le programme de démonstration proposé à droite prend un tableau de double, transforme chacun de ses éléments par la composition `f(g(x))` décrite par le foncteur, puis affiche les éléments transformés.

```
#include <iostream>
#include <algorithm>
#include <iterator>
int main() {
   using namespace std;
   double tab[]{ 1.0, 2.0, 3.0 };
   transform(begin(tab), end(tab), begin(tab),
             f_g_x(racine_carree, Cube{}));
   copy(begin(tab), end(tab),
        ostream_iterator<double>{cout, " "});
   cout << endl;
}
```

Pour une étude beaucoup plus en profondeur de la composition de fonctions et des lieurs, voir [hdFComp], premier article d'une série de trois.

Avec une expression λ de C++ `14`, `f_g_x()` s'exprime comme un charme :

```
template <class F, class G>
   auto f_g_x(F f, G g) {
      return [=](auto x) { return f(g(x)); };
   }
```

C'est à la fois simple et efficace, par-dessus le marché!

Exercices – Série 07

EX00 – Notez que le foncteur de composition `f_g_x_impl` décrit dans cette section est lui-même un foncteur, et qu'il pourrait être impliqué dans une composition de fonctions. Écrivez un programme de test pour en faire la démonstration.

EX01 – En quoi une fonction de génération, comme `f_g_x()`, simplifie-t-elle le code client en comparaison avec une utilisation directe du foncteur `f_g_x_impl`? Écrivez le code client avec fonction génératrice, le même code client sans la fonction génératrice, et indiquez en quoi les exigences pour le code client diffèrent d'une version à l'autre.

EX02 – Donnez un exemple où une composition de fonctions constitue une optimisation sur le plan de la vitesse d'exécution pour un programme, et expliquez pourquoi.

Gestion avancée de la mémoire

La gestion de la mémoire, surtout celle allouée dynamiquement, est un dossier de bas niveau mais que même les algorithmes et les types les plus propres et les plus génériques ne peuvent escamoter complètement.

Il se trouve que C++ permet de prendre en charge ce mécanisme fondamental qu'est la stratégie d'allocation dynamique de mémoire d'un programme. Le système est très flexible : la prise en charge peut être faite de manière globale[46], de manière localisée pour une classe en particulier, de manière collaborative ou de plusieurs autres manières encore.

Dans cette section, nous examinerons plusieurs approches permettant de prendre en charge les mécanismes de gestion de la mémoire allouée dynamiquement, et nous verrons quand et comment les utiliser.

Dans les sections qui suivent, j'utiliserai allocation dynamique pour parler à la fois d'allocation et de libération dynamiques de mémoire, sauf s'il semble important de mettre spécifiquement l'accent sur la libération, ceci pour alléger le texte.

Pourquoi gérer soi-même la mémoire allouée dynamiquement

On peut se demander pourquoi un programme choisirait de gérer lui-même la stratégie d'allocation et de libération dynamique de mémoire. Après tout, n'est-ce pas un mécanisme fondamental de programmation? En ce sens, ne sera-t-il pas nécessairement mieux pris en charge par le langage lui-même que par des programmeurs près du domaine d'application?

En fait, non. Le langage peut (et doit, sans doute) offrir un mécanisme par défaut raisonnable mais que le mécanisme optimal pour tous les cas d'espèces n'existe pas.

Toute stratégie d'allocation dynamique de mémoire a des défauts, en particulier parce qu'elle ne connaît pas *a priori* les patrons d'utilisation de la mémoire des programmes; chaque programme a des besoins différents. Les usages culturels influencent le choix d'une bonne stratégie d'allocation dynamique de mémoire : par exemple, en langage C, le recours à de gros blocs de mémoire constitue souvent la principale raison derrière l'utilisation d'allocation dynamique de mémoire, alors qu'en Java, tous les objets, quels qu'ils soient, sont instanciés à travers de la mémoire allouée dynamiquement. La meilleure approche pour l'une et l'autre de ces cultures ne sera clairement pas la même.

Collecte automatique d'ordures

Parmi les questions ouvertes quant à l'allocation dynamique de mémoire, on pense entre autres à la pertinence d'une collecte automatique d'ordures. Devrait-on y avoir recours en tout temps? Jamais? Selon les besoins? Ces questions mettent feu à plusieurs discussions de fin de soirée entre spécialistes et *aficionados* de l'un ou l'autre des camps impliqués.

N'oubliez pas la distinction importante entre la libération de mémoire et la finalisation. Il est possible d'automatiser la libération de la mémoire par une collecte automatique d'ordures, mais les interdépendances entre objets rendre impraticable (peut-être même indécidable) la finalisation intelligente en présence de collecte non déterministe d'ordures. Il faudrait, pour y arriver, analyser dynamiquement et efficacement l'ensemble des relations entre objets…

[46] Prendre en charge les mécanismes globaux d'allocation dynamique de mémoire est risqué, puisque cela affecte même la gestion dynamique de simples entiers!

Java impose la collecte automatique d'ordures pour les objets et procède autrement pour les types primitifs, ceux-ci étant gérés sur la pile d'exécution (à moins bien sûr de servir d'attribut d'instance). Le recours systématique à la collecte automatique d'ordures est un choix technique, mais aussi un modèle de programmation, une philosophie.

Sous C++[47], aucun moteur de collecte automatique d'ordures n'est imposé. La philosophie de base du langage est qu'un programme ne doit payer que pour ce qu'il utilise.

C++ s'applique à une vaste gamme de domaines, et sur une énorme quantité de plateformes. Cela implique des contraintes *a priori* insoupçonnées par la plupart des développeurs, pour qui un pointeur est une adresse typée, sans plus. Parmi les questions à considérer, on trouve celles-ci :

- le concept de pointeur est-il le même pour un pointeur sur du code (par exemple un pointeur sur une fonction) et pour un pointeur sur une donnée? Cela en surprendra plusieurs mais il se trouve que la réponse varie d'un système à l'autre;
- quel est l'impact de développer sur un système où l'adressage se fait sur `16` bits, sur `32` bits ou sur `64` bits?
- peut-on faire en sorte qu'une classe donnée soit construite dans une zone précise de la mémoire, par exemple pour représenter directement une entité matérielle? *etc*.

Le modèle `.NET` est hybride au sens de la gestion de la mémoire : certains objets, les `struct`, sont alloués sur la pile et éliminés de manière déterministe, alors que d'autres, les `class`, sont alloués sur le tas, par un mécanisme d'allocation dynamique, et collectés de manière automatique et indéterministe par un moteur de collecte d'ordures.

Les moteurs de collecte d'ordures de Java et de `.NET` sont très sophistiqués et offrent, dans certaines circonstances, des performances supérieures à celles de programmes où la mémoire est gérée manuellement, comme le sont la plupart des programmes C et C++.

Ce niveau de performance est atteint par des économies d'échelle : des algorithmes peuvent être mis au point de manière à s'appliquer sur une masse d'objets à collecter, reposant sur un savoir plus global et offrent de meilleures performances d'ensemble que des algorithmes travaillant à la pièce sur de petits tronçons de mémoire.

Conséquence directe d'un recours continuel à l'allocation dynamique de mémoire, les langages reposant sur ces plateformes fragmentent beaucoup la mémoire. Conséquemment, la collecte d'ordures doit périodiquement y défragmenter la mémoire disponible, ce qui implique de déplacer les références pour compacter les blocs utilisés.

Toutes les références y sont donc à au moins deux niveaux; la collecte d'ordures doit être capable d'arrêter un *thread*, de relocaliser les objets et de mettre à jour les références que possède ce *thread* sur les objets en question.

[47] Sous C++ `11`, un moteur de collecte d'ordures ne réalisant pas de finalisation devait être proposé sur une base volontaire, mais un programme bien écrit ne devrait pas en avoir besoin.

Ce qui crée un objet

Le standard de C++ définit un objet de la manière suivante (§1.8) :

> « *An object is a region of storage*. »

Ceci peut surprendre; on ne parle pas ici de classes ou de grands principes, mais bien d'un lieu adressable et ayant, à un certain point dans l'exécution d'un programme, pris un sens. Le passage complet est :

> « *An object is a region of storage. [Note: A function is not an object, regardless of whether or not it occupies storage in the way that objects do. —end note] An object is created by a definition (3.1), by a new-expression (5.3.4) or by the implementation (12.2) when needed. The properties of an object are determined when the object is created. An object can have a name (Clause 3). An object has a storage duration (3.7) which influences its lifetime (3.8). An object has a type (3.9). The term object type refers to the type with which the object is created. Some objects are polymorphic (10.3); the implementation generates information associated with each such object that makes it possible to determine that object's type during program execution. For other objects, the interpretation of the values found therein is determined by the type of the expressions (Clause 5) used to access them* »

De nombreuses considérations techniques suivent ce passage et précisent l'intention derrière le texte, évidemment.

Le standard décrit la création d'un objet comme suit (§3.8) :

> « *The lifetime of an object is a runtime property of the object. An object is said to have non-vacuous initialization if it is of a class or aggregate type and it or one of its members is initialized by a constructor other than a trivial default constructor. [Note: initialization by a trivial copy/move constructor is nonvacuous initialization. —end note] The lifetime of an object of type T begins when:*
>
> *(1.1) — storage with the proper alignment and size for type T is obtained, and*
>
> *(1.2) — if the object has non-vacuous initialization, its initialization is complete.* »

Cela signifie essentiellement que dans le code suivant :

```
X x;
X *p0 = new X;
char buf[sizeof(X)];
X *p1 = reinterpret_cast<X*>(&buf[0]);
X *p2 = new (static_cast<void*>(&buf[0])) X;
```

… `x` est un objet. De même, `p0` et `p2` pointent tous deux sur des objets (leur pointé a été initialisé par un constructeur). Toutefois, `p1` ne pointe pas sur un objet; conséquemment, l'utiliser comme tel mènerait à un comportement indéfini.

Le standard décrit la « mort » d'un objet comme suit (§3.8) :

> « *The lifetime of an object of type T ends when:*
>
> *(1.3) — if T is a class type with a non-trivial destructor (12.4), the destructor call starts, or*
>
> *(1.4) — the storage which the object occupies is reused or released.* »

Essentiellement, cela explique les commentaires dans le code suivant :

```
{
   X x;
   X *p0 = new X;
   char buf[sizeof(X)];
   X *p2 = new (static_cast<void*>(&buf[0])) X;
   delete p0; // fin de la vie de *p0
   X *p3 = new (static_cast<void*>(&buf[0])) X; // on réutilise la mémoire associée à
                                                // *p2; utiliser p2 devient un problème
} // fin de la vie de x
```

Comprendre les mécanismes de construction et de destruction

La construction d'un objet se fait en plusieurs étapes. Nous décrirons ce mécanisme sous la lueur de l'allocation dynamique de mémoire, mais la situation est, au sens de l'ordre des opérations, semblable dans la situation où l'objet est créé et supprimé automatiquement[48].

Prenons à titre d'exemple le programme à droite. L'opération `new X` peut être découpée comme suit :

- allouer un bloc de mémoire de `sizeof(X)` *bytes*. Ceci explique que la taille des instances d'une classe doive être connue avant que cette classe puisse être instanciée;
- appeler le constructeur approprié de `X` (ici : `X::X()`), pour que ce sous-programme initialise le bloc de mémoire en question; puis
- retourner l'adresse où débute ce bloc de mémoire, convertie en pointeur de `X`. En effet, *a priori*, une adresse n'est qu'un lieu en mémoire, sans autre sémantique associée;
- l'invocation de `delete` sera implicitement précédé d'une invocation de `X::~X()` sur le pointeur, puis l'opérateur `delete` libérera la mémoire associée à l'adresse sur laquelle il aura été appliqué.

```
class X {
   // ...
};
void f() {
   auto p = new X;
   // ...
   delete p;
}
```

Examinons maintenant l'allocation dynamique d'un tableau de `X`, à droite. La même séquence s'opère (à un détail près) mais la taille calculée lors de l'allocation de la mémoire sera `n*sizeof(X)`.

Comme dans le cas où un seul `X` aurait été alloué, un bloc de mémoire brute de la taille demandée sera obtenu, suite à quoi `X::X()` y sera appliqué `n` fois, donc une fois par objet en devenir dans le tableau.

L'opérateur `delete[]` invoquera `n` fois `X::~X()` (une fois par objet en cours de disparition) puis libérera l'adresse `p`.

```
// ...
void f(int n) {
   auto p = new X[n];
   // ...
   delete[] p;
}
```

On constate donc que le constructeur a pour rôle d'initialiser un bloc de mémoire préalablement alloué, pas d'obtenir le bloc de mémoire en premier lieu. L'inverse est aussi vrai : un destructeur a pour rôle de libérer les ressources que s'est approprié l'objet en cours de destruction, mais ne libère pas vraiment le bloc de mémoire dans lequel se situe l'objet.

Le constructeur initialise, le destructeur nettoie (ou finalise). De leur côté, `new` (tout comme `new[]`) et `delete` (tout comme `delete[]`) gèrent l'allocation et la libération de blocs de mémoire brute mais ne construisent, ni ne détruisent, les objets.

[48] J'entends par cela tout objet placé sur la pile, donc dont l'existence est délimitée par la portée du bloc où il aura été déclaré, ou tout objet qualifié `static`, qui sera créé avant le début de l'exécution du programme principal et détruit après la fin de cette exécution.

Exemple simple

Examinons tout d'abord un exemple simple des opérateurs `new` et `delete` (de même que `new[]` et `delete[]`) sur des versions spécialisées d'une hypothétique classe `X`. Cet exemple se veut illustratif; nous reviendrons sur la syntaxe et les nuances un peu plus loin.

Cet exemple ne fera qu'afficher à l'écran une trace des allocations et des libérations dynamiques de mémoire et des constructeurs et destructeurs invoqués.

La classe `X` à droite expose une version spécialisée des opérateurs `new`, `delete`, `new[]` et `delete[]`. Cette version n'est applicable qu'à elle et à ses descendants[49].

Dans chaque cas, la spécialisation de l'opérateur affiche à la console la valeur des paramètres qu'elle a reçu puis délègue le travail véritable à la version globale de l'opérateur correspondant (ceci explique la présence du préfixe `::` apposé au nom de chacun des opérateurs).

Le programme principal montre un exemple d'invocation de ces mécanismes[50]. Il est clair ici que `sizeof(X)==1`.

```
X::operator new(1)
X::X()
X::~X()
X::operator delete()
X::operator new[](7)
X::X()
X::X()
X::X()
X::~X()
X::~X()
X::~X()
X::operator delete[]()
```

```
#include <iostream>
// ... using ...
struct X {
   X() {
      cout << "X::X()" << endl;
   }
   ~X() {
     cout << "X::~X()" << endl;
   }
   void* operator new(size_t n) {
      cout << "X::operator new(" << n
           << ")" << endl;
      return ::operator new(n);
   }
   void* operator new[](size_t n) {
      cout << "X::operator new[](" << n
           << ")" << endl;
      return ::operator new[](n);
   }
   void operator delete(void *p) {
      cout << "X::operator delete()" << endl;
      ::operator delete(p);
   }
   void operator delete[](void *p) {
      cout << "X::operator delete[]()" << endl;
      ::operator delete[](p);
   }
};
int main() {
   auto x = new X;
   delete x;
   x = new X[3];
   delete[] x;
}
```

[49] Puisque `X` n'expose pas de méthodes polymorphiques, en dériver publiquement serait une erreur de design.

[50] L'affichage peut varier selon les compilateurs.

Déclinaisons de `new` et de `delete`

Il est possible en C++ de contrôler finement la mécanique d'allocation dynamique de mémoire, que ce soit pour un type donné ou pour tous les types. L'exemple simple proposé plus haut montre d'ailleurs une spécialisation pour une classe spécifique (la classe `X`) qui délègue une partie du travail vers une déclinaison globale des opérateurs correspondants.

Dans les sections qui suivent, nous examinerons les diverses déclinaisons des opérations permettant de contrôler cette mécanique. Par souci de simplicité, nous utiliserons `new` pour parler de `new` et de `new[]`, tout comme nous utiliserons `delete` pour parler à la fois de `delete` et de `delete[]`, à moins que nous ne souhaitons discuter spécifiquement de la version applicable à un tableau de ces opérateurs.

Débutons par quelques considérations d'ensemble :

- dans tous les cas, `delete 0;` et `delete nullptr;` sont des opérations légales en C++. Les versions spécialisées de `delete` doivent traiter ce cas sans planter;
- selon le standard `ISO` de C++, si l'opérateur `new` ne parvient pas à allouer la quantité de mémoire demandée, alors cet opérateur devrait lever une exception, typiquement `std::bad_alloc`[51].

> Avant l'avènement du standard `ISO` de C++, `new` retournait `0` lorsqu'ils ne parvenaient pas à accomplir leur tâche. Certaines implémentations peuvent ne pas être à jour et agir ainsi encore aujourd'hui, alors prudence.

- la déclaration `int tab[0];` est illégale en C++, puisqu'elle impliquerait allouer un espace nul sur la pile, mais l'opération `int *p= new int[0];` est légale et retourne l'équivalent, au sens des itérateurs standards, d'un itérateur de fin de séquence. Ceci permet d'avoir recours à peu près sans danger à la majorité des algorithmes standards avec un tableau dynamique de taille zéro.

L'un des principes de fond à intégrer immédiatement est que le rôle de `new` (et de `new[]`) n'est pas tant d'allouer de la mémoire que de trouver un endroit pour placer un objet. Conséquemment, la déclinaison la plus fondamentale de l'opérateur `new`, le *Placement New* (plus de détails dans la section ***Allocation positionnelle***), laisse le code client choisir l'emplacement qu'occupera un objet :

```
char buf[sizeof(X)]; // zone de mémoire brute, assez grande pour contenir un X
X *p = new (static_cast<void*>(&buf[0])) X; // le X est créé dans buf!
```

[51] Cette classe est définie dans le fichier d'en-tête standard `<new>`.

Spécialisation en tant que méthode

Comme le montrait l'exemple simple plus haut, il est possible de contrôler la mécanique d'allocation dynamique de mémoire pour une classe en particulier. On y arrive en spécifiant les opérateurs `new` et `delete` sur ce type.

L'opérateur `new` sera invoqué pour déléguer à l'objet la tâche de se réserver lui-même un bloc de mémoire, donc avant l'appel du constructeur. L'opérateur `delete` sera quant à lui invoqué pour procéder à la libération du bloc de mémoire dans lequel réside l'objet, donc après le destructeur.

Clairement, dans les deux cas, il serait malsain d'utiliser un membre d'instance pour mettre en place la mécanique d'attribution dynamique de la mémoire, car ces membres n'existent alors pas encore.

Il est légal en C++ de ne définir que l'opérateur `new` ou que l'opérateur `delete` sur un objet, par exemple pour insérer le code requis pour faire une trace de ces opérations.

Habituellement, toutefois, on voudra implémenter ces opérateurs par quartet, du fait qu'il soit probable qu'on veuille appliquer aux tableaux une stratégie semblable à celle appliquée aux scalaires.

En effet, s'il est décidé d'appliquer une stratégie spéciale d'allocation dynamique de mémoire pour un type donné, alors il est peu probable qu'on souhaite laisser la mécanique normale de C++ se charger de l'opération correspondante de libération dynamique de mémoire.

Quatre opérateurs[52] distincts peuvent donc être surchargés pour une classe donnée dans l'optique de contrôler la mécanique d'attribution dynamique de la mémoire. Les signatures de ces opérateurs sont celles exposées dans l'ébauche de classe `X` ci-dessous.

Les opérateurs `new` et `new[]` prennent en paramètre la taille du bloc à réserver. Leur rôle est donc d'obtenir une zone mémoire convenable pour un `X` ou pour une séquence de `X` puis, une fois la zone mémoire obtenue, de retourner un pointeur abstrait (un `void*`) vers le début de cette zone.

Cette zone sera par la suite initialisée par des constructeurs qui ne sont connus que du moteur de C++. Les opérateurs `new` et `new[]` ne savent pas (sauf dans quelques rares exceptions[53]) lequel des constructeurs de `X` a été invoqué.

```
class X {
   // ...
public:
   // ...
   void* operator new(size_t);
   void operator delete(void *p);
   void* operator new[](size_t);
   void operator delete[](void *p);
   // ...
};
```

Les opérateurs `delete` et `delete[]` prennent quant à eux un pointeur abstrait, qu'on présume vers un `X` et préalablement attribué à l'aide de la méthode `new()` ou `new[]()` de `X`, et libèrent la mémoire correspondante. Ils ne se préoccupent pas d'invoquer un destructeur, puisque celui-ci a déjà été invoqué à ce stade.

[52] En pratique, c'est plutôt huit opérateurs qu'il faudra surcharger, parfois plus (mais toujours un multiple de quatre). Nous y reviendrons plus loin.

[53] L'exception survient bien sûr quand `X` n'expose qu'un seul constructeur.

Exemple : une armée d'orques

Imaginons que nous œuvrions à un jeu vidéo de type médiéval fantastique et que nous sachions, de par le scénario du jeu, que nous aurons à gérer, fréquemment, des armées d'orques, ou *hordes*. Imaginons aussi que nous pensions être capables de profiter du fait que la taille d'un `Orque` est connue dès la compilation… Après tout, toute instance de la classe `Orque` occupera `sizeof(Orque)` *bytes* en mémoire!

Mise en garde importante : bâtir une stratégie d'allocation de mémoire qui soit optimisée en fonction de la taille des objets à allouer est risqué si la classe visée est susceptible d'avoir des dérivés, puisque l'ajout d'attributs dans les enfants aura (évidemment) un impact sur leur taille. Pour une classe terminale, ou dont les enfants auront nécessairement la même taille qu'elle, cette approche peut être raisonnable.

Nous mettrons ici en place un programme de test qui créera plusieurs instances de la classe `Orque`, les utilisera (de manière fictive) puis les détruira à la pièce. Nous utiliserons aussi une allocation en bloc d'un tableau d'instances de `Orque` (des renforts pour ceux déjà créés!) pour vérifier la qualité de l'implémentation de l'opérateur `new[]`.

En contrepartie, nous utiliserons l'opérateur `delete[]` pour nettoyer les renforts, puis nous appellerons massivement l'opérateur `delete` pour éliminer les instances de `Orque` créées à la pièce en début de programme.

Une fois des mesures faites pour les versions standards des opérateurs `new`, `new[]`, `delete` et `delete[]`, nous implémenterons notre propre petit mécanisme maison pour gérer la mémoire allouée dynamiquement et nous mesurerons le programme résultant. Ceci nous permettra de voir l'impact de nos choix d'implémentation.

Pour mesurer la vitesse des mécanismes de gestion dynamique de mémoire dans leurs déclinaisons par défaut et maison, nous utiliserons la fonction standard `system_clock::now()` de la bibliothèque `<chrono>` qui retourne une unité de mesure propre à l'horloge système.

```
#include <chrono>
using namespace std::chrono;
template <class F>
   auto tester(F f) {
      auto pre = system_clock::now();
      f();
      auto post = system_clock::now();
      return post - pre;
   }
```

La section de code dont on veut mesurer la vitesse d'exécution doit être entourée une fonction, ou simplement une λ englobant le code à tester.

Le code client est alors responsable de l'affichage et du choix de l'unité de mesure.

```
// ...
int main() {
   auto dt = tester([] { f(); });
   cout << "Temps ecoule pour f(): "
        << duration_cast<
              milliseconds
           >(dt).count()
        << " ms." << endl;{
}
```

Décrire un Orque

Décrire un `Orque` est une chose simple en soi. Nous pourrions insérer un nom (typiquement, une instance de `string`), des points de vie et quelques attributs décoratifs et nous en tenir là.

Par principe, je vous proposerai ici une classe `Orque` *légèrement* plus complète et plus sérieuse, mais retenez que, pour nos fins, une version naïve aurait suffi. Après tout, nous sommes intéressé(e)s par les mécanismes d'allocation dynamique de mémoire, pas par la représentation correcte d'une créature hideuse et agressive.

Tout `Orque` sera `Frappable`. Nous entendrons par-là que chaque `Orque` sera voué à une existence violente et sera une créature de conflits, sujette à blesser et à être blessée.

Dans la plupart des sagas médiévales fantastiques, un `Orque` est la créature vilaine standard et tend à faire preuve d'une personnalité que je qualifierais de quelque peu belliqueuse.

Pour faciliter les conflits, nous dirons ici que les créatures sujettes à se battre seront des dérivés de `Frappable`.

```
#include <random>
class Frappable {
protected:
   // discutable...
   template <class T>
      static gen_val(T minv, T maxv) noexcept {
      using namespace std;
      static random_device rd;
      static mt19937 prng { rd() };
      uniform_int_distribution<T> de{ min, maxv };
      return de(prng);
   }
public:
   using vie_t = short;
   vie_t vie_ = generer_vie();
private:
   static vie_t generer_vie() noexcept {
      return gen_val(vie_t{5}, vie_t{20});
   }
public:
   Frappable() = default;
   Frappable(vie_t vie) noexcept : vie_{vie} {
   }
   virtual ~Frappable() = default;
   vie_type vie() const noexcept {
      return vie_;
   }
   bool est_decede() const noexcept {
      return vie() <= 0;
   }
   void endommager(const vie_type degats) noexcept {
      if (degats > 0) vie_ -= degats;
   }
   void guerir(const vie_type soins) noexcept {
      if (soins > 0) vie_ += soins;
   }
   virtual void frapper(Frappable &) = 0;
};
```

Les diverses instances de la classe `Orque` auront un certain seuil de méchanceté.

Ce seuil influencera les dégâts faits au `Frappable` avec lequel un `Orque` est en conflit lorsque l'instance d'`Orque` en question lui portera un coup. Sachant que chaque `Orque` possède un attribut de type `string`, on peut dire que `sizeof(Orque)` sera au moins aussi grand que

```
sizeof(Frappable)+sizeof(short)+sizeof(string)
```

Ceci nous donne une classe `Orque` qui a au moins un minimum d'intérêt et dont chaque instance, en pratique, occupe probablement entre `30` et `40` *bytes* en mémoire, selon la plateforme et le compilateur.

```
class Orque : public Frappable {
   enum { MECHANCETE_DEFAUT = 5 };
   string nom_ = "Euh...";
   short mechancete_ = MECHANCETE_DEFAUT;
   vie_t generer_degats() const noexcept {
      return gen_val(
         vie_t{0}, vie_t{ mechancete_}
      );
   }
public:
   Orque(const string &nom) : nom_{nom} {
   }
   void frapper(Frappable &f) {
      f.endommager(generer_degats());
   }
};
```

Il nous faudra maintenant un programme de test pour vérifier la vitesse à laquelle s'exécuteront les opérateurs d'allocation et de libération dynamique de mémoire dans leur déclinaison standard comme dans leur déclinaison maison. Parmi les bibliothèques auxquelles nous aurons recours, remarquez la bibliothèque `<list>`.

Sans que ce ne soit strictement nécessaire avec le programme de test qui sera proposé ici, une masse d'orques en situation de conflit risquerait de générer des décès inopinés ici et là et des suppressions ailleurs qu'à la fin d'un conteneur sont beaucoup plus rapides sur une `std::list` que sur un `std::vector`.

```
#include "Orque.h"
#include "MesureDeTemps.h"
#include <iostream>
#include <list>
#include <string>
#include <random>
#include <algorithm>
// ...using...
```

Nous mettrons en scène une bande de `NORQUES_TOTAL` instances d'`Orque`, réparties de manière aléatoire entre deux hordes (une néfaste et l'autre horrible).

```
int main() {
   const int NORQUES_TOTAL = 100'000;
   list<Orque*> HordeNefaste;
   list<Orque*> HordeHorrible;
   random_device rd;
   mt19937 prng { rd() };
   uniform_int_distribution<int> cenne(0,1);
```

La création des hordes exploite massivement `new` : elle crée chaque `Orque` et le positionne dans l'une ou l'autre des hordes avec une probabilité de `0,5`.

Les membres de la horde néfaste se nomment `Urg` suivi d'un numéro de série alors que les membres de la horde horrible se nomment `Arg` suivi d'un numéro de série.

Que voulez-vous : les orques sont comme ça.

```
    auto dt = tester([&] {
       for (int i = 0; i < NORQUES_TOTAL; ++i)
          if (cenne(prng) == 0)
             HordeNefaste.push_back
                (new Orque{"Urg "+to_string(i)});
          else
             HordeHorrible.push_back
                (new Orque{"Arg "+to_string(i)});
    });
    cout << "Création des hordes : "
         << duration_cast<milliseconds>(dt).count()
         << " ms."
         << "\nTaille de HordeNefaste : "
         << HordeNefaste.size()
         << "\nTaille de HordeHorrible : "
         << HordeHorrible.size() << endl;
```

Un combat s'ensuit. J'ai censuré ce tronçon pour les âmes sensibles, mais sachez qu'aucun `Orque` ne sera retiré de son conteneur pendant le combat (pour éviter de fausser les données lorsque nous mesurerons la performance de `delete`).

Arrivent soudainement des renforts potentiels (non alignés).

Ceci nous permet de tester et de mesurer l'opérateur `new[]`.

Un malheur survient et les renforts potentiels décèdent subitement d'un mal inconnu. Ceci nous permet évidemment de tester et de mesurer l'opérateur `delete[]`).

Vient enfin l'étape du nettoyage du champ de bataille, qui exploite massivement `delete` et nous permet de mesurer la vitesse d'exécution de cet opérateur.

Tester le programme dans sa version actuelle permettra de connaître la vitesse d'exécution des quatre opérateurs standards de gestion de la mémoire allouée dynamiquement. Avec ces résultats, nous pourrons voir si nos propres opérateurs se comportent à la hauteur des attentes.

```
    Orque *pMoton = nullptr;
    dt = tester([&] {
       pMoton = new Orque[1000];
    });
    cout << "Création d'un moton d'Orques : "
         <<  duration_cast<milliseconds>(dt).count()
         << " ms.\n";
    auto dt2 = tester([&] {
       delete[] pMoton;
    });
    cout << "Nettoyage d'un moton d'Orques : "
         <<  duration_cast<milliseconds>(dt).count()
         << " ms.\n";


    tester([&]{
       for(auto & p : HordeHorrible)
          delete p;
       for(auto & p : HordeNefaste)
          delete p;
    });
    cout << "Nettoyage : "
         << duration_cast<milliseconds>(dt).count()
         << " ms.\n";
}
```

Nous allons maintenant implémenter des versions naïves (j'insiste!) des opérateurs de gestion de la mémoire allouée dynamiquement pour la classe `Orque`. Étonnamment, nous obtiendrons des performances très similaires à celles obtenues par les opérateurs standards, ce qui suggère que nous pourrions faire beaucoup mieux si nous y investissions un peu plus d'énergie.

Exposer `new` et `delete` sous forme de méthodes d'instance dans une classe donnée est chose banale, comme le montre l'exemple à droite.

Les signatures sont en effet les mêmes que dans le cas des versions globales de ces opérateurs. Cependant, pour implémenter ces opérateurs, notre classe `Orque` a l'avantage de connaître la taille de ce qui sera alloué. Présumant que la classe `Orque` soit terminale, la taille des blocs à allouer sera connue *a priori* et sera nécessairement un multiple de `sizeof(Orque)`.

```
class Orque : public Frappable {
   enum { MECHANCETE_DEFAUT = 5 };
   string nom_ = "Euh...";
   short mechancete_ = MECHANCETE_DEFAUT;
   vie_t generer_degats() const noexcept {
      return gen_val(
         vie_t{0}, vie_t{ mechancete_}
      );
   }
public:
   Orque() = default;
   Orque(const string &nom) : nom_{nom} {
   }
   void frapper(Frappable &f) {
      f.endommager(generer_degats());
   }
   void* operator new(size_t);
   void* operator new[](size_t);
   void operator delete(void *);
   void operator delete[](void *);
};
```

L'implémentation de ces méthodes sera, elle aussi, relativement simple. Cette simplicité cache toutefois un effort de programmation un peu plus grand.

Tout d'abord, remarquons que toutes les stratégies d'allocation de mémoire reposant sur des blocs de taille fixe ont la particularité de reposer sur une mémoire à répartir dont la taille sera un multiple de la taille d'un seul bloc.

Nous utiliserons donc une classe générique spécialisée en ce sens, que nous nommerons `ArenaFixe`, examinée plus bas, et nous ferons en sorte que la classe `Orque` pige sa mémoire dans une `ArenaOrque`, qui sera un singleton spécialisant `ArenaFixe` pour des blocs de taille `sizeof(Orque)`.

Les méthodes d'instance de la classe `Orque` pour allouer et libérer de la mémoire délégueront ces tâches au singleton `ArenaOrque`.

Les méthodes de `ArenaOrque` servant à allouer un ou plusieurs orques se nommeront `allouer_un()` et `Allouer()`, alors que les méthodes pour libérer la mémoire pour un ou plusieurs orques se nommeront quant à elles `liberer_un()` et `liberer()`.

```
#include "Orque.h"
#include "ArenaFixe.h"
namespace {
   enum { POPULATION_MAX = 200'000 };
}
class ArenaOrque
   : public ArenaFixe<
        sizeof(Orque), ::POPULATION_MAX
     > {
   static ArenaOrque singleton;
   ArenaOrque() = default;
public:
   ArenaOrque(const ArenaOrque&) = delete;
   ArenaOrque&
      operator=(const ArenaOrque&) = delete;
   static ArenaOrque& get() noexcept {
      return singleton;
   }
};
ArenaOrque ArenaOrque::singleton;
void* Orque::operator new(size_t) {
   return ArenaOrque::get().allouer_un();
}
void* Orque::operator new[](size_t n) {
   return ArenaOrque::get().allouer
                          (n/sizeof(Orque));
}
void Orque::operator delete(void *p) {
   ArenaOrque::get().liberer_un(p);
}
void Orque::operator delete[](void *p) {
   ArenaOrque::get().liberer(p);
}
```

La grande (et un peu plus complexe) question est donc : *comment pourrait-on implémenter `ArenaFixe`?* Une réponse simpliste mais opérationnelle suit.

Implémenter `ArenaFixe` : une approche possible

Nous mettrons en place une classe générique `ArenaFixe` basée sur deux paramètres :

- la taille d'un individu (d'un bloc de mémoire); et
- la population attendue (le nombre maximum de blocs prévus par le code client).

Cette stratégie est naïve pour plusieurs raisons, mais en partie du fait que nous ne mettrons aucun mécanisme en place pour réagir si la population réelle dépasse les prévisions du code client. Notre `ArenaFixe` manquera simplement de mémoire et lèvera un `std::bad_alloc` si cela se produit.

Dans une instance de la classe `ArenaFixe` proposée ici, on trouvera un bassin de mémoire disponible (bêtement : un bloc de *bytes* dont la taille est le produit de la population par la taille d'un individu). Nos blocs ne seront donc pas nécessairement alignés sur la taille d'un mot mémoire, ce qui peut ralentir l'utilisation que le code client fera d'eux.

Sur certaines plateformes, il est possible que l'allocation doive retourner des adresses alignées sur les mots mémoire. Assurez-vous de consulter la documentation de votre plateforme cible si vous implémentez des mécanismes aussi fondamentaux que les opérateurs de gestion dynamique de la mémoire.

Pour simplifier la recherche de blocs alloués, nous utiliserons une séquence de bits (un `std::bitset`) qui contiendra, de manière compacte, un bit par individu. Nous considérerons qu'un bit à zéro signifie que le bloc correspondant dans le bassin est libre.

Pour accélérer l'allocation de blocs en moyenne, nous conserverons la position du dernier bloc alloué dans un attribut, nommé `plus_recent_`). Cette stratégie n'est pas sans faille (loin de là!) mais tend à accélérer l'allocation de mémoire en comparaison avec une stratégie qui chercherait toujours des blocs libres à partir du tout premier bloc du bassin.

Une stratégie qui chercherait à partir du plus récent bloc libéré serait probablement encore plus efficace, avec une complexité $O(1)$ pour allouer un seul bloc).

La recherche d'un bloc libre se fera de manière cyclique à partir du plus récent bloc alloué. L'opération `prochain()` facilitera le passage d'un bloc à l'autre.

Du point de vue des inclusions, des attributs et des types internes et publics, donc, le portrait de cette stratégie simpliste ressemblerait à ce qui est proposé à droite.

Le raison du recours à une `ArenaFixe` générique est (en partie) de permettre de déclarer un bassin de mémoire sans avoir recours à de l'allocation dynamique de mémoire. Si la taille du bassin est grande, cette stratégie devient moins sage et, que `ArenaFixe` soit générique ou non, il devient alors avantageux d'allouer le bassin dynamiquement.

L'attribut `tailles` sert à retenir la taille des tableaux, au besoin, dans le but de simplifier la libération de gros blocs.

```
#ifndef ARENA_FIXE_H
#define ARENA_FIXE_H
#include <bitset>
#include <new>
// ...using...
template <size_t TailleBloc, size_t NBlocs>
   class ArenaFixe {
   public:
      using size_type = size_t;
   private:
      char bassin[TailleBloc * NBlocs];
      bitset<NBlocs> pris;
      size_type tailles[NBlocs];
      size_type plus_recent {};
```

Un `std::bitset` représente de manière compacte un nombre de bits fixé à la déclaration. Ici, le nombre de bits à représenter correspond, tel qu'annoncé, au nombre de blocs dans le bassin de mémoire disponible.

La méthode `prochain()` retourne la prochaine position dans le bassin à partir d'un compteur. C'est une fonction utilitaire privée dont le principal rôle est d'alléger le code.

```
      static size_type
         prochain(size_type n) noexcept {
            return (n + 1) % NBlocs;
         }
```

La méthode `compter_disponibles()` compte le nombre de blocs libres à partir de `Base` jusqu'à concurrence de `max_pas` blocs. Elle sert à trouver l'espace contigu nécessaire à l'allocation d'un tableau de `max_pas` blocs.

Elle retourne le nombre de blocs consécutifs disponibles dans le but d'accélérer la recherche (le code appelant sait qu'il peut escamoter ce nombre de blocs sans risque).

```
      size_t compter_disponibles
         (size_type base, size_type max_pas) {
         if (base + max_pas >= NBlocs)
            throw bad_alloc{};
         size_type n = max_pas;
         while (!pris.test(nase) && n) {
            base = prochain(base);
            --n;
         }
         return max_pas - n;
      }
```

Remarquez le caractère potentiellement naïf de cette stratégie : si la recherche commence avec une `Base` trop haute, il est possible que `compter_disponibles()` approche de la fin du bassin et lève un `std::bad_alloc` alors que le début du bassin contient peut-être assez de blocs disponibles pour servir une requête d'allocation.

La méthode `allouer_un()` cherche et retourne un bloc de mémoire brute pris dans le bassin.

Tel que mentionné plus haut, la recherche de blocs disponibles commence juste après le plus récent bloc alloué et navigue à travers le bassin de manière circulaire jusqu'à ce qu'un bloc soit trouvé, levant un `std::bad_alloc` le cas échéant.

Le `std::bitset` permet de vérifier si un bloc est pris (méthode `test()`) et d'indiquer qu'un bloc a été pris (méthode `set()`).

La méthode `allouer()` alloue un séquence de `n` blocs consécutifs en mémoire dans le bassin (ce qui doit exclure une séquence circulaire qui commencerait à la fin du bassin et se terminerait au début). La méthode `compter_disponibles()` permet à la fois une recherche de blocs consécutifs et une progression rapide quand le nombre de blocs trouvé est insuffisant.

Notez que la boucle cherchant une séquence de blocs disponibles pourrait être raffinée (la version proposée ici cherche systématiquement à partir du début) mais ce raffinement est plus subtil à implémenter qu'il n'y paraît.

```
public:
   void* allouer_un() {
      auto i = prochain(plus_recent);
      while (pris.test(i) && i != plus_recent)
         i = prochain(i);
      if (i == plus_recent)
         throw bad_alloc{};
      pris.set(i);
      plus_recent = i;
      return bassin + i * TailleBloc;
   }



   void* allouer(size_type n) {
      size_type i = 0;
      auto ndispo = compter_disponibles(i, n);
      while (ndispo < n) {
         i += ndispo + 1;
         ndispo = compter_disponibles(i, n);
      }
      for (size_type j = i, cpt = 0;
           cpt < n; j = prochain(j), ++cpt)
         pris_.set(j);
      tailles[i] = n;
      plus_recent = i;
      return bassin + i * TailleBloc;
   }
```

L'une des difficultés de l'implémentation de stratégies d'allocation dynamique de mémoire est qu'elles doivent être à la fois sans failles et très efficaces. Un raffinement à la stratégie ci-dessus qui ralentirait un peu la recherche de blocs pourrait avoir un effet catastrophique sur la performance d'ensemble d'un programme.

Libérer un bloc est une chose relativement simple. En effet, à partir de l'adresse à libérer, on n'a qu'à retrouver l'indice du bloc dans le bassin et à modifier le bit indiquant si ce bloc est pris ou non.

```
   void liberer_un(void *p) {
      const size_type pos =
         (static_cast<char *>(p) - bassin)/
          TailleBloc;
      pris_.set(pos, false);
   }
```

Libérer une séquence de blocs implique de retrouver le bloc du début, le nombre de blocs à libérer et à modifier les différents bits indiquant si ces blocs sont pris ou non.

Remarquez que `std::bitset` n'est pas un conteneur standard et n'offre (tristement) pas d'itérateurs.

```
void liberer(void *p) {
   const size_type pos =
      (static_cast<char *>(p) - bassin)/
       TailleBloc;
   const size_type FIN = pos + tailles[pos];
   for (size_type j = pos; j < FIN; ++j)
      pris.set(j, false);
}
```

Enfin, le constructeur et le destructeur sont des plus banals. Une version plus gentille d'`ArenaFixe` utiliserait le constructeur pour allouer le bassin (avec la fonction `std::malloc()`) et le destructeur pour le libérer (avec `std::free()`), ce qui réduirait la taille du programme une fois celui-ci compilé.

```
      ArenaFixe() = default;
   };
#endif
```

Spécialisation en tant que fonction globale

Si l'objectif visé et de contrôler la mécanique d'allocation dynamique pour tous les objets, pas seulement pour les instances de classes ciblées, alors il est possible de surcharger les opérateurs `new` et `delete` implémentés sous forme de fonctions globales.

Les signatures sont celles proposées à droite. Le moteur de C++ utilisera cette version des opérateurs globaux dans la mesure où ils ont été définis, bien sûr, et dans la mesure où les classes sur lesquelles sont appliqués ces opérateurs n'en ont pas leur propre version sous forme de méthode.

```
void* operator new(size_t);
void* operator new[](size_t);
void operator delete(void*);
void operator delete[](void*);
```

Choisir de prendre en charge la totalité de la mécanique d'allocation dynamique de mémoire est une décision lourde de conséquences. On n'agira ainsi que si le besoin s'en fait réellement sentir, comme par exemple si une stratégie spéciale d'allocation dynamique de mémoire doit être appliquée systématiquement, peut-être à cause de contraintes de performance.

Les versions globales des opérateurs `new` et `delete` risquent d'être invoquées tôt dans la séquence de création des divers objets globaux, ce qui implique qu'il ne faut pas compter sur autre chose que des mécanismes très primitifs dans ces fonctions (p. ex. : n'invoquez pas les services du flux standard `std::cout` dans `operator new()` car cet objet n'a peut-être pas encore été créé).

Exemple : comptabiliser la mémoire allouée dynamiquement

Dans le but d'apprendre comment surcharger les opérateurs d'allocation et de libération dynamique de mémoire que sont `new`, `new[]`, `delete` et `delete[]`, une application pratique est de mettre au point une stratégie de comptabilité de la mémoire allouée et de la mémoire libérée. En ayant accès à ces données, il devient possible de détecter certains types de fuite de ressources.

Pour faciliter la comptabilité des ressources, j'utiliserai un singleton [POOv02], qui sera un cas particulier d'incopiable (pour alléger la syntaxe).

Le singleton en soi se nommera `ComptableMemoire` et se déclinera comme suit :

```
#ifndef ALLOCATION_H
#define ALLOCATION_H
class ComptableMemoire {
public:
   using size_type = long;
   ComptableMemoire(const ComptableMemoire&) = delete;
   ComptableMemoire& operator=(const ComptableMemoire&) = delete;
private:
   static ComptableMemoire singleton;
   size_type qte {};
   ComptableMemoire() = default;
   void fuite_identifiee();
   void surliberation_identifiee();
public:
   void allocation(size_t n) {
      qte += static_cast<size_type>(n);
   }
   void deallocation(size_t n) {
      qte -= static_cast<size_type>(n);
      if (quantite() < 0)
         surliberation_identifiee();
   }
   size_type quantite() const noexcept {
      return qte;
   }
   static ComptableMemoire& get() noexcept {
      return singleton;
   }
   ~ComptableMemoire() {
      if (quantite()) fuite_identifiee();
   }
   class TraceurLocalSimplet {
      size_type avant;
   public:
      TraceurLocalSimplet() : avant{ComptableMemoire::get().quantite()} {
      }
      ~TraceurLocalSimplet();
   };
};
#endif
```

Les méthodes clés de `ComptableMemoire` sont `allocation()`, `deallocation()` et `quantite()`. Retenons que `ComptableMemoire` comptabilisera la mémoire allouée mais n'allouera rien par elle-même.

Notez que certains blocs ne sont pas supposés être balancés; un `TraceurLocalSimplet` est un outil `RAII` dont on se servira pour tester l'hypothèse qu'on bloc a une stratégie équilibrée d'allocation dynamique de mémoire, tout simplement.

Remarquez la classe locale `TraceurLocalSimplet`. Cette classe aura un rôle tout simple :

- son constructeur saisira l'état courant de l'allocation dynamique de mémoire; et
- son destructeur fera de même, en plus d'émettre un message sur l'erreur standard dans l'optique où les valeurs obtenues à la construction et à la destruction sont différentes.

Ceci permettra de vérifier si, dans un bloc de code donné, les opérations d'allocation et de libération dynamique de mémoire sont balancées.

Remarquez aussi que le destructeur de `ComptableMemoire` invoque une méthode nommée `fuite_identifiee()` dans son destructeur dans la mesure où une fuite de mémoire a été détectée. Cette stratégie peut être risquée : un `ComptableMemoire` est un singleton statique et tout objet dont il pourrait se servir pour souligner la fuite détectée (incluant les flux de sortie et d'erreur standard) est susceptible d'avoir été détruit au moment de la destruction du `ComptableMemoire` en tant que tel.

Nous verrons plus tard une stratégie plus stable et moins dangereuse de signaler les fuites (à court terme, déclarer un `TraceurLocalSimplet` est beaucoup mieux).

En ajoutant les prototypes des opérateurs d'allocation et de libération dynamique de mémoires dans leur déclinaison globale, on obtient le fichier `Allocation.h` suivant :

```
#ifndef ALLOCATION_H
#define ALLOCATION_H
//
// Surcharge des opérateurs globaux; comptabilise pour faciliter le traçage des fuites.
//
void* operator new(size_t);
void* operator new[](size_t);
void operator delete(void*);
void operator delete[](void*);
class ComptableMemoire {
   // ... voir plus haut ...
};
#endif
```

Le fichier source `Allocation.cpp` accompagnant ces déclarations se détaille quant à lui comme suit :

```
#include "Allocation.h"
#include <cstdlib>
#include <new>
#include <iostream>
// ...using...
void* operator new(size_t qte) {
   //
   // Alloue un size_t supplémentaire à chaque new (très coûteux, mais simple). L'adresse
   // retournée est sur l'espace attribué pour les données; la taille attribuée se trouve
   // juste avant les données en mémoire. Le size_t supplémentaire n'est pas comptabilisé
   // par la mécanique de détection des fuites (il s'agit d'un détail d'implémentation)
   //
   auto p = malloc(qte + sizeof(size_t));
   if (!p) throw bad_alloc{};
   ComptableMemoire::get().allocation(qte);
   auto q = new (static_cast<void*>(p)) size_t{ qte };
   return q + 1;
}
void* operator new[](size_t qte) {
   // identique à operator new()
}
void operator delete(void *p) {
   auto q = static_cast<size_t *>(p) - 1;
   ComptableMemoire::get().deallocation(*q);
   free(q);
}
void operator delete[](void *p) {
   // identique à operator delete ()
}
void ComptableMemoire::fuite_identifiee() {
   cerr << "Fuite (" << quantite () << " bytes non libérés)\n";
}
void ComptableMemoire::surliberation_identifiee() {
   cerr << "Libération abusive de mémoire\n";
}
ComptableMemoire ComptableMemoire::singleton;
ComptableMemoire::TraceurLocalSimplet::~TraceurLocalSimplet() noexcept {
   auto diff = ComptableMemoire::get().quantite() - avant;
   if (diff) cerr << "Fuite locale (" << diff << " bytes)" << endl;
}
```

Remarquez principalement que `new` et `new[]` sont identiques et que `delete` et `delete[]` le sont aussi. Ce n'est pas toujours le cas; il s'avère simplement que, pour ce schème très simple d'allocation et de libération de mémoire, il n'est pas nécessaire d'être plus sophistiqué que cela.

L'essentiel du travail fait par `new/ new[]` et `delete/ delete[]` ici est :

- dans le cas de `new/ new[]`, allouer (avec la fonction C `std::malloc()`) un bloc de la taille demandée plus un petit espace pour conserver la taille demandée (pour faciliter la comptabilité), déposer la taille en question au début du bloc et retourner l'adresse juste après le point où la taille se termine; et
- dans le cas de `delete/ delete[]`, retrouver la taille qui se situe juste avant le pointeur reçu en paramètre et libérer la mémoire réellement allouée (qui commence là où était entreposée la taille).

Ce qui complique un peu la sauce est l'incontournable ensemble de conversions explicites de types appliquées aux divers pointeurs manipulés ici (de `void*` à `size_t*` et inversement)

Un programme de test suit. Le petit bloc interne allouant de la mémoire à travers les pointeurs `temp` et `temp2` détectera une fuite de `sizeof(int)` *bytes* (`4` *bytes* sur ma machine) du fait que `temp2` n'y est pas libérée et le programme dans son ensemble détectera une fuite de `11*sizeof(int)` *bytes* (les dix `int` du tableau `q` et le `int` pointé par `temp2`).

```
#include "Allocation.h"
int main() {
   {
      ComptableMemoire::TraceurLocalSimplet tr;
      auto temp = new int[5];
      atemp2 = new int(4);
      delete[] temp;
   }
   auto p = new int(3);
   auto q = new int[10];
   delete p;
}
```

Exemple : superviser les allocations suspectes

L'approche précédente montre au moins deux importants défauts :

- le premier est qu'elle émet des données à la console dans son destructeur, or le comptable est un singleton statique ce qui signifie qu'on ne peut dire au préalable s'il sera détruit avant ou après le flux d'erreur standard; et
- le deuxième est qu'elle n'offre des services que pour détecter des déséquilibres en fin d'exécution d'un programme (à travers le destructeur du singleton comptable) et dans le cas où un bloc de mémoire inclut une combinaison allocation/ libération de mémoire qui serait déséquilibrée. Tout autre scénario doit être implémenté manuellement, ce qui est fastidieux.

Nous allons raffiner ce modèle de manière à permettre aux intéressés de réagir à n'importe quelle allocation dynamique de mémoire et à n'importe quelle libération de mémoire. Ainsi, il deviendra possible de surveiller des allocations suspectes (par exemple, tout ce qui dépasse un certain seuil de mémoire demandée) et de réagir en conséquence).

L'idée de permettre à un nombre arbitrairement grand d'abonnés de réagir à un événement est une application directe du schéma de conception Observateur [hdPatt].

La première étape sera d'offrir un service d'abonnement à même le singleton comptable. Ainsi, chaque fois que le comptable sera informé d'une allocation ou d'une libération de mémoire, il sera en mesure de signaler l'occurrence de cet événement à ses abonnés.

L'implémentation proposée ici repose sur une interface interne nommée `Rappelable`. Des instances de toute classe dérivant de `Rappelable` pourront s'abonner au comptable pour se faire offrir une opportunité de réagir lorsque se produite une allocation ou une libération de mémoire.

Puisque `Rappelable` est une abstraction polymorphique, le comptable conservera un conteneur de `Rappelable*` plutôt qu'un conteneur de `Rappelable`. Ceci permettra de réaliser l'invocation polymorphique des méthodes des abonnés.

Remarquez aussi que `sur_allocation()`,invoquée lors d'une allocation de mémoire, et `sur_deallocation()`,invoquée lors d'une libération de mémoire, sont polymorphiques sans être abstraites, ce qui facilitera la tâche des abonnés : ces derniers n'auront pas à spécialiser les deux méthodes s'ils n'en veulent qu'une seule.

```
// ... voir version antérieure ...
#include <vector>
#include <algorithm>
// ...using...
class ComptableMemoire {
public:
   // ... voir version antérieure ...
   struct Rappelable {
      virtual void
         sur_allocation(size_type){
         }
      virtual void
         sur_deallocation(size_type) {
         }
      virtual ~Rappelable() = default;
   };
private:
   vector<Rappelable *> abonnes;
```

Abonner un `Rappelable` non nul implique simplement de l'insérer dans le conteneur d'abonnés.

Désabonner un `Rappelable` non nul implique de le retrouver puis de l'enlever du conteneur. On pourrait aussi, au besoin, ajouter une opération supplémentaire pour compacter le conteneur si la mémoire disponible se faisait rare.

```
public:
   void abonner(Rappelable *p) {
      if (p) abonnes.push_back(p);
   }
   void desabonner(Rappelable *p) {
      if (p)
         abonnes.erase(
            find (begin(abonnes),end(abonnes),p)
         );
   }
```

Enfin, les méthodes `allocation()` et `deallocation()` seront adaptées à la réalité d'un service d'abonnement par l'insertion, dans chacune d'entre elles, d'une répétitive pour rappeler correctement les abonnés.

```
    void allocation(size_type n) {
       qte += static_cast<size_type>(n);
       for(auto & p : abonnes)
           p->sur_allocation(n);
    }
    void deallocation(size_type n) {
       for(auto & p : abonnes)
           p->sur_deallocation(n);
       qte_ -= static_cast<size_type>(n);
       if (quantite() < 0)
          surliberation_identifiee();
    }
    // ... voir version précécente...
};
```

Quoi faire lors d'allocation et de libération dynamiques de mémoire?

L'allocation dynamique de mémoire se fera habituellement comme suit :

- tout d'abord, un bloc de mémoire de la taille demandée (donnée par le paramètre passé à `new` ou à `new[]`) sera identifié. Habituellement, on s'assurera que ce bloc soit aligné en mémoire de manière à y rendre l'accès le plus rapide possible, ce qui signifiera souvent faire en sorte que l'adresse où il débutera sera un multiple de la taille en *bytes* d'un mot mémoire sur l'ordinateur cible[54];
- si un problème survient (par exemple si la mémoire manque), alors deux possibilités s'offrent au développeur: lever une exception ou laisser le gestionnaire d'erreurs en place procéder (voir ***Gérer les échecs***, plus loin); et
- retourner un pointeur abstrait sur le début du bloc de mémoire ainsi alloué.

Habituellement, dans le cas de l'allocation dynamique de mémoire pour un tableau, on notera au passage la taille en *bytes* du bloc alloué, de manière à libérer le bon nombre de *bytes* lorsque l'invocation correspondante de l'opérateur `delete[]` sera faite.

De son côté, la libération de mémoire allouée dynamiquement se fait habituellement de la manière suivante :

La stratégie la plus simple est d'utiliser les `sizeof(size_t)` *bytes* situés juste avant le bloc de mémoire obtenu pour déposer la taille (en nombre de *bytes*) de ce bloc. Ainsi, une invocation de `delete[]` n'aura qu'à regarder les `sizeof(size_t)` *bytes* précédent le pointeur à libérer pour savoir combien de mémoire récupérer.

Ceci ne s'adapte pas à toutes les situations. Par exemple, avec `ArenaFixe` (plus haut), une approche cachant des tailles juste avant les tableaux détruit les avantages de la stratégie de gestion par blocs fixes.

- traiter le cas où le pointeur à libérer est nul; et
- libérer la mémoire correspondant à ce pointeur d'une manière convenant à la stratégie d'allocation dynamique de mémoire originale.

Il arrivera que les mécanismes de libération de la mémoire allouée dynamiquement profitent d'un appel à `delete` ou à `delete[]` pour réduire la fragmentation de la mémoire ou procéder à d'autres opérations de nettoyage ou d'entretien. Évidemment, il y a un prix à payer pour faire de telles opérations (tout comme il y a un prix à payer pour ne pas les faire), alors tout choix d'implémentation en ce sens doit être calculé et mesuré avec soin.

[54] On parle d'un multiple de `4` sur les machines dont les registres ont une capacité de `32` bits.

Gérer les échecs

Si le besoin de gérer manuellement les erreurs d'allocation de mémoire se fait sentir, trois grandes stratégies sont possibles :

- la manière traditionnelle, qui est de retourner `nullptr`, est conforme au standard C mais n'est pas conforme au standard C++;
- se conformer au standard C++ et lever un `std::bad_alloc`; et
- la manière générique, qui est de remplacer le gestionnaire d'erreurs par défaut lors d'allocation dynamique de mémoire par un gestionnaire maison.

Remplacement global

Le gestionnaire standard d'erreurs d'allocation dynamique de mémoire est une fonction globale dont la signature est celle d'un `new_handler`, donc une fonction globale ou une méthode de classe de type `void` et ne prenant pas de paramètres.

On peut remplacer ce gestionnaire par un autre avec `set_new_handler()`. Cette fonction globale reçoit en paramètre l'adresse du `new_handler` à appeler lors d'un problème, et retourne l'adresse du `new_handler` précédent.

Cette stratégie symétrique se prête à une application de l'idiome `RAII`, comme dans l'exemple à droite.

Le standard définit le type `std::new_handler` comme étant une fonction `void` ne prenant pas de paramètre. Elle a pour rôle de prendre une décision quant aux actions à prendre lors d'un problème d'allocation de mémoire.

```
#include <iostream>
#include <new>
// ...using...
class GestionnaireAllocation {
   new_handler handler; // le précédent
   static void newHandler() { // le notre
      cerr << "Échec de new"; // bof
      throw bad_alloc{};
   }
   static GestionnairAllocation singleton;
   // remplacer l'ancien gestionnaire
   GestionnaireAllocation() noexcept
      : handler{set_new_handler(newHandler)} {
   }
public:
   // replacer l'ancien gestionnaire
   ~GestionnaireAllocation() {
      set_new_handler(handler);
   }
};
```

Remplacement local

Rien n'empêche d'avoir plusieurs gestionnaires potentiels en poche et de faire en sorte que la portée de chacun garantisse une gestion des erreurs spécifique. En pratique, cela peut d'ailleurs être très utile : avec une approche `RAII`, un échec d'allocation locale par l'opérateur `new` peut mener à une deuxième chance (et à une troisième, puis une quatrième, *etc*.) en basculant du gestionnaire le plus local au gestionnaire le plus global.

Un `std::new_handler` devra, au choix :

- obtenir de la mémoire puis la retourner (peut-être a-t-il accès à des réserves d'urgence? Peut-être peut-il « débloquer » des ressources au niveau du système d'exploitation?). Par exemple, il pourrait allouer au préalable une réserve de dernier recours qui sera sollicitée pour des opérations d'urgence si le programme vient à manquer de mémoire;
- lever un `std::bad_alloc`, dans le cas où lui ne parvient pas à trouver de nouvelles ressources; ou
- invoquer `std::abort()` ou `std::exit()` dans le cas où il n'y a plus rien à faire et où il vaut mieux abandonner, tout simplement.

L'exemple proposé à droite montre une approche `RAII` pour remplacer temporairement et localement le gestionnaire habituel d'erreurs d'allocation dynamique de mémoire par un gestionnaire plus spécifique aux besoins de la classe `X`.

```
#include <new>
using std::new_handler;
using std::set_new_handler;
class auto_new_hdl {
   new_handler hdl;
public:
   auto_new_hdl(new_handler p)
      : hdl{set_new_handler(p)} {
   }
   ~auto_new_hdl() {
      set_new_handler(hdl);
   }
};
class X {
   // ...
   static void hdl_local();
public:
   // ...
   void* operator new(size_t n) {
      auto_new_hdl hdl{hdl_local};
      return ::operator new(n);
   }
   // ...
};
```

Remarquez que, si l'opérateur `new` global lève `std::bad_alloc`, il déléguera à `hdl_local` la tâche d'obtenir de la mémoire par d'autres moyens; si `hdl_local` lève lui aussi `std::bad_alloc`, alors l'opérateur `new` de `X` se terminera, et le destructeur de la variable `anh` remettra en place le `std::new_handler` précédent et la mécanique suivra son cours.

Notez que, lors du `delete` (ou du `delete[]`) correspondant à un `new` (ou à un `new[]`), il n'est pas possible de savoir si un `std::new_handler` a dû intervenir à l'origine. Il faut donc que l'opérateur `delete` (ou `delete[]`) interagisse sainement avec la mémoire obtenue par l'action d'un `std::new_handler`.

Allocation assistée

Les stratégies d'allocation dynamique de mémoire décrites ci-dessus sont des stratégies côté serveur : c'est chaque fois le mécanisme d'allocation dynamique de mémoire en place qui choisit, en fonction de ses politiques, l'endroit où il lui faut prendre la mémoire pour un bloc donné.

D'autres approches sont possibles, en fonction des besoins du code client. Ces approches s'expriment dans la signature des opérateurs `new`, `new[]`, `delete` et `delete[]` par l'injection de paramètres supplémentaires. Nous explorerons, dans les sections qui suivent, quelques-unes des déclinaisons les plus répandues de ces approches.

Allocation positionnelle

À bien des égards, cette déclinaison de `new` est la plus fondamentale. Les diverses versions de `new` possibles, en effet, allouent toutes un bloc de mémoire pour que celui-ci soit initialisé par un constructeur. L'allocation positionnelle, elle, applique un constructeur sur un bloc de mémoire préalablement obtenu.

L'allocation positionnelle, qu'on nomme aussi le *Placement `new`*, décrit le cas où le code appelant détermine lui-même le lieu où sera placé l'objet. Ceci permet non pas de réaliser une *allocation dynamique* de mémoire mais bien de réaliser un *positionnement délibéré* pour un objet.

On a habituellement recours à cette stratégie quand l'objet doit se trouver dans une zone mémoire spécifique, comme par exemple placer le vecteur d'interruptions du système d'exploitation à une adresse fixe, ou en un lieu physique précis, comme dans le cas où les attributs d'un objet doivent correspondre avec précision à des registres ou à d'autres lieux matériels.

```
// adresse arbitraire
const void *LIEU =
   static_cast<void*>(0x0000ffff0000ffff);
// on veut que l'objet vers lequel pointe
// pX soit placé précisément à LIEU
X *p0 = new (LIEU) X;
// autre possibilité: zone connue
char tampon [sizeof(X)];
X *p1 = new (static_cast<void*>(&tampon[0])) X;
```

Syntaxiquement, l'allocation positionnelle demande simplement qu'on intercale entre `new` et le type à instancier une adresse brute (un `void*`), qui est l'adresse où placer l'objet créé. Cette adresse doit être placée entre parenthèses.

On ne surcharge pas l'allocation positionnelle, pour laquelle le compilateur fournira une implémentation par défaut correcte prenant la forme d'un simple *Pass-Through*., du fait qu'aucune allocation réelle n'y est faite.

```
void* operator new(size_t, void *p) {
   return p;
}
void* operator new[](size_t, void *p) {
   return p;
}
void operator delete(void*, void*) {
}
void operator delete[](void*, void*) {
}
```

Puisque l'allocation positionnelle n'alloue pas vraiment de mémoire, il est important de ne pas invoquer `delete` sur le pointeur ainsi obtenu; cela impliquerait une libération de mémoire jamais allouée, et mettrait le programme dans un état qu'on pourrait qualifier d'instable; il y a toutefois un besoin pour un `delete` correspondant à chaque `new`; la section ***Allocation assistée***, plus bas, discute de cette question.

Notez que, dans du code générique, il est parfois laborieux d'exprimer correctement le type de l'objet construit. Par exemple, prenons le code suivant :

```
#include <iostream>
#include <string>
using namespace std;
string creer_message() {
   return "J'aime mon prof";
}
int main() {
   char buf[sizeof(string)];
   auto p = new (static_cast<void*>(&buf[0])) string(creer_message());
   // ...
}
```

… où le type `std::string` de la valeur retournée par `creer_message()` est évident sur la base du contexte, mais pourrait l'être moins dans une situation générique. Il est possible d'exprimer le même code de la manière suivante :

```
#include <iostream>
#include <string>
using namespace std;
string creer_message() {
   return "J'aime mon prof";
}
int main() {
   char buf[sizeof(string)];
   auto p = new (static_cast<void*>(&buf[0])) auto(creer_message());
   // ...
}
```

Allocation sans exceptions

Une autre approche à l'allocation assistée est celle qui évite le recours à une levée d'exception lorsque la mémoire demandée à `new` ne peut être allouée, privilégiant plutôt que cet opérateur retourne alors un pointeur nul, comme le fait le code C avec `std::malloc()`. Bien que cette approche ne soit pas à privilégier en général (elle rend, entre autres, plus complexe la rédaction de constructeurs résilients), elle peut être nécessaire dans des systèmes embarqués ou dans des systèmes soumis à de contraintes de temps réel[55].

Cette version de `new` prend l'une des formes décrites à droite. Le paramètre supplémentaire est une référence `const` sur un `std::nothrow_t`, qui est typiquement une classe vide, et l'objet global `std::nothrow` est une instance de ce type.

Tel que mentionné plus haut, l'allocation *no throw*, lorsqu'elle échoue, retourne un pointeur nul. Ceci correspond au comportement de `new` dans le passé, et correspond au comportement attendu de certains programmes choisissant d'éviter les exceptions.

```
void* operator new
   (size_t, std::nothrow_t);
void* operator new[]
   (size_t, std::nothrow_t);
// ...
X *p = new (std::nothrow) X;
if (p) {
   // ... allocation réussie; utiliser p
}
```

Allocation gérée par un tiers

L'allocation assistée permet aussi, de par l'allocation gérée par un tiers, de tracer une frontière entre le gestionnaire de mémoire et l'objet pour lequel la mémoire est allouée. En d'autres mots, au moment de l'allocation de mémoire, plutôt que de déterminer l'adresse où sera construit un objet, le code client pourra déterminer *quel gestionnaire de mémoire* sera responsable d'allouer la mémoire cet objet.

Cette stratégie tient compte du fait qu'il existe plusieurs objets capables de gérer l'allocation dynamique de mémoire dans un programme. Certains peuvent avoir une stratégie très économique du point de vue de la gestion de l'espace, récupérant toutes les miettes possibles; d'autres peuvent être très rapides, limitant au possible la récupération, quitte à ce que la mémoire se fragmente plus rapidement; d'autres encore peuvent utiliser des zones de mémoire spéciales, par exemple une plage mémoire dupliquée sur un média physique de manière à assurer une forme de persistance des objets lors de pannes de courant.

Sachant cela, il est possible d'exposer une version de l'opérateur `new` (sous forme de fonction globale ou sous forme de méthode) qui prenne en paramètre une indirection (un pointeur ou une référence, selon les prédilections) vers un objet capable d'allouer de la mémoire, pour que la mémoire demandée soit obtenue à l'aide de cet objet.

```
void* operator new
   (size_t n, ZoneMemoire &zm) {
   return zm.reserverBloc(n);
}
```

[55] Avec l'état actuel des connaissances, il est plus simple de prédire le temps d'exécution requis pour tester un pointeur, dans le but de savoir s'il est nul ou non, que de mesurer le temps requis pour passer d'une levée d'exceptions (`throw`) à sa captation (`catch`).

Ceci ouvre la porte à une multitude de stratégies souples d'attribution dynamique de la mémoire. Il devient possible de concevoir plusieurs gestionnaires d'allocation dynamique de mémoire spécialisés, et de laisser les objets choisir sur une base individuelle ou par projet la stratégie qui leur convient le mieux[56].

Sachant cela, il devient simple de créer un objet en déléguant à un gestionnaire de mémoire la tâche d'allouer un bloc de mémoire.

```
ZoneMemoire zm{ /* ... */ };
// ...
X *pX = new (zm) X;
```

Le terme gestionnaire de mémoire est un peu abusif ici; on pourrait envisager une allocation dans un lieu particulier (pensez à une scène, dans un jeu vidéo), dans quel cas le `new` pourrait être perçu comme le gestionnaire de mémoire alors que le tiers qui lui est offert deviendrait la zone dans laquelle la mémoire doit être pigée.

Bien que cette approche soit moins connue et moins répandue, il est possible d'écrire des implémentations assistées de l'opérateur `new` (et, bien sûr, de l'opérateur `new[]`) qui exploitent plus qu'un objet assistant.

L'exemple (académique, par souci de simplicité) à droite présente la syntaxe d'un `new` appuyé sur deux tiers (un `X` et un `Y`), qui ne fait en pratique que déléguer l'allocation réelle vers l'opérateur `new` global à un seul paramètre.

Cette approche peut être intéressante si les divers assistants passés à `new` jouent des rôles différents dans le programme (un représentant une zone mémoire et l'autre une stratégie d'allocation, par exemple).

```
#include <new>
#include <iostream>
// ...using...
class X {};
class Y {};
void *operator new(size_t n, X&, Y&) {
   cout << "Demande de " << n
        << " bytes..." << endl;
   return ::operator new(n);
}
int main() {
   X x;
   Y y;
   auto p = new (x, y) int{3};
   delete p;
}
```

[56] En fait, si la conception a été pensée en conséquence, un simple `using` pourrait permettre à un programme de choisir la stratégie globale d'allocation de mémoire qui lui convient le mieux!

Allocation assistée et finalisation

L'allocation positionnelle sous sa forme la plus épurée, celle où le code client choisit l'endroit où l'objet doit être construit (celle où le paramètre positionnel est un `void*`) ne fait aucune réservation de mémoire. Conséquemment, bien que l'objet construit doive être finalisé, l'adresse associée à l'objet ne doit pas être libérée.

L'exemple à droite montre une allocation positionnelle brute d'un `X` dans un tableau de *bytes* alloué sur la pile. Appliquer `delete p;` dans ce programme le ferait planter, du fait que le lieu où pointe `p` est sur la pile, pas dans le tas.

```
class X { /* ... */ };
int main() {
   char espace[sizeof(X)];
   auto lieu =
      static_cast<void *>(espace);
   X *p = new (lieu) X;
   // ...
   p->~X(); // surtout pas delete p;!
}
```

Dans un tel cas, il faut finaliser l'objet (en invoquant explicitement son destructeur), sans plus. Une fois l'objet finalisé, l'espace qui lui avait été associé pourra être réutilisé.

Le cas de l'allocation positionnelle assistée comporte ses propres problèmes logistiques. En effet, une fois un objet créé, il n'est plus possible pour le moteur de se souvenir de la bonne version de `delete` à lui appliquer.

De même, en cours de création, le moteur connaît la version de l'opérateur `new` utilisée et peut avoir besoin de solliciter la bonne version de l'opérateur `delete` si le constructeur appelé lève une exception.

À titre d'exemple, imaginez le cas suivant :

- une classe `X` quelconque est telle qu'à certains moments, sa construction est susceptible de lever une exception;
- dans l'exemple à droite, `X::X()` lèvera une exception (un `Oups`, bien nommé) lors de sa deuxième invocation, mais pas lors de la première;
- cette exception est distincte du `bad_alloc` que l'opérateur `new` est susceptible de lever. En effet :
  - un `bad_alloc` (levé par `new`) signifierait une incapacité à trouver la mémoire demandée; alors que
  - un `Oups` (levé par le constructeur) signifierait un problème lors de l'initialisation de la mémoire ainsi allouée;

```
class Oups {};
class X {
   static int n;
public:
   X() {
      if (n++ > 0) throw Oups{};
   }
};
// dans X.cpp, quelque part...
int X::n = 0;
```

```
#include <new>
using std::bad_alloc;
using std::size_t;
class Tiers {};
void *operator new(size_t n, Tiers&) {
   cout << "Allocation assistée..." << endl;
   return ::operator new(n);
}
void operator delete(void *p, Tiers&) {
   cout << "destruction assistée..." << endl;
   ::operator delete(p);
}
```

Le programme à droite illustre à la fois l'utilisation du `new` assisté, du `delete` assisté et des exceptions :

```
int main() {
   Tiers tiers;
   Try {
      X *p = new (tiers) X;
      delete p;
      p = new (tiers) X; // oups!
      delete p;
   } catch (Oups) {
      cerr << "Oups!\n";
   } catch (bad_alloc&) {
      cerr << "bad_alloc!\n";
   } catch (...) {
      cerr << "Suspect...\n";
   }
}
```

- le premier appel à `new(tiers)X` instanciera un `X`, avec un (très) faible risque de lever `bad_alloc` dans des circonstances limites;
- il n'y a pas d'appel explicite possible à la forme assistée de `delete`. Dans ce cas bien précis, sachant que l'allocation effective est faite par l'opérateur new conventionnel, notre code invoque `delete` pour libérer chaque `X`;
- la deuxième invocation de `new(tiers)X` échouera non pas lors de l'allocation (outre un peu probable `bad_alloc`, évidemment) mais lors de l'initialisation, par `X::X()`, de la zone allouée par new. Ici, le moteur du langage est dans l'eau chaude : il est responsable de la mémoire qu'il a allouée, mais n'est pas parvenu à l'initialiser tel que le code client le lui a demandé. Il doit donc libérer lui-même la mémoire obtenue par `new(tiers)` et laisser filtrer l'exception levée par `X::X()` jusqu'au code client, par souci de neutralité.

C'est dans un tel cas que le moteur aura recours au `delete` assisté dont la signature (en fait, dont les types des paramètres additionnels) correspondra à celle du `new` assisté suite auquel l'initialisation aura échoué. Le code client n'y a pas accès, du moins pas à travers la signature habituelle (voir plus bas pour un exemple d'invocation de `delete` assisté par le code client), mais le moteur la connaît et l'invoquera au moment opportun.

En pseudocode, on pourrait dire qu'un `new` assisté a, à haut niveau (pas dans l'intérieur de l'opérateur new le comportement suivant :

- allouer la mémoire demandée, et lever un `bad_alloc` dans le cas où il s'avère impossible de réaliser l'allocation demandée;
- tenter (bloc `try`) d'appliquer le constructeur demandé par le code client. Si cela fonctionne, le travail est terminé et le pointeur sur la mémoire (correctement, on le présume) initialisée est retourné au code client;
- attraper (`catch(...)`) toute exception levée lors de l'initialisation. Dans ce cas, remettre la mémoire obtenue plus haut (`delete` assisté correspondant au `new` assisté utilisé), puis relancer l'exception (`throw;`) au code client pour fins de neutralité.

En résumé :

- il faut écrire un *delete* assisté correspondant à chaque *new* assisté, pour que le moteur soit capable d'y faire appel *dans le cas où le constructeur aurait levé une exception*. L'appel au *delete* assisté sera réalisé automatiquement;
- évidemment, il faut trouver une stratégie pour détruire correctement l'objet créé avec un *new* assisté puisqu'il est hautement probable que l'opérateur `delete` normal n'offre alors pas la fonctionnalité désirée;
- en pratique, coder une version de `new` devrait impliquer la rédaction de `new[]`, `delete`, `delete[]`, et ce au moins sous deux versions, soit celles susceptibles de lever un `bad_alloc` et celles qui respectent la signature munie de `std::nothrow`;

⇒ de manière générale, donc, ***quiconque souhaite coder l'une de ces fonctions en codera au moins huit***.

En pratique, écrire un *delete* assisté est relativement simple. Effectivement, présumant une classe `X` exposant un `new` assisté, le *delete* assisté correspondant sera généralement comme proposé à droite.

```
class X {
   // ...
public:
   void * operator new
      (size_t n, ZoneMemoire &zm) {
      return zm.reserverBloc(n);
   }
   void operator delete
      (void *p, ZoneMemoire &zm) noexcept {
      if (!p) return;
      zm.libererBloc(p);
   }
   // ...
};
```

La signature suit toujours le même patron, celui l'opérateur `delete` normal mais avec un ou plusieurs paramètres supplémentaires.

La correspondance entre un *new* assisté et le *delete* qui lui convient se fait sur la base du type de ces paramètres supplémentaires. Rappelons que, dans la majorité des cas, il n'y aura qu'un seul paramètre supplémentaire.

Suite à l'allocation assistée d'une instance d'une classe `X`, la destruction se fera en deux temps : d'abord par un appel explicite au destructeur, pour finaliser l'objet à détruire, puis par un appel explicite au *delete* désiré (remarquez la syntaxe de l'invocation).

```
ZoneMemoire zm { /* ... */ };
// ...
auto p = new (zm) X;
// ...
p->~X();
operator delete(p, zm);
```

Une autre stratégie viable, si plusieurs classes d'objets sont susceptibles d'être prises en charge par un même gestionnaire de mémoire, et de les faire dériver toutes d'une même classe abstraite possédant un destructeur virtuel.

Une manière intéressante de réaliser cette stratégie est de faire de la classe en question une classe interne et publique du gestionnaire de mémoire choisi (p. ex. : une classe `ZoneMemoire::Destructible`).

Introduction aux allocateurs

Il est possible de construire des programmes OO qui soient à la fois élégants, efficaces et pragmatiques. La section sur l'allocation positionnelle, ci-dessus, en fait foi.

Certains langages OO, dont Java et les langages .NET, comptent sur une machine virtuelle, compilée pour chaque plateforme ciblée, pour assurer la portabilité de leurs binaires. Ce faisant, ces langages offrent un modèle de programmation homogène, car la machine pour laquelle les programmes sont compilés sera indépendante, en surface, de la plateforme sous-jacente.

Dans le cas de C++, où la philosophie de base est de ne faire payer les programmes que pour ce qu'ils utilisent et où, conséquemment, un rapport plus immédiat aux ressources de la machine est parfois requis, la question de la plateforme sous-jacente se pose à l'occasion. Un programme C++ peut vouloir connaître les caractéristiques locales (bornes, tailles, représentation interne des adresses, signe du type `char`) des types primitifs; là où une machine virtuelle offre une définition stable pour toute plateforme pour ce qui y joue le rôle de type primitif, un compilateur C++ privilégiera pour ses types primitifs la représentation la plus efficace possible pour la plateforme en fonction de laquelle le programme sera compilé.

Une considération fondamentale pour réaliser du code qui soit à la fois portable, stable, élégant et efficace en C++ est donc de définir une représentation efficace et efficiente du modèle de mémoire sous-jacent de la plateforme.

Par exemple, connaître la taille du mot mémoire d'une une machine donnée importe pour qui développe des algorithmes génériques devant fonctionner pour toute séquence, du plus humble tableau au plus complexe conteneur. Cette considération ne doit par contre pas influencer outre mesure la généricité du code, que cette généricité s'exprime par du polymorphisme ou par des mécanismes de programmation générique. Ceci explique le souci d'encapsuler ces détails derrière des objets, typiquement des itérateurs, et le recours à des types opérationnellement homogènes. Encapsuler l'effort d'abstraction des considérations sous-jacentes dans des objets permet d'élever le niveau du discours dans les structures plus générales que sont les conteneurs et les algorithmes standards.

Le problème a été rencontré par les gens œuvrant sur la bibliothèque STL : comment représenter le concept de différence entre deux itérateurs, donc potentiellement entre deux pointeurs génériques, sans brûler dans le code le concept de taille d'un registre? Comment rédiger du code générique en sachant que, sur certaines plateformes, les pointeurs de fonction ne sont pas soumis aux mêmes règles que les pointeurs sur des données? Les règles applicables aux mémoires persistantes et celles applicable aux mémoires partagées sont-elles les mêmes que celles applicables aux mémoires conventionnelles?

Une réflexion quant à la définition d'un modèle de mémoire[57] s'avérera donc nécessaire pour réaliser un idéal de programmation OO à la fois générique et efficace. Parmi les objectifs principaux guidant cette réflexion, on trouvera celui de dissocier les types et les algorithmes des considérations locales au modèle de mémoire : comment faire en sorte que les idéaux visés soient rencontrés tout en limitant au minimum le couplage entre le code et la représentation du modèle de mémoire sous-jacent?

[57] Qu'est-ce qu'un pointeur sur une donnée? Qu'est-ce qu'un pointeur sur du code? Comment représenter la différence entre deux adresses? Qu'est-ce qu'une optimisation admissible? Les réponses à ces questions sont-elles les mêmes en monoprogrammation et en multiprogrammation?

Le concept sophistiqué résultant de cette réflexion est celui d'**allocateur**[58]. Le découplage résultant est tellement bien réalisé que la plupart des programmeurs C++ ne savent même pas que les allocateurs existent.

À titre d'exemple, la signature de `std::vector` est non pas celle-ci…

```
template <class T>
   class vector { /* ... */ };
```

… mais bien celle-ci, ce qui influence l'écriture de code générique utilisant des conteneurs comme lui et les autres conteneurs standards `STL`.

```
template <class T,
          class A = allocator<T> >
   class vector { /* ... */ };
```

Pour introduire le sujet, voici une citation d'**Alexander Stepanov** (l'un des principaux parents de la bibliothèque `STL`) tirée de [StepDobb]. Stepanov répond à une question à propos de l'absence d'un conteneur implémentant la persistance des objets dans `STL` (les caractères gras et les passages entre crochets sont de moi) :

> *« This point was noticed by many people. STL does not implement persistence for a good reason. STL is as large as was conceivable at that time. I don't think that any larger set of components would have passed through the standards committee. But persistence is something that several people thought about.* ***During the design of STL and especially during the design of the allocator component, Bjarne [Stroustrup] observed that allocators, which encapsulate memory models, could be used to encapsulate a persistent memory model****. The insight was Bjarne's, and it is an important and interesting insight. Several object database companies are looking at that. In October 1994 I attended a meeting of the Object Database Management Group. I gave a talk on STL, and there was strong interest there to make the containers within their emerging interface to conform to STL. They were not looking at the allocators as such. Some of the members of the Group are, however, investigating whether allocators can be used to implement persistency. I expect that there will be persistent object stores with STL-conforming interfaces fitting into the STL framework within the next year. […]* ***Irrespective of Intel architecture, memory model is an object, which encapsulates the information about what is a pointer, what are the integer size and difference types associated with this pointer, what is the reference type associated with this pointer, and so on. Abstracting that is important if we introduce other kinds of memory such as persistent memory, shared memory, and so on****. A nice feature of STL is that the only place that mentions the machine-related types in STL—something that refers to real pointer, real reference—is encapsulated within roughly 16 lines of code. Everything else, all the containers, all the algorithms, are built abstractly without mentioning anything which relates to the machine. From the point of view of portability, all the machine-specific things which relate to the notion of address, pointer, and so on, are encapsulated within a tiny, well-understood mechanism. Allocators, however, are not essential to STL, not as essential as the decomposition of fundamental data structures and algorithms*. »

[58] … rien n'est parfait. Dans [EffStl], item `10`, **Scott Meyers** débute sa section sur les allocateurs par cette phrase révélatrice : « *Allocators are weird* », puis utilise l'item `10` pour explique plusieurs situations en fonction desquelles les allocateurs ne sont pas utiles, contrairement aux apparences. Heureusement, l'item `11`, tout juste après, montre en quoi ils sont utiles.

Les allocateurs ne sont pas essentiels aux algorithmes et aux types génériques de bibliothèques aussi générales que STL, mais permettent à des algorithmes d'œuvrer aussi efficacement que possible sur un modèle de mémoire sans connaître le détail de ce modèle de mémoire.

Le concept d'allocateur a été abordé par plusieurs individus, et est un sujet au sujet duquel plusieurs chaudes discussions ont été entretenues.

L'approche de STL[59] au sens standard est de discuter d'allocateur au sens de mécanisme d'attribution de blocs de mémoire, donc dans un sens se rapprochant de l'allocation positionnelle discutée plus haut.

Cette approche se défend très bien pour une bibliothèque comme STL, dont l'optique principale est celle de conteneurs d'un type donné ou d'algorithmes applicables à une séquence d'un type donné.

Elle tente de faire en sorte que le modèle applicable aux erreurs d'allocation de mémoire soit aussi stable que possible sur plusieurs plateformes et de réutiliser le plus possible des blocs de mémoire d'une même taille.

Les stratégies d'allocation de mémoire par type, cherchant à réutiliser la mémoire de manière locale à chaque classe, tendant à fragmenter la mémoire si le nombre de classes est élevé.

Dans le cas des conteneurs STL, le modèle programmation tend à focaliser sur un type à la fois, ce qui réduit de beaucoup le potentiel de fragmentation de la mémoire résultant de l'application d'une telle stratégie.

L'approche de C++[60] est différente[61], la portée du langage étant différente de celle d'une partie de sa bibliothèque. En C++, un allocateur est donc une classe respectant l'interface de la classe `std::allocator`, et `std::allocator` en tant que tel est l'allocateur par défaut sur une plateforme donnée.

[59] Voir [StlAllo] pour un bref texte défendant cette position.

[60] Voir [CppAllo] pour une discussion de cette approche.

[61] Il faut sans doute mentionner que, bien que STL ait d'abord pris son envol en C++ étant donné la force de son système de types et sa capacité d'offrir une véritable mécanique d'abstraction des types, cette bibliothèque a été développée de manière indépendante, initialement avec d'autres langages (mais avec moins de succès, la puissance expressive n'étant pas égale pour tous les langages) et a été insérée dans le standard ISO de C++ un peu comme si elle tombait du ciel. **Alex Stepanov** a rencontré **Andrew Koening** (un proche de **Bjarne Stroustup**) et, suite à une discussion serrée, Koenig a réalisé le potentiel de STL et a travaillé, de concert avec Stepanov et Stroustrup, à l'insertion d'une version allégée de STL dans le standard ISO de C++.

Divergences de modèle

La conception d'allocateurs a ceci de particulier qu'elle examine la gestion de la mémoire sous une lueur différente de cette utilisée par les mécanismes d'allocation que sont `new` et `delete`, vus précédemment :

- un allocateur est un objet offrant une gamme précise de services et de types, alors que `new` et `delete` sont des fonctions;
- un allocateur est générique sur la base du type en vertu duquel il alloue la mémoire travaille, et peut donc invoquer des services de ce type, en particulier certains de ses constructeurs et son destructeur;
- un allocateur ne peut avoir d'états, du fait que le standard indique que, pour un allocateur générique `A` donné, `A<T>` et `A<U>` doivent être considérés égaux au sens d'une comparaison sans égard aux types `T` et `U`. Cette contrainte est rigide puisqu'elle ne permet pas d'utiliser un attribut d'instance (par exemple, l'adresse d'une zone de mémoire à utiliser) dans un allocateur. En retour, elle simplifie la gestion de la duplication et de la portée des allocateurs, question qui ne serait pas simple du tout en temps normal;
- un allocateur doit pouvoir se représenter lui-même tel qu'il serait si on l'appliquait sur un autre type. Bien que cette phrase puisse paraître suspecte, la manœuvre de programmation derrière cette idée sera expliquée plus bas (elle repose sur un type générique interne nommé `rebind`). C'est de la poésie.

L'allocateur standard

Cette section doit être encore un peu mise à jour pour tenir compte de raffinements depuis C++ 11. Simple question de temps… En attendant, voir [hdAllocCpp11].

Un allocateur a pour mandat de représenter un modèle de mémoire tel qu'applicable à un type `T` donné, d'une manière homogène (pour que tout conteneur puisse l'utiliser à loisir). Ceci implique un respect strict, par tout allocateur, de l'interface de l'allocateur standard.

Un allocateur doit définir, à partir de types internes et publics, les idées suivantes :

- que signifie être un pointeur, constant ou non, ou une référence, constante ou non, sur une instance de `T`;
- ce que signifie être une taille ou une différence de pointeurs sur un `T`;
- ce que signifie être une valeur de type `T`; et
- ce que signifie obtenir l'adresse, constante ou non, d'un `T`.

```
template <class T>
   class allocator {
      using size_type = size_t;
      using difference_type = ptrdiff_t;
      using pointer = T*;
      using const_pointer = const T*;
      using reference = T&;
      using const_reference = const T&;
      using value_type = T;
```

Ces types pourront être utilisés par les conteneurs eux-mêmes dans la détermination de leurs propres choix de types internes et publics, bien que rien ne les y oblige.

Dans certains cas, par exemple celui d'une liste de `T`, il est possible que le conteneur n'alloue pas tant le type indiqué (des instances de `T`) que des représentations internes accompagnant ce type (des nœuds contenant des instances de `T`).

La (superbe) manœuvre permettant de dupliquer un allocateur passe par un type générique interne nommé `rebind`. Dupliquer un allocateur `A<T>` pour le type `U` signifie utiliser `A<T>::rebind<U>::other`, ce qui ne coûte rien puisque les allocateurs sont des objets sans états.

```
      // ...
      template <class U>
         struct rebind {
            using other = allocator<U>;
         };
      // ...
```

Les services de construction, de copie et de destruction, de types apparentés ou non, devraient tous être banals et sans risques de lever une exception puisqu'il n'y aura aucune duplication d'état.

```
      allocator() noexcept;
      allocator(const allocator&) noexcept;
      template <class U>
         allocator(const allocator<U>&) noexcept;
      template <class U>
         allocator& operator=(const allocator<U>&) noexcept;
      ~allocator();
```

L'absence d'états signifie aussi que, si la construction de copie apparentée et l'affectation apparentée ne sont pas requises, alors la Sainte-Trinité s'appliquera.

La méthode `address()` doit retourner l'adresse (constante ou non) correspondant à une référence, alors que la méthode `max_size()` doit retourner le nombre d'instances de `T` pouvant être allouées avant qu'il ne reste plus de mémoire disponible.

```
pointer address(reference) const;
const_pointer
   address(const_reference) const;
size_type max_size() const noexcept;
```

Remarquez cette idée de traiter la consommation de mémoire en termes d'instances de `T` plutôt qu'en termes de *bytes*, qui distingue la métaphore privilégiée par les allocateurs de celle applicable à `new` et à `delete` (qui, eux, comptabilisent la mémoire un *byte* à la fois).

Les services d'allocation et de libération de mémoire réalisent aussi leur comptabilité eux en fonction du nombre d'objets, pas du nombre de *bytes*. La méthode `allocate()` alloue de la mémoire pour n instances de type `T` alors que la méthode `deallocate()` libère `n` blocs de taille `sizeof(T)` à partir d'une certaine adresse.

Le 2e paramètre à `allocate()` est un indice fourni par le code client quant à un bon endroit réaliser la prochaine allocation.

Ceci peut servir à des fins d'optimisation, comme dans le cas de séquences d'allocation et de libération d'objets ayant une durée de vie très limitée.

```
pointer allocate(size_type n,
   typename allocator<
      void
   >::const_pointer = {});
void deallocate(pointer p, size_type n);
```

Les services `construct()` et `destroy()` servent respectivement à construire (par copie, normalement par allocation positionnelle) et à finaliser un `T` à une adresse donnée.

```
void construct(pointer, const T&);
void destroy(pointer);
};
```

Enfin, il devrait être possible de comparer deux allocateurs génériques, quel que soit le type sur la base duquel s'applique la généricité (`==` devrait être une tautologie et `!=` devrait être une contradiction).

```
template <class T, class U>
   bool operator==(const allocator<T>&,
                   const allocator<U>&) noexcept;
template <class T, class U>
   bool operator!=(const allocator<T>&,
                   const allocator<U>&) noexcept;
```

Allocateurs depuis C++ 11

Les allocateurs ont été introduits dans C++ pour permettre au code client d'utiliser un conteneur tel que `std::list<T>` ou `std::vector<T>` tout en contrôlant les mécanismes d'allocation (et de libération) dynamique de mémoire.

Écrire un allocateur pour un conteneur avec C++ `03` est une tâche relativement complexe. Outre un ensemble d'alias (pour les types internes et publics que sont `value_type`, `pointer`, `reference`, `const_pointer`, `const_reference`, etc.), il faut écrire plusieurs méthodes dont :

Service	Rôle

`void construct(`
`  pointer p,const_reference r`
`)`
	Initialise la mémoire pointée par `p` avec une copie de l'objet auquel `r` réfère (typiquement : `new(static_cast<void*>(p))T(r)`)

`void destroy(pointer p)`	Finalise l'objet pointé par p (typiquement : `p->~T()`)

`pointer allocate(`
`  size_type n,pointer p = {}`
`)`
	Alloue l'espace pour `n` éléments contigus en mémoire (n'appelle aucun constructeur). Le paramètre `p` est un indice pour que le conteneur puisse suggérer à l'allocateur un endroit où commencer à chercher, ce qui peut être utile pour une pile par exemple.
	Implémentation possible :
	`auto p = malloc(n * sizeof(T));`
	`if (!p) throw bad_alloc{};`
	`return static_cast<T*>(p);`

`void deallocate(pointer p,size_type n)`	Libère l'espace pour `n` éléments débutant au point `p` (n'appelle aucun destructeur). Implémentation possible : `free(p);`
`typename rebind<U>::other`	Permet de cloner un type d'allocateur donné, dans la mesure où ce type est sans états (*Stateless*)
`pointer address(reference r)`	Retourne un « pointeur » sur `r` (typiquement : `return &r;`)

`const_pointer address(`
`  const_reference r`
`)`
	Retourne un « pointeur » sur `r` (typiquement : `return &r;`)

Notez que si les fonctions de bas niveau (`::operator new`, `std::malloc()`) calculent l'espace en *bytes*, les allocateurs sont typés et calculent l'espace en nombre d'éléments.

Un allocateur complet (mais simpliste, déléguant ses tâches vers `std::malloc()` et `std::free()`) pour C++ 03 (mais écrit pour C++ 11, pour alléger le tout) serait :

```
// ... inclusions et using ...
template <class T>
   struct tit_allocateur {
      using value_type = T;
      using pointer = T*;
      using const_pointer = const T*;
      using reference = T&;
      using const_reference = const T&;
      using size_type = std::size_t;
      using difference_type = std::ptrdiff_t;
      size_type max_size() {
         return std::numeric_limits<size_type>::max(); // bof
      }
      template <class U>
         struct rebind {
            using other = tit_allocateur<U>;
         };
      pointer address(reference r) {
         return &r;
      }
      const_pointer address(const_reference r) {
         return &r;
      }
      pointer allocate(size_type n) {
         auto p = static_cast<pointer>(malloc(n * sizeof(value_type)));
         if (!p) throw std::bad_alloc{};
         return p;
      }
      void deallocate(pointer p, size_type) {
         free(p);
      }
      void construct(pointer p, const_reference r) {
         new (static_cast<void*>(p)) value_type{r};
      }
      void destroy(const_pointer p) {
         p->~value_type();
      }
   };
template <class T, class U>
   bool operator==(const tit_allocateur<T>&, const tit_allocateur<U>&) { return true; }
template <class T, class U>
   bool operator!=(const tit_allocateur<T>&, const tit_allocateur<U>&) { return false; }
```

Depuis C++ 11, écrire un allocateur est plus simple, du moins de prime abord. Pour atteindre un résultat équivalent à ce qui précède, le code minimal requis est :

```
template <class T>
   struct allocateur_minimaliste {
      using value_type = T;
      allocateur_minimaliste(/*ctor args*/) = default;
      template <class U>
         allocateur_minimaliste(const allocateur_minimaliste<U>&) {
         }
      value_type* allocate(std::size_t n) {
         auto p = static_cast<value_type*>(std::malloc(n * sizeof(T)));
         if (!p) throw std::bad_alloc{};
         return p;
      }
      void deallocate(value_type *p, std::size_t) {
         free(p);
      }
   };
template <class T, class U>
   bool operator==(const allocateur_minimaliste<T>&, const allocateur_minimaliste<U>&) {
      return true;
   }
template <class T, class U>
   bool operator!=(const allocateur_minimaliste<T>&, const allocateur_minimaliste<U>&) {
      return false;
   }
```

Le reste de la classe est déduit par des traits (type `std::allocator_traits<T>`).

Exemple plus complet – allocateur séquentiel sur un tampon

Pour mieux comprendre le fonctionnement des allocateurs depuis C++ 11, et pour voir les avantages d'une spécialisation des allocateurs pour un programme donné, examinons le cas d'un allocateur séquentiel sur un tampon. Ses caractéristiques sont :

- il « alloue » la mémoire dans un tampon préalablement alloué, qui lui est fourni à la construction;
- la mémoire est allouée séquentiellement;
- la libération de mémoire est un *no-op*;
- si la mémoire du tampon est insuffisante, alors on délègue à `::operator new` pour allouer des ressources (puisque la libération est un *no-op*, on aura alors une fuite et mieux vaut la noter dans un fichier de journalisation).

Une implémentation possible serait :

```
template <class T>
   class buffered_sequential_allocator {
      char *buf, *cur;
      std::size_t size;
   public:
      using value_type = T;
      buffered_sequential_allocator(const buffered_sequential_allocator&) = default;
      buffered_sequential_allocator(buffered_sequential_allocator&&) = delete;
      buffered_sequential_allocator(char *buf, std::size_t size)
         : buf{ buf }, cur{ buf }, size{ size } {
      }
      template <class U>
         buffered_sequential_allocator(const buffered_sequential_allocator<U>& other) {
         assert(false);
      }
      value_type* allocate(std::size_t n) {
         if (cur + n * sizeof(value_type) >= buf + size)
            return static_cast<value_type*>(::operator new(n)); // ICI: fuite volontaire
         auto p = cur;
         cur += n * sizeof(value_type);
         return reinterpret_cast<value_type*>(p);
      }
      void deallocate(value_type *, std::size_t) {
         // no-op
      }
   };
```

Je pense que vous conviendrez que c'est relativement simple. Comparons maintenant trois allocateurs :

- le type `allocateur_minimaliste<T>`, décrit plus haut;
- le type `buffered_sequential_allocator<T>`, aussi décrit plus haut, et
- le type `std::allocator<T>`, offrant les services par défaut pour un conteneur donné.

Notre programme de test sera :

```
template <class C, class ... Args>
   void test(C & cont, int n, Args && ... args) {
      for(int i = 0; i < n; ++i)
         cont.emplace_back(std::forward<Args>(args)...);
   }
int main() {
   enum { N = 10'000 };
   enum : size_t { BUFSIZE = N * sizeof(double) };
   alignas (double) char buffer[BUFSIZE];
   buffered_sequential_allocator<double> ze_alloc (&buffer[0], BUFSIZE);
   auto avant = high_resolution_clock::now();
   {
      vector<double> v; // standard
      test(v, N, 3.5);
   }
   auto apres = high_resolution_clock::now();
   auto temps = apres - avant;
   cout << "Vanille : " << duration_cast<microseconds>(temps).count() << " us.\n";
   avant = high_resolution_clock::now();
   {
      vector<double, allocateur_minimaliste<double>> v;
      test(v, N, 3.5);
   }
   apres = high_resolution_clock::now();
   temps = apres - avant;
   avant = high_resolution_clock::now();
   cout << "Minimaliste : " << duration_cast<microseconds>(temps).count() << " us.\n";
   {
      vector<double, buffered_sequential_allocator<double>> v{ze_alloc};
      test(v, N, 3.5);
   }
   apres = high_resolution_clock::now();
   temps = apres - avant;
   cout << "Buffered-sequential : " << duration_cast<microseconds>(temps).count() << " us."
        << endl;
}
```

Pour chacun des trois types d'allocateurs, nous réalisons les mêmes opérations :

- ajouter `N` éléments à un vecteur;
- laisser le vecteur détruire ces éléments.

À l'exécution, nous obtiendrons quelque chose comme ce qui suit (les résultats varieront bien sûr selon les compilateurs, les ordinateurs, etc.) :

```
Vanille : 671 us.
Minimaliste : 232 us.
Buffered-sequential : 193 us.
```

Je pense que vous conviendrez que le jeu en vaut la chandelle[62].

[62] Notez que ce code, bien que légal, ne fonctionne pas sous Visual Studio 2015.

Exercices – Série 08

EX00 – Dans la section ***Exemple : superviser les allocations suspectes***, plus haut, serait-il possible de remplacer les λ qui rappellent les abonnées lors d'une allocation ou d'une libération par une seule classe nommée `Rappeler` sans entraîner de délai inacceptable? Introduire un `if` dans la définition de l'opérateur `()`, par exemple, est une solution inacceptable. Si oui, montrez comment; sinon, montrez pourquoi.

EX01 – Implémentez une classe dérivée de `Rappelable` telle qu'elle réagira si une libération de mémoire est faite pour une taille de mémoire jamais encore allouée dans le programme. À quoi cela pourrait-il servir?

EX02 – La classe `MesureDeTemps` proposée dans la section ***Exemple : une armée d'orques*** est un peu limitative du fait qu'elle signale le temps écoulé durant sa vie en émettant un message à la console. Suggérez au moins trois stratégies distinctes pour rendre ce modèle plus flexible et implémentez-les.

EX03 – Raffinez la méthode `allouer()` de `ArenaFixe` pour que celle-ci n'ait plus nécessairement à traverser le bassin à partir du début lors de chaque demande d'allocation. Testez rigoureusement votre solution.

EX04 – Modifiez `ArenaFixe` pour qu'elle recherche des blocs à partir du point de la dernière libération plutôt qu'à partir du point de la dernière allocation. Testez rigoureusement votre solution. Quel impact cela a-t-il sur la performance de votre mécanisme d'allocation de mémoire?

EX05 – Modifiez `ArenaFixe` pour qu'elle chercher à attribuer des blocs un à un à partir du début du bassin et des tableaux à partir de la fin du bassin. Testez rigoureusement votre solution. Quel impact cela a-t-il sur la performance de votre allocateur?

EX06 – Remarquez que la classe `ArenaFixe` présentée plus haut est générique à la fois sur la base de la taille d'un bloc et sur le nombre maximal de blocs. Un irritant de cette approche est que, dans le cas où deux classes distinctes `T` et `U` utilisant une instance d'`ArenaFixe` étaient telles que `sizeof(T)==sizeof(U)` et que le nombre maximal d'instances de `T` et de `U` était identique, alors les deux classes partageraient la même zone de mémoire. Corrigez cet irritant, puis testez et documentez votre solution.

EX07 – Implémentez un allocateur conforme au standard et qui allouerait la mémoire dans un `ArenaFixe` plutôt que sur le tas. Utilisez votre allocateur dans un conteneur de votre choix. Basez-vous sur la version d'`ArenaFixe` développée à `EX06`.

La réflexivité

Les programmeuses et les programmeurs choisissent un langage de programmation en fonction d'un projet donné pour plusieurs raisons, incluant :

- la capacité d'exprimer des entités logiques;
- la facilité avec laquelle ils parviendront à résoudre les problèmes auxquels ils font face;
- leur capacité d'utiliser ce langage pour construire des outils leur permettant de résoudre des problèmes plus complexes;
- l'aisance avec laquelle ils pourront exposer formellement par écrit une vision du monde; *etc*.

Ceci dégage une vision du langage[63] comme outil pour, entre autres choses, concevoir d'autres outils. Comme canevas pour organiser la pensée.

L'avènement du modèle OO a mené les informaticien(ne)s à :

- penser les programmes en terme d'objets liés structurellement ou associés entre eux;
- définir les entités logicielles en des structures cohérentes regroupant données et opérations, capables d'existence autonome;
- réfléchir aux relations entre entités logicielles sous forme de réseaux, de dynamiques avec des nœuds à vision subjective; *etc*.

Ce faisant, l'approche OO a entraîné des recherches plus poussées dans la relation entre les objets manipulés et les structures des langages avec lesquels ces objets sont construits. Le vocabulaire de base, qui réfère à l'héritage, à la composition, à l'agrégation et à l'abstraction, en est témoin. Dans la même veine, mais dans une optique plus riche, les schémas de conception décrivent des approches de conception logicielle basées sur l'expression de relations entre objets pour résoudre de larges catégories de problèmes.

Plusieurs langages poussent la conceptualisation OO jusqu'à permettre la représentation du langage à même les structures du langage. Par exemple, pour plusieurs pour lesquels les liens dynamiques sont privilégiés (dont Smalltalk, C# et Java[64]), il est possible :

- de charger en mémoire une classe à partir de son nom;
- de consulter dynamiquement la liste des méthodes d'instance exposées par une classe, incluant la liste de ses constructeurs;
- de consulter par programmation la liste des paramètres à une méthode donnée;
- d'invoquer une méthode à partir de sa représentation dans le langage; *etc*.

[63] Le mot langage ici ne réfère d'ailleurs pas seulement aux langages de programmation. Tout langage est susceptible d'être utilisé à ces fins.

[64] Il se trouve que les hiérarchies de classes servant de bibliothèques standards à Java et à C# sont extrêmement similaires, du fait qu'elles ont toutes deux été conçues à partir d'un même patron, conçu par le groupe NIST (http://www.nist.gov/).

Réflexitivité statique ou dynamique?

Dans la plupart des langages où elle est appliquée, la réflexivité est une mécanique dynamique. Vous noterez d'ailleurs des approches se rapprochant beaucoup de la réflexivité dans certains langages dynamiques comme ceux souvent utilisés dans les applications Web (JavaScript, Perl, Python et d'autres langages de cette nature).

En C++, ce qui se rapproche le plus de la réflexivité serait la métaprogrammation, qui est une introspection statique sur la nature de types à des fins algorithmiques. Les traits (voir ***Traits, polymorphisme statique et bases de métaprogrammation***) font aussi partie de ces mécanismes de réflexivité statique. Des travaux sont en cours pour accroître l'offre de réflexivité de C++ en vue de C++ 17, mais ce qui est actuellement sur la table est plutôt humble.

La présente section se penchera strictement sur le volet dynamique; la métaprogrammation est un sujet immense qui aura sa propre section dans un document ultérieur.

Pouvoir agir ainsi implique avoir accès à une représentation OO non seulement de ce qui est fait avec le langage, mais aussi du langage lui-même, au moins dans une certaine mesure :

- avoir une classe pour représenter l'idée de classe;
- avoir une classe pour représenter l'idée d'instance;
- avoir une classe pour représenter chaque type primitif;
- avoir une classe pour représenter l'idée de méthode, dans le but de pouvoir invoquer dynamiquement une méthode découverte à l'exécution plutôt que connue à la compilation;
- avoir une classe pour représenter l'idée de constructeur, dans le but de pouvoir invoquer l'instanciation d'une classe dont on aura découvert dynamiquement l'existence;
- pouvoir décoder dynamiquement le caractère privé, public, abstrait… de tout membre; *etc*.

⇒ On nomme **réflexivité** la capacité d'introspection qu'offrent certains langages de programmation OO et par laquelle les structures du langage sont représentées dans le langage et manipulables, à divers degrés, par programmation.

Une définition alternative, mais moins répandue de la réflexivité dirait qu'il s'agit de la capacité pour un programme de modifier certaines caractéristiques de l'implémentation du langage lui-même, par instance ou par classe :

- schèmes d'allocation de mémoire
- stratégies de synchronisation
- manière d'invoquer une méthode
- stratégies de construction d'objets, *etc*.

En fait, la réflexivité et le modèle OO sont tellement liés l'un à l'autre qu'on en est venu à considérer la réflexivité comme faisant partie des atouts d'une certaine catégorie de langages OO.

Certaines applications de la POO, comme l'introduction au sens entendu en POA, sont impossibles ans support de la réflexivité. Nous survolerons la POA dans la section ***Programmation orientée aspect (POA)***, plus bas.

L'une des stratégies les plus répandues pour offrir la réflexivité dans un langage donné est d'utiliser des *métaobjets*, donc des objets représentant des concepts de haut niveau, incluant le concept de classe lui-même (ce dont nous discuterons un peu plus loin dans la section ***Les métaclasses***).

L'un des textes séminaux en ce sens[65] décrit le rôle des classes comme offrant des services d'instanciation d'instances et décrit aussi le rôle des métaclasses comme offrant des services d'instanciation pour les classes, tout en offrant un schème par lequel il est possible de construire une hiérarchie arbitrairement riche de classes et de métaclasses.

Éléments constitutifs de la réflexivité

La réflexivité peut être séparée en deux idées distinctes, soit **l'introspection**, qui permet l'examen des structures du langage ou des objets qui peuplent un programme, et **l'intercession**, qui permet la modification de ces structures (certains parleront alors d'***introduction***).

On verra parfois aussi le terme ***réification*** (en argot contemporain, on dirait sans doute *chosification*) pour nommer le mécanisme par lequel les états d'un langage ou d'un programme sont représentés sous forme d'objets.

Ce terme est particulièrement à propos dans le cas d'invocation dynamique de constructeurs.

En pratique, l'introspection est rencontrée plus fréquemment que l'intercession, entre autres parce qu'elle est moins dangereuse, donc moins controversée.

Situer la réflexivité

La réflexivité est une capacité introspective structurelle, servant à comprendre la manière dont sont construits les objets, leurs opérations, leurs attributs, mais n'est pas (dans les langages les plus répandus, du moins) une capacité de bris d'encapsulation. On ne peut habituellement pas utiliser la réflexivité pour extraire de force la valeur d'un attribut privé dans un objet, mais on peut, par réflexivité, connaître la présence et le type des attributs d'une classe donnée, ou invoquer une méthode publique d'une instance.

En ce sens, la réflexivité s'apparente plus à la syntaxe d'un langage qu'à sa sémantique. Elle représente la capacité d'un langage OO *d'examiner* (et, parfois, *de modifier*) sa forme, pas les valeurs des instances qui peuplent les systèmes qu'il permet d'exprimer. La réflexivité représente aussi une manière homogène, dans un système informatique, d'intégrer de nouveaux objets et de nouveaux types de manière dynamique.

En Java, par exemple, on peut charger une classe par son nom, à partir du nom d'un fichier préalablement compilé. Cette fonctionnalité permet, entre autres, de charger certains pilotes, vraiment requis, parmi ceux possibles pour un programme accédant à une base de données.

Comme tout mécanisme hautement dynamique, la flexibilité gagnée par la réflexivité entraîne un coût en temps et en mémoire. Rien n'est gratuit.

[65] Cointe, P. : *Metaclasses are First Class: the ObjVlisp model*, OOPSLA '87 Proceedings, 1987, Orlando, Florida, SIGPLAN Notices, vol. 22, no 12, pp. 156-167.

Applications de la réflexivité

D'un langage OO à l'autre, les applications de la réflexivité varient. Examinons, à titre d'illustration, quelques exemples concrets de recours à la réflexivité, que ces exemples soient applicables en Java, dans les langages .NET, ou dans les deux cas.

Sécurisation de conversions risquées

La plupart des langages OO offrent une forme de réflexivité, si simple soit-elle, en particulier pour permettre l'inférence dynamique de types, qui est un questionnement à l'exécution sur la nature de certaines entités manipulées indirectement.

À titre d'exemple, les langages limitant l'héritage à sa déclinaison simple (si on ne leur ajoute pas la généricité) doivent ramener les opérations abstraites à ce qui tient pour eux le rôle de racine unique, typiquement la classe Object. Ce faisant, ces langages ont souvent recours à ce qu'on nomme en anglais les *Downcasts*, soit le transtypage d'une indirection (référence ou pointeur) d'une classe parent en l'une de ses classes enfants.

Sous Java, la conversion d'un parent en l'un de ses enfants lève, lorsque la conversion est impropre (lorsqu'on se trompe d'enfant) un ClassCastException. Dans les langages .NET, l'exception levée dans un tel cas est un InvalidCastException.

En C++, l'inférence dynamique de types se fait à travers l'opérateur dynamic_cast, qui ne s'applique qu'à des types polymorphiques et nécessite l'inclusion dans un programme d'information supplémentaire (le RTTI, pour *Run-Time Type Information*). Cela dit, dû à l'héritage multiple et à l'absence de racine unique, l'inférence dynamique de types peut *presque* toujours être évitée en C++. Un compilateur C++ cherche en général à minimiser le travail nécessaire à *l'exécution*, donc à faire le maximum possible lors de *la compilation*, et les programmeuses comme les programmeurs C++ font de même.

Lorsqu'on y a recours, dynamic_cast retourne 0 lors d'une tentative de conversion illégale sur un pointeur ou lève std::bad_cast lors d'une tentative de conversion illégale sur une référence.

En C++, dû au support à l'héritage multiple, l'inférence dynamique de types permet non seulement de réaliser des *Downcasts* mais aussi de réaliser des *Crosscasts*, soit la conversion d'une indirection sur une instance d'une classe donnée en un pointeur d'une classe ayant une distante parenté structurelle (une classe sœur, par exemple).

```
class X { };
class Y0 : public X { };
class Y1 : public X { };
int main() {
   Y0 y0;
   X &p = y0;
   // bad_cast (illégal!)
   Y1 &q = dynamic_cast<Y1&>(p);
}
```

Support à la sérialisation

La réflexivité offre un support des plus utiles aux moteurs désireux d'implémenter la sérialisation. En effet, écrire un objet sur un flux en revient alors à y écrire la représentation lisible de sa structure interne, qui est faite d'objets… à plus forte raison si tout objet peut s'exprimer de manière lisible sur un flux (ce que permettent la plupart des moteurs OO).

Reconstituer *à même une machine virtuelle* un objet extrait d'un flux devient aussi chose beaucoup plus simple si les objets sont *a priori* autodescriptifs. S'il est possible d'écrire les objets sur un flux de manière à ce que la description de chacun soit précédée d'un indicateur quant à sa nature, il est aussi possible d'écrire un moteur général de désérialisation qui consommera les indicateurs puis reconstruira les objets correspondants.

Ceci présume bien entendu que la représentation des objets sur un flux permet de les reconstituer par la suite. Prenons note que la reconstruction d'un objet, pris isolément, à partir d'une représentation sérialisée est une tâche moins complexe que celle de reconstituer le système d'objets dans lequel il habitait en mémoire.

Nous y reviendrons sur les questions propres à la sérialisation et à la persistance des objets dans la section ***Sérialisation et persistance***, plus bas.

Allégement de la machine virtuelle

Supporter la réflexivité dans les langages reposant sur une machine virtuelle, comme le sont Java et les langages .NET, a pour effet secondaire non négligeable d'alléger la charge de la machine virtuelle en tant que telle.

En effet, si les classes moins susceptibles d'être utilisées par tout programme, comme par exemple des classes servant de pilotes pour l'accès aux BD, peuvent être chargées et analysées à l'exécution seulement, alors elles ne coûteront des ressources au programme que si cela s'avère véritablement nécessaire.

Le modèle préconisé par Java implique d'ailleurs qu'une classe ne soit chargée en mémoire que lors du premier accès à l'un de ses membres, quel qu'il soit.

En java, la méthode de classe **`forName()`** de la classe **`Class`** prend un nom de classe[66] et en charge la représentation OO en mémoire, dans la JVM, pour que cette classe puisse ensuite être utilisée par le programme. Habituellement, on chargera ainsi un dérivé d'une racine polymorphique choisie et on procédera ensuite par polymorphisme dans le programme.

```
final String SOURCE = "jdbc:odbc:Poolscan",
             PILOTE="sun.jdbc.odbc.JdbcOdbcDriver";
try {
   Class.forName(PILOTE);
   Connection connexionBD =
      DriverManager.getConnection(SOURCE);
} catch (ClassNotFoundException cnfe) {
  System.err.println(cnfe.getMessage());
} catch (SQLException sqle) {
   System.err.println(sqle.getMessage());
}
```

Dans les langages .NET, la méthode **`AppDomain.CreateInstanceAndUnwrap()`** fait à peu près la même chose. On peut aussi charger un assemblage (DLL, EXE) du monde .NET avec la méthode de classe **`AppDomain.Load()`**.

[66] En Java, les règles des paquetages et des noms de classes font en sorte que ces noms correspondent à un nom de fichier dans une structure de répertoires.

Développer des IDE de type RAD

Les environnements intégrés de développement (en anglais : *Integrated Development Environment*, ou IDE) offrant des fonctionnalités de prototypage et de développement rapide (en anglais : *Rapid Application Development*, ou simplement RAD[67]) qui permettent entre autres de positionner des contrôles[68] et de modifier leurs propriétés (position, dimension, texte affiché par défaut, *etc*.) profitent aussi beaucoup de la réflexivité.

En effet, les langages offrant un support pour la réflexivité permettent aux IDE de type RAD d'extraire des contrôles qu'ils manipulent la liste des accesseurs et des mutateurs (ou des propriétés) pour permettre leur manipulation simplifiée par un humain. Pensez par exemple à la fenêtre de propriétés de *Visual Studio* ou d'*Eclipse*.

Un bon côté des propriétés

Tel que mentionné dans [POOv00], si le formalisme des propriétés a une réelle utilité, c'est bien parce qu'il présente les propriétés comme des entités distinctes des autres méthodes à même le langage. Ceci permet, au besoin, de *ne lister qu'elles et elles seules*.

Les outils RAD peuvent profiter de cette particularité, et construire les fenêtres de propriétés d'un objet donné à partir des propriétés seules. Ceci permet, par exemple, de reconnaître les propriétés en lecture seule par l'absence de propriété set.

Sans propriétés, il demeure possible de construire par réflexivité des IDE de type RAD, dans la mesure où un formalisme autre (ou une discipline de programmation stricte) permet de reconnaître les accesseurs et les mutateurs de premier ordre ou ce qui en tient lieu.

Un bon exemple d'approche disciplinaire (plutôt que langage, comme dans le cas des propriétés) est la convention Java (en particulier avec les *Beans*) de nommer getX() un accesseur public pour l'attribut X, du moins pour que celui-ci soit reconnu comme tel par la plupart des IDE faits pour Java.

Les propriétés ayant un support particulier du langage de programmation, elles proposent aux programmeurs un cadre de travail plus strict, reposant moins sur une convention que sur un formalisme réel.

[67] Il est venu à mon oreille en 2005 que le mot RAD serait parfois présenté comme une philosophie plutôt que comme une catégorie d'outil. Si RAD représente pour vous une catégorie philosophique, sachez que la philosophie est pour moi le ***prototypage rapide*** et que RAD signifie pour moi un outil permettant de mettre en œuvre le prototypage rapide.

[68] En bref, et de manière simplifiée, disons que les contrôles sont des objets à contrepartie visuelle, comme des boutons de commande ou des listes déroulantes, avec lesquels on peut interagir sur une interface personne/ machine.

Raisonnement métacognitif et métalangage

De manière générale, la principale force de la réflexivité est de permettre un raisonnement algorithmique sur le langage et sur le programme en cours d'exécution. Selon les qualités des implémentations, il sera possible de naviguer et d'évaluer la structure des objets et des relations entre eux, celle des classes auxquelles ils appartiennent, celles des structures qui mènent à la construction des classes comme à celle des instances, etc.

Le développement de langages spécifiques aux domaines d'application, en anglais *Domain-Specific Languages* (DSL), peuvent profiter de ces facilités langagières pour écrire des langages à part entière à l'intérieur des langages, ou des programmes raisonnant sur des programmes de manière générale plutôt que spécifique.

Selon les déclinaisons de la réflexivité, ceci peut mener à des applications riches et variées de la réflexivité, par exemple :

- concevoir un débogueur capable d'étudier des objets *a priori* inconnus, de présenter leur nom et leurs états et, si possible, de modifier ces états (surtout pendant la phase de développement du programme);
- concevoir un profileur capable d'étudier, de manière générale, le comportement dynamique d'un programme et d'en tirer des diagnostics;
- mettre au point de la POA (voir plus loin), soit des opérations transversales et découplées des objets eux-mêmes. Un exemple d'application de la POA serait de demander une autorisation humaine avant toute invocation de méthode dont la signature respecte certaines règles, ce qui est plus facile à implémenter s'il est possible de traverser dynamiquement la structure d'un programme, de repérer les opérations visées, et d'injecter le code requis aux endroits appropriés;
- développer un éditeur RAD (voir plus haut);
- rédiger des inspecteurs surveillant certains comportements suspects des programmes; *etc*.

Réflexions sur la réflexivité

Dans une optique OO, que penser de la réflexivité? Il y a plusieurs opinions sur le sujet, on s'en doutera, et le niveau d'abstraction du concept peut compliquer son analyse. Pour une thématique comme la réflexivité, on trouve plus d'utilisateurs que de penseurs.

Les tenants de la position selon laquelle la réflexivité est un concept à conserver, à soutenir et à mousser dans le curriculum OO basent (au moins en partie) leur prise de position sur les arguments suivants :

- du point de vue de **l'élégance** conceptuelle ou scientifique, on doit voir d'un œil positif qu'un langage se décrive lui-même selon les mêmes règles que celles par lesquelles il exprime et produit ses propres concepts. La capacité d'autoréférence est un signe d'intelligence, après tout, et l'autodescription peut être vu comme un signe de complétude interne;
- du point de vue de **l'extensibilité**, la réflexivité (au sens dynamique) permet d'ajouter dynamiquement à tout programme, et au langage lui-même, sans compiler à nouveau le programme ni solliciter des mécanismes dynamiques propres à la plateforme[69]. Un langage capable de réflexivité peut, en théorie, croître et s'adapter plus facilement (et de manière bien plus élégante) qu'un langage pour lequel le support à la réflexivité est limité ou absent;
- dans la même veine, **mettre à jour** un produit déjà construit est chose beaucoup plus simple si le langage dans lequel ce produit a été conçu permet la mise à jour dynamique ou le remplacement pur et dur des objets qui en font partie, incluant l'insertion dynamique de nouveaux objets dans un produit existant. Ceci permet, en théorie, d'éviter les redémarrages et les bris, dans la mesure où – pour les applications pragmatiques, du moins – la racine polymorphique des objets pour lesquels on exploite la réflexivité a été réfléchie avec soin (cela dit, en toute honnêteté, rien ne remplace un design solide);
- tel qu'indiqué plus haut, les plateformes qui comprennent de vastes bibliothèques de classes (des *Frameworks*, pour utiliser l'expression anglophile consacrée), comme les plateformes Java et .NET, profitent de la réflexivité pour assurer un **allégement de la charge** qu'ils imposent sur les ressources du système;
- toujours tel que mentionné précédemment, la réflexivité facilite l'implémentation formelle d'un **moteur de sérialisation** complet et évite le recours à des stratégies *ad hoc*;
- la réflexivité facilite la **génération de documentation** automatisée des classes à partir de leur structure même. Là où générer de la documentation automatisée dans plusieurs langages implique une analyse du texte des programmes, faire de même pour un langage supportant la réflexivité peut se faire en chargeant les classes en mémoire et en dialoguant avec elles, ou encore en s'insérant dans un programme en cours d'exécution et en interagissant avec les classes qui en font partie.

L'outil standard Javadoc du monde Java exploite massivement la réflexivité dans le cadre de son travail. Plusieurs autres types d'outils (débogueurs, profileurs, optimisateurs, *dissimuleurs*, *etc*.[70]) peuvent profiter de cette capacité, d'ailleurs.

[69] Sans charger des .DLL sous Windows et des .SO sous Linux.

[70] En anglais, dans l'ordre : *Debugger*, *Profiler*, *Optimizer* et *Obfuscator*.

D'un autre côté, ceux selon qui la réflexivité devrait être vue soit comme un concept OO accessoire plutôt que fondamental, ou selon qui la réflexivité est dangereuse, donc à restreindre, contraindre ou simplement (pour les plus radicaux) à bannir, rétorqueront entre autres que :

- la réflexivité, même dans une optique simplifiée de consultation seule, représente fondamentalement un **bris d'encapsulation**. Extraire de l'information descriptive de la structure d'un objet ou d'une classe, même compilée, expose l'information qu'on pourrait trouver dans la déclaration d'une classe (par analogie : dans le fichier d'en-tête d'une classe C++) qui ne serait pas cachée derrière un voile d'abstraction. Un tel bris d'encapsulation s'apparente, d'un point de vue structurel, à un ***bris de confidentialité***[71];
- si la réflexivité rend le langage deviendrait malléable dans sa structure même (voir les puces suivantes), plusieurs estiment que la **sécurité** devient compromise dans les systèmes développés à l'aide de ces langages ou utilisant des produits développés à l'aide de ces langages. Il est déjà difficile de prédire le comportement de systèmes moins malléables, alors on peut comprendre les spécialistes d'être craintifs face au prospect de voir surgir des systèmes informatiques malléables au point de s'*automodifier*; des programmes d'allure *a priori* innocente mais qui se transformeraient éventuellement en virus des plus vicieux[72];
- la POA (voir la section à cet effet) permet d'insérer des opérations de manière transversale en spécifiant des critères systémiques plutôt que subjectifs (par exemple, pour tout objet d'un système pour lequel un mutateur est invoqué, sérialiser l'objet avant invocation du mutateur en question); certaines technologies de POA exploitent la réflexivité pour y arriver[73]. Certaines applications de la POA appliquent le mécanisme d'**introduction** ou d'**intercession**, par lequel un *aspect*, en opérant sur un objet, pourrait aussi y insérer un ou plusieurs membres qui n'y existaient pas auparavant, donc utiliser la réflexivité d'une manière modifiant l'objet. Bien qu'il y ait des applications manifestes à pouvoir modifier structurellement des instances ou des classes, il s'agit d'un **bris d'encapsulation**, cette fois au sens de ***bris d'intégrité structurelle***, du moins dans la mesure où un objet autre que l'objet sur lequel s'opère l'introduction devient capable de modifier structurellement un autre objet sans son consentement. Évidemment, si les modifications structurelles faites *sur* un objet sont faites *par* cet objet[74], la question se pose autrement;

---

[71] Merci à **Pierre Prud'homme** pour cette jolie expression!

[72] Des systèmes autoréférentiels capables de se modifier eux-mêmes ne seraient pas si novateurs, il faut bien le souligner. À une époque pas si lointaine, lorsque les ressources se faisaient rares, il arrivait que de tels systèmes soient conçus par des esprits drôlement tournés pour qu'un même programme puisse faire plus de choses qu'il n'aurait semblé être capable de faire au préalable.

[73] La réflexivité n'est pas *nécessaire* pour appliquer des techniques de POA à un programme, tout comme la réflexion n'est pas *nécessaire* pour générer de la documentation primitive à partir du texte d'un programme. Dans les deux cas, un utilitaire de type macro peut procéder à partir de spécifications et du texte d'un programme pour en arriver au même résultat, ce qui donne une application statique – à la compilation, ou du moins avant exécution – plutôt que dynamique – à l'exécution – de la POA ou de la génération automatique de la documentation.

[74] … du moins, *par l'objet tel qu'il était avant la modification en question*, puisque ce type de modification peut, en fait, aller même jusqu'à changer le concept même d'identité chez l'objet!

- dans tous ces cas, comme on peut s'y attendre d'un point de vue OO, les bris d'encapsulation en lecture ou en écriture **privent indirectement chaque objet de sa capacité à garantir sa propre intégrité** du début à la fin de son existence, et brisent donc fondamentalement l'un des piliers de l'approche OO. La situation est pire si l'intercession est appliquée que si la seule introspection est applicable;
- enfin, la lisibilité structurelle absolue des objets dans un système implémentant la réflexivité pose un très sérieux problème de **propriété intellectuelle**. Qu'est-ce qui empêche un individu malhonnête, capable d'analyser dynamiquement la structure du code, d'écrire un programme qui en clone un autre, et de le vendre comme s'il lui appartenait? De remplacer le compilateur utilisé pour développer ses propres programmes puis de refuser par la suite de payer pour l'utiliser? D'introduire dynamiquement un générateur de virus dans le code d'un compilateur pour que les autres développeurs s'en servant obtiennent des programmes infectés à leur insu? Certains diront que procéder par code ouvert est la clé ici, mais les différends philosophiques à ce sujet sont nombreux…

Réflexivité dynamique et C++

À première vue, il y a un conflit idéologique entre l'idée même de réflexivité (au sens dynamique[75]) et l'utilisation d'un langage comme C++, où les principes premiers énoncent qu'un programmeur ne doit payer que pour ce qu'il utilise. Traîner un surcroît de métadonnées à l'exécution implique un coût en espace mémoire pouvant devenir important pour de gros systèmes, et on voit mal comment un langage se voulant capable de permettre la rédaction efficace de systèmes embarqués pourrait aller dans cette direction. Il demeure qu'un groupe d'étude s'intéressant spécifiquement à cette question est actif en vue de C++ 20.

Cela dit, la réflexivité est une thématique importante de la programmation contemporaine. Le besoin de métadonnées est omniprésent dans Internet, dans les banques de données, pour les éditeurs, les systèmes répartis, *etc*. En retour, un programme trop gourmand en espace mémoire est un programme plus lent qu'il ne le devrait, utilisant moins bien l'antémémoire du processeur, et qui ne peut être exécuté sur les plateformes munies de contraintes physiques astreignantes. Un langage comme C++ ne peut payer un tel prix, alors que les langages comme Java et les langages .NET, qui ne visent pas une telle universalité, le peuvent.

Le travail connexe à la réflexivité dynamique et aux métadonnées qui est réalisé autour de C++ porte sur les langages SELL, pour *Semantically Enhanced Library Languages*[76]. L'idée est, en gros, de faire générer deux structures par le compilateur : le programme en tant que tel d'un côté, et la structure sémantique consultable dynamiquement comme entité conjointe.

Le programme résultant ne sera pas nécessairement malléable à l'exécution mais restera sujet à introspection et à navigation dynamique.

[75] Tel que mentionné précédemment, la réflexion statique existe déjà avec la métaprogrammation.

[76] Pour en savoir plus, je vous invite à consulter les articles de **Bjarne Stroustrup** et de **Gabriel Dos Reis** disponibles sur [StrouSellR] et [StrouSellH].

En vue de C++ 20 ou de C++ 23

Un groupe d'étude du comité de standardisation de C++, le groupe SG7 piloté par **Chandler Carruth**, œuvre à la mise en place de mécanismes permettant de réaliser de la réflexivité avec C++. Le langage ayant un fort penchant pour les opérations à coût zéro, ceci teinte la saveur des propositions qui sont discutées dans ce groupe.

Une avenue prometteuse en semble être l'ajout d'un opérateur statique, nommé `reflexpr`, qui permettrait de tirer des métadonnées statiques d'une expression. L'intention serait de permettre, par exemple, de parcourir efficacement les valeurs d'un type énuméré, ou de connaître la liste des noms exposés par une classe donnée.

Plusieurs questions difficiles sont posées par ces travaux. Par exemple, peut-on représenter efficacement la signature d'une fonction surchargée? Mais les travaux vont bon train, et il y a de l'espoir pour ce qui est d'obtenir ce mécanisme vers `2020` ou `2023`.

Dans d'autres langages

La réflexivité est offerte de différentes manières et à différents niveaux (dynamique, statique) en fonction des langages de programmation et des philosophies.

En C++, la programmation générique, sous la forme de la ***métaprogrammation***, permet une exploration statique de la nature des objets. Nous y reviendrons.

En Java, toute classe dérive de près ou de loin de **`Object`**, qui expose la méthode virtuelle nommée **`getClass()`**. Cette méthode d'instance retourne une instance de la classe **`Class`** décrivant la structure de la classe dont l'instance propriétaire est une instance, et à partir de laquelle une introspection devient possible.

Exemple concret : le code Java suivant chargera dynamiquement la classe `X`, instanciera un `X` par défaut et affichera ce que retournera sa méthode `f()` :

```
import java.lang.reflect.Constructor;
import java.lang.reflect.Method;
import java.lang.reflect.InvocationTargetException;
public class Z {
   public static void main(String[] args) {
      try {
         Class classX = Class.forName("X");
         Constructor constrX = classX.getConstructor();
         Object objX = constrX.newInstance();
         Method methf = classX.getMethod("f");
         System.out.println("Résultat de l'invocation de f() sur un X par défaut: " +
                            methf.invoke(objX));
      } catch (ClassNotFoundException cnfe) {
         System.err.println("Incapable de charger la classe X. Détails: " + cnfe);
      } catch (NoSuchMethodException nsme) {
         System.err.println("Incapable de trouver la méthode. Détails: " + nsme);
      } catch (SecurityException se) {
         System.err.println("Problème d'accès. Détails: " + se);
      } catch (InstantiationException ie) {
         System.err.println("Incapable d'instancier X. Détails: " + ie);
      } catch (IllegalAccessException iae) {
         System.err.println("Incapable d'invoquer f(). Détails: " + iae);
      } catch (InvocationTargetException ite) {
         System.err.println("Exception levée par une méthode. Détails: " + ite);
      }
   }
}
```

Notez que la prise en charge des exceptions, en Java, est obligatoire, ce qui explique le bloc `try` et grand nombre de clauses `catch` l'accompagnant dans cet exemple.

Dans les langages `.NET`, on trouve des stratégies semblables avec la méthode de classe **`GetType()`** de la classe **`Type`**.

Exemple concret : le code C# suivant chargera dynamiquement la classe `X`, instanciera un `X` par défaut et affichera ce que retournera sa méthode `f()` :

```
using System.Reflection;
namespace zz
{
   class Program
   {
      public static void Main(string[] args)
      {
         Assembly aX = Assembly.Load("zz");
         System.Type classX = aX.GetType("zz.X");
         System.Reflection.ConstructorInfo constrX =
             classX.GetConstructor(System.Type.EmptyTypes);
         object objX = constrX.Invoke(null);
         System.Reflection.MethodInfo methf = classX.GetMethod("f");
         System.Console.WriteLine
            ("Résultat de l'invocation de f() sur un X par défaut: {0}",
             methf.Invoke (objX, null));
      }
   }
}
```

Remarquez que le nom de la classe doit être qualifié par son espace nommé. Le nom doit être chargé à partir d'un assemblage, et une liste vide de paramètres à l'invocation de `GetConstructor()` ne peut être représenté par `null`; il faut utiliser l'attribut de classe qualifié `readonly` et nommé `EmptyTypes` qui constitue un tableau vide de types. En retour, la méthode `Invoke()` accepte `null` pour une liste vide de paramètres.

Exemple concret : le code `VB.NET` suivant chargera dynamiquement la classe `X`, instanciera un `X` par défaut et affichera ce que retournera sa méthode `f()` :

```
Imports System.Reflection
Module Test
   Public Sub Main(ByVal args As String())
      Dim aX As Assembly = Assembly.Load("zz")
      Dim classX As System.Type = aX.GetType("zz.X")
      Dim constrX As System.Reflection.ConstructorInfo = _
            classX.GetConstructor(System.Type.EmptyTypes)
      Dim objX As Object = constrX.Invoke(Nothing)
      Dim methf As System.Reflection.MethodInfo = classX.GetMethod("f")
      System.Console.WriteLine _
            ("Résultat de l'invocation de f() sur un X par défaut: {0}", _
             methf.Invoke(objX, Nothing))
   End Sub
End Module
```

Les métaclasses[77]

Existe-t-il des concepts dont les instances sont d'autres concepts plutôt que des individus? Est-ce bien ce qu'on décrit quand on parle d'héritage, donc de spécialisation conceptuelle, du général au particulier? Combien de niveaux de concepts décrivant d'autres concepts devrions-nous avoir pour vraiment décrire notre pensée, pour résoudre tous les problèmes structurels en conceptuels que nous sommes susceptibles de confronter? Est-ce que supporter une infinité de niveaux conceptuels (concepts de concepts de concepts de…) est une nécessité pour un modèle OO qui se voudrait complet?

Voilà des questions à la fois pertinentes et pour lesquelles les applications concrètes restent bien souvent à réfléchir. L'exemple classique de la problématique est trompeur :

- imaginons le concept Espèce;
- imaginons le concept Aigle;
- imaginons Fred, un aigle bien précis.

Dans l'optique de ce cours, on tendrait à dire qu'Aigle est une spécialisation d'Espèce, donc qu'Aigle dérive d'Espèce, alors que Fred serait une instance d'Aigle, donc indirectement une instance d'Espèce.

Les logiciens feront remarquer, et c'est intéressant, que ce point de vue (pragmatique, qui donne des résultats concrets) est un point de vue conceptuellement floué. En effet, s'il est vrai qu'Aigle est un cas particulier d'Espèce, et s'il est vrai que Fred est un cas particulier d'Aigle, il est faux de dire que Fred est un cas particulier d'Espèce. L'habitude que nous avons de procéder par polymorphisme pose ici problème si on veut utiliser un pointeur d'Espèce comme racine polymorphique menant à l'instance Fred du fait que Fred, conceptuellement, n'est pas un cas particulier d'Espèce.

Cette remarque sera appuyée par des schémas et par un discours sur des caractéristiques de la logique des prédicats du premier ordre et sur la logique d'ordre supérieur. On mettra de l'avant que le concept d'instance est lié à celui d'individu, alors que celui de classe s'apparente au concept d'ensemble, ce qui nous amène à réaliser qu'Aigle est un individu du point de vue de l'ensemble Espèce.

[77] Cette section est à la fois importante, pertinente, peu applicables dans la majorité des langages de programmation, essentielle pour les efforts de standardisation tels ceux pris en charge par les comités ISO, et plutôt difficile à digérer. Pour des articles intéressants sur le sujet, dont ceux dont est tiré l'exemple initial de cette section, voir :

- http://www.cs.vassar.edu/faculty/welty/papers/p24-welty.pdf;
- http://www.cs.vassar.edu/faculty/welty/papers/instances/instances.pdf;
- http://research.sun.com/projects/plrg/core-calculus.pdf; ou
- http://research.sun.com/projects/plrg/p109-allen.pdf.

On peut être tenté de critiquer le choix architectural de la hiérarchie (quoique le choix ne soit pas arrêté à savoir si on devrait parler de hiérarchie ou non dans ce cas) proposée ici, mais ce serait une critique discutable. Il est raisonnable de vouloir exprimer la relation entre `Espèce` et `Aigle`, et il est raisonnable de vouloir exprimer la relation entre `Aigle` et `Fred`. Ces trois concepts sont apparentés, et on peut très bien imaginer des programmes qui soient tentés de les exploiter tous les trois.

D'autres exemples classiques du genre :

- les idées de `Classe`, `Forme`, `Rectangle` et de petit rectangle bleu;
- les idées de `Dimension`, `Temps`, `Longueur`, `Mètre` et `Minute` (les deux derniers étant des unités de dimensions spécifiques);
- les idées que sont `Abstraction`, `Idée` et invention du téléphone; *etc*.

Des classes qui engendrent des classes

Tous ces groupements ont des caractéristiques communes :

- ils mettent en relation deux abstractions distinctes, l'une plus abstraite que l'autre (pensez à `Espèce` et à `Aigle`);
- ils mettent en relation une abstraction avec un individu (`Aigle` et `Fred`); et
- dans chaque cas, lier les deux abstractions par héritage entraînerait une faute logique puisque l'individu n'est pas un cas particulier de l'abstraction la plus élevée des deux (`Fred` n'est pas une `Espèce`, mais `Aigle` est une `Espèce` et `Fred` est un `Aigle`).

Ces quelques cas, ces quelques remarques peuvent ressembler à des fientes de mouches, à des préoccupations de logiciennes et de logiciens qui devaient retomber sur terre et s'occuper de vraies applications du modèle `OO`, mais il faut être attentif à la remarque. Les questions de fondement sont des questions douloureuses, qui nous forcent à réfléchir plus fort, à mieux réfléchir. Il se peut fort bien que les cas du genre soient légion, et que nous les évitions simplement par habitude, par entraînement, manquant peut-être au passage des possibilités de représentation de la connaissance qui auraient des conséquences très positives sur notre travail.

Dans tous les cas proposés en exemple, on retrouve une même réalité : il y a plus d'un niveau de représentation distinct. *Certaines classes sont des abstractions telles que les instancier devrait nous donner nos pas des individus, mais bien des* ***classes***. La classe `Class` qu'on voit dans Smalltalk, Java ou dans les langages du modèle `.NET` (où elle se nomme `Type`) est un cas bien connu d'une telle classe.

⇒ On nomme **métaclasse** une classe dont les instances sont elles-mêmes des classes.

Les définitions du concept de métaclasse sont parfois amusantes, en grande partie à cause du niveau d'abstraction impliqué. Le Wiki sur http://c2.com/cgi/wiki?MetaClass commence d'ailleurs ainsi (traduction libre) :

> Une métaclasse est un objet décrivant une classe. Par exemple, présumant une méthode d'instance `getClass()` retournant un objet décrivant la classe de cette instance :
>
> ```
> instance = new Object()
> classe = instance.getClass()
> metaclasse = classe.getClass()
> ```
>
> Selon **Tim Peters**, *[Les métaclasses] vont au-delà de ce dont 99% des programmeurs devraient se préoccuper. Si vous vous demandez pourquoi vous en auriez besoin, alors vous n'en avez probablement pas besoin – les gens qui en ont besoin savent précisément pourquoi et n'ont pas besoin d'explications* (traduction libre).

Métaclasses, réflexivité et pureté OO

Le site Wiki susmentionné ajoute que certains vont jusqu'à affirmer qu'un langage sans métaclasses est brisé, défectueux (au sens OO)… qu'il n'est pas vraiment OO.

On remarquera un argumentaire semblable à celui sur la réflexivité (voir ***La réflexivité***, plus haut) dans les langages OO, et qui s'apparente à un désir de pureté. Avec la réflexivité, on exige d'un langage qu'il permette de d'exprimer ses propres concepts à travers ses propres objets, au moins à des fins introspectives. Les métaclasses sont une partie importante d'une implémentation de la réflexivité puisqu'elles permettent d'en exprimer l'abstraction fondamentale[78].

En fait, ce qui semble très probable est qu'une implémentation dynamique de la réflexivité ne soit possible dans un langage OO *que* si le concept de métaclasse y existe.

Structure de l'abstraction stratifiée

Prenons une optique relativement puriste où toute entité instanciée est un objet, et où tout objet est une instance d'une classe.

Si tout est objet dans un langage donné, alors même l'idée de classe est un objet. C'est le principe de base qui sous-tend la réflexivité, discutée plus haut.

Il en découle qu'une classe doit être instance de quelque chose. C'est ce quelque chose qu'on nomme sa métaclasse. Logiquement, toute classe est alors instance de sa métaclasse.

On dira d'un objet qu'il est une classe si et seulement si il peut être instancé. Posé en ces termes, une métaclasse est aussi une classe, et aura aussi une métaclasse.

> Dans le langage Smalltalk, tout est objet.
>
> En C++, les types primitifs et les objets ont droit à peu près au même support du langage, mais les types primitifs sont à peu près exclus du discours sur les métaclasses (quoique, dans le monde merveilleux de la métaprogrammation, cela soit discutable).
>
> En Java, les types primitifs sont des citoyens de second ordre du langage. Dans les langages .NET, les types primitifs sont des cas particuliers de `struct`, qui sont des cousins des `class`.
>
> Dans tous les cas, on ne s'intéressera ici qu'aux objets, le discours portant sur les métaclasses.

[78] Il aurait été raisonnable d'inscrire le discours sur les métaclasses dans la section sur la réflexion, mais le concept est d'un niveau d'abstraction particulier et mérite, à mon avis, une étude plus spécifique.

Les plus astucieuses et les plus astucieux se posent sans doute déjà les questions suivantes : quelle est la métaclasse de la classe `Class` et Java ou de la classe `Type` des langages `.NET`? Quelle est la métaclasse de ces métaclasses? N'y a-t-il pas risque de régression infinie dans l'abstraction?

Oui, ce risque est bien réel, et tout modèle `OO` qui se propose de jouer à un niveau d'abstraction tel qu'il supportera pleinement les concepts de réflexivité et de métaclasse doit aborder ce problème avec sérieux et rigueur.

Dans le langage Smalltalk, ou dans Squeak qui en est le fier héritier, l'héritage simple est le seul possible et toute classe possède donc un et un seul parent. Ceci implique une racine unique à la relation d'héritage, pour éviter la régression infinie; une coupure arbitraire à partir de laquelle l'héritage s'arrête. Cette classe mère de toutes les autres est `Object`. Le même choix est fait par Java et par les langages `.NET`.

Il demeure un problème de fond, même avec ce choix : quels sont les membres d'`Object`? Qu'est-ce qui est vraiment commun à tous les objets d'un langage, outre d'être des objets? Et la réponse donnée par Java comme par les langages `.NET` est déficiente. Peut-être est-ce une fausse question…

La question qui découle naturellement de ce choix est *quelle est la métaclasse d'`Object`*? Et la réponse est habituellement qu'*il s'agit de la classe `Class`* (ou de la classe `Type`, dans l'optique des langages `.NET`). Remarquez que dans ces langages, `Class` dérive d'`Object`, puisque toute classe est un objet, mais `Class` est quand même la métaclasse d'`Object`.

Ceci met en relief le fait que les relations d'héritage, qui dénotent un accroissement de spécialisation, sont d'un autre ordre que les relations entre classe et métaclasse. Il n'est pas aussi paradoxal qu'il y paraît qu'une classe enfant soit métaclasse de son parent[79].

Poursuivant le raisonnement, toutes les métaclasses peuvent être instanciées, donc toutes les métaclasses sont, de manière directe ou indirecte, des classes dérivées de `Class` (ou de `Type`). Puisqu'une métaclasse peut être instanciée de manière à obtenir une classe, alors toute métaclasse est une instance d'une métaclasse.

Comment évite-t-on la régression à l'infini de l'abstraction par laquelle toute métaclasse est instance d'une métaclasse? Un peu comme on le fait avec l'héritage et la classe `Object`: en spécifiant un seuil au-delà duquel cette abstraction croissante s'arrête brutalement. En Smalltalk, ce seuil est la classe `Metaclass`, qui est instance d'elle-même. Ceci provoque, par définition, un seuil de traitement particulier dans le système de types.

[79] La mécanique est particulière, évidemment : si la métaclasse `Class` est nécessaire pour créer la classe `Object`, mais si `Object` est parent de `Class` (donc s'il y a un `Object` dans chaque `Class`), comment créera-t-on le premier `Object`? La clé est que la spécialisation par héritage est, au moins pour ces classes fondamentales, définie de manière statique, à la compilation des classes et de manière analogue à un plan, *ne faisant intervenir aucun objet*, alors que l'instanciation dynamique d'une classe à travers sa métaclasse est une mécanique dynamique, donc faite à l'exécution et impliquant des objets. Un peu comme la fabrication d'humains par zygotes interposés, la stratégie ici est de contourner un problème d'œuf et de poule en travaillant de manière simultanée à deux niveaux conceptuels et mécaniques distincts.

Membres des métaclasses

Certaines caractéristiques semblent être communes aux diverses implémentations existantes du concept de métaclasse. En général, les membres d'instance d'une métaclasse seront des membres de classe de ses instances, les instances d'une métaclasse étant elles-mêmes des classes.

Ainsi, dans la classe `Espèce`, si on déclare un attribut *d'instance* entier initialisé à `0` et nommé `CptIndividus`, servant à compter les individus d'une espèce donnée, alors chaque instance d'`Espèce` (qui seront des classes comme `Aigle`, `Lion`, `Cloporte` ou `Humain`) aura un attribut de classe nommé `CptIndividus` et initialisé à `0`.

On voit poindre un bris de symétrie : on ne peut en général pas faire les mêmes choses avec des instances de métaclasses (des classes) qu'avec des instances de classes (des individus). Mais pourra-t-on faire une forme de polymorphisme sur des métaclasses?

Selon les implémentations, la réponse pourra être oui : en Java, la classe `Class` est une classe dérivant de `Object`, et à travers laquelle sont possibles des opérations polymorphiques. Les classes à proprement dit, instances de `Class` selon les termes des métaclasses, ne permettent pas le polymorphisme, mais le polymorphisme est permis sur leurs instances…

Étrange, mais raisonnable, puisque la classe `Class` est une représentation par réflexivité du concept de classe, et non pas le concept de classe lui-même. L'instanciation d'une métaclasse pour obtenir une classe n'est pas quelque chose qu'on fait réellement en Java, du moins pas selon les mêmes mécanismes que ceux appliqués pour instancier une classe en tant que tel.

Les mécanismes d'instanciation des métaclasses sont souvent des mécanismes indirects. En C++, ce qui se rapproche le plus d'une métaclasse est un *template*.

Métaclasses et standardisation

Les métaclasses sont aujourd'hui surtout utilisées dans la définition formelle, lorsque cette définition suit une terminologie `OO`, de standards et de normes. Les normes `ISO` qui déterminent des modèles de données ou des protocoles utilisent abondamment les métaclasses pour définir :

- les catégories types de données qui feront partie du modèle;
- les schémas applicatifs desquels ont concevra (desquels on *dérivera*) des applications pour le modèle;
- les structures langagières qui encadreront les diverses implémentations du modèle; *etc*.

Les métaclasses se mêlent bien à la notation `UML` et constituent un outil formel par lequel se définissent les formalismes. En ce sens, elles offrent un langage commun, une structure commune aux gens qui auront à penser comment implémenter un modèle ou un standard.

Métaclasses et programmation générative en C++

En 2017, ***Herb Sutter*** a proposé une implémentation concrète des métaclasses pour C++. Celles-ci constitueraient un outil génératif, décrivant par une entité nommée des règles provoquant la génération de code sur la base de ce nom[80].

Sa démonstration (remarquée!) a entre autres généré l'équivalent technique de propriétés ou d'interfaces comme on les rencontre en C#, mais avec une page de code plutôt que plus d'une dizaine de pages de spécifications formelles.

La proposition est ambitieuse et enthousiasmante, mais amène avec elle le risque de provoquer la génération de dialectes (chose vue d'un mauvais œil en C++).

Tel que proposé, le mécanisme ne permettrait pas de changer la syntaxe du langage, mais permettrait d'expliquer comment générer certains passages de code (des classes, des fonctions, des types) et comment assurer le respect, par le code résultant, des règles décrites par les métaclasses. Un objectif à moyen terme de la démarche serait de réduire le poids sur le standard lui-même, en permettant d'enrichir le langage par des moyens moins intrusifs.

[80] https://herbsutter.com/2017/07/26/metaclasses-thoughts-on-generative-c/

Contraintes et concepts

Les concepts, sous forme de spécification technique, ont été adoptés en 2015 mais n'ont pas été adoptés dans le standard de C++ 17, une majorité souhaitant expérimenter avec ce puissant mécanisme avant d'en officialiser l'adoption. En retour, les probabilités que les concepts fassent partie de C++ 20 sont très élevées.

Chaque gain d'abstraction augmente la capacité d'expression des individus et mène à repenser les techniques et les philosophies qui nous accompagnent au quotidien, dans notre travail comme dans notre vie.

Souvent, en programmation, gagner en abstraction entraîne le développement de nouvelles techniques et résulte, contrairement à ce que certains croiraient, en des programmes plus performants, en taille comme en rapidité. La mauvaise presse que subissent parfois les idées novatrices (incluant la POO) tient souvent à un examen de ces idées à la lueur de leur premiers pas : les applications initiales ne sont pas toujours aussi élégantes qu'elles le pourraient, et les compilateurs peinent à traduire ces idées de manière à en tirer véritablement profit.

Une fois que les compilateurs et les optimisateurs atteignent un niveau de maturité raisonnable face à une percée conceptuelle donnée, une fois aussi que la pensée des programmeurs a intégré correctement les nouvelles idées, le monde des possibles s'en trouve grandi.

Le potentiel de généricité d'un langage OO peut s'exprimer entre autres :

- par le ***polymorphisme***, d'abord, qui permet de spécialiser de manière opératoire des abstractions de haut niveau (parfois explicitement abstraites comme dans le cas des interfaces ou des classes abstraites) à travers diverses entités plus concrètes; et
- par les modèles de la ***programmation générique***, où des types et des algorithmes sont conçus pour s'appliquer à un large éventail de types dans la mesure où ceux-ci offrent un certain nombre de caractéristiques. Ces caractéristiques peuvent être des attributs mais, en situation d'encapsulation, sont habituellement des opérations.

Une autre catégorie, un peu moins importante toutefois, est celle des ***délégués***, qui est une forme de polymorphisme basée sur la signature, cousine OO des pointeurs de fonctions. C++ supporte directement cette pratique par `std::function`, et C# le fait depuis longtemps par `delegate`. J'escamote ici plusieurs thématiques propres à l'abstraction (*mixin*, classes anonymes, expressions λ et foncteurs pour ne nommer que celles-là) parce qu'elles importent moins pour notre propos dans la présente section.

Le polymorphisme et la programmation générique sont tous deux disponibles à la fois en C++, en Java et en C#, quoique la programmation générique soit plus puissante sous C++ pour diverses raisons techniques et conceptuelles[81].

[81] Parmi ces raisons, on compte le plein support des opérateurs, la présence de STL dans la bibliothèque standard, mais surtout le fait que le métalangage des *templates* est un langage complet au sens de Turing.

Le polymorphisme dynamique traditionnel permet une abstraction opératoire basée sur une communauté structurelle; c'est une approche intrusive (le parent fait partie de l'enfant), mais que certaines techniques permettent de rendre extrusive. Les types auxquels s'applique le polymorphisme ont une parenté commune. Les opérations exploitant la généricité par le polymorphisme procèdent à travers un lien indirect (pointeur ou référence) sur quelque chose qui mène au moins vers ce parent commun, sollicitant des opérations que tous les enfants de ce parent ont nécessairement implémenté.

La programmation générique permet une abstraction opératoire basée sur la possibilité d'appliquer des opérations sur les types impliqués. Nul besoin d'un lien de parenté entre les types impliqués. Évidemment, rien n'empêche d'appliquer un algorithme générique sur des types polymorphiques; une application de l'abstraction n'empêche pas l'autre, après tout.

Toute technique a ses avantages et ses inconvénients.

Le polymorphisme traditionnel permet une généricité dynamique, où la méthode à invoquer sur une abstraction donnée ne sera véritablement connue qu'au moment de l'exécution d'un programme. Ce dynamisme facilite l'évolution des programmes, tout en permettant le développement d'outils extrêmement flexibles, s'appliquant à des types *a priori* inconnus. C'est ce que les anglophones nomment du *Late Binding*.

À travers une abstraction polymorphique, des opérations arbitrairement complexes peuvent être sollicitées de manière transparente. Le polymorphisme, joint au dynamisme qu'il permet, constitue la base de plusieurs schémas de conception populaires comme observateur ou *proxy* [hdPatt].

```
struct B {
   virtual int f(int) const = 0;
   virtual ~B() = default;
};
struct D0 : B {
   int f(int n) const {
      return -n;
   }
};
struct D1 : B {
   int f(int n) const {
      return n * n;
   }
};
// retournera -3? 9? Autre chose?
int g(B *pB) {
   return pB->f(3);
}
```

Le dynamisme du polymorphisme a un coût, par contre.

Tout appel de méthode polymorphique est indirect et appelle une fonction qu'il est impossible de déterminer *a priori*, outre dans quelques cas très particulier, ce qui empêche les compilateurs de procéder à plusieurs optimisations qui seraient possibles sur des opérations non polymorphiques. Invoquer une méthode polymorphique coûte donc plus cher en temps d'exécution qu'appeler une méthode qui ne l'est pas.

Implémenter le polymorphisme sur une classe demande aussi d'y insérer un peu d'information supplémentaire pour faciliter l'aiguillage des méthodes. Cette information n'occupe pas un très grand espace, mais il est sage d'en être conscient.

Ces coûts ne sont pas prohibitifs sur la majorité des systèmes, mais peuvent l'être si les ressources sont comptées (dans un système embarqué ou en temps réel, par exemple).

La programmation générique, de son côté, permet une généricité statique, où les opérations à solliciter sont connues au moment où le programme est compilé. Ceci permet au compilateur de pratiquer un certain nombre d'optimisations significatives, par exemple du *Method Inlining*. À moins que les opérations appliquées aux types par un algorithme générique ne soient elles-mêmes polymorphiques, un bon compilateur pourra générer du code extrêmement optimisé dans le cas de programmes génériques.

```
template <class T>
   T minimum(T a, T b) {
      return a < b? a : b;
   }
#include <string>
#include <iostream>
int main() {
   // ...using...
   // sollicite < entre deux int
   if(int i0, i1; cin >> i0 >> i1)
      cout << minimum(i0, i1);
   // sollicite < entre deux std::string
   if (string s0, s1; cin >> s0 >> s1)
      cout << minimum(s0, s1);
}
```

Évidemment, il y a aussi un coût associé à la programmation générique. En particulier, chaque algorithme appliqué à un type donné (*chaque instanciation d'un modèle générique*) doit être généré pour ce type, ce qui peut entraîner une croissance importante de la taille d'un programme une fois celui-ci compilé (les sources, elles, seront plus compactes).

Il y a ici encore des trucs pour s'en sortir, et les compilateurs sont devenus excellents pour optimiser au maximum le code générique.

Grandeurs et misères de la programmation générique

L'une des difficultés qu'entraîne la programmation générique a trait à la capacité qu'on les programmeurs d'y identifier les bogues.

En effet, le compilateur ne reconnaîtra un problème dans l'application d'un algorithme générique à un type donné que lorsqu'il tentera de générer le code correspondant et constatera l'absence d'une caractéristique clé.

Dans le cas à droite, la classe `Incomparable` n'expose pas d'opérateur `<` capable de comparer deux instances de la classe `Incomparable`. Si nous essayons d'appliquer l'opérateur `<` à deux instances d'`Incomparable`, alors le compilateur détectera l'erreur, évidemment. Le problème, par contre, sera celui de la génération d'un message d'erreur significatif.

Ici, on s'en tirera pas trop mal : le compilateur signalera que l'opérateur `<` est manquant pour le type `Incomparable` dans la fonction `minimum()`.

Règle générale, toutefois, surtout quand les algorithmes génériques s'appellent les uns les autres, un compilateur risque de devoir signaler une erreur sur une ligne obscure dans une fonction dont le programmeur n'a à peu près jamais entendu parler.

```
#include <ostream>
#include <string>
#include <iostream>
// ...using...
template <class T>
   T minimum(T a, T b) {
      return a < b? a : b;
   }
class Incomparable { };
ostream &operator<<
   (ostream& os, Incomparable)
      { return os << 3; } // disons
int main() {
   if (int i0, i1; cin >> i0 >> i1)
      cout << minimum(i0, i1);
   if (string s0, s1; cin >> s0 >> s1)
      cout << minimum(s0, s1);
   Incomparable ic0, ic1;
   // erreur, pas d'opérateur <
   cout << minimum(ic0, ic1);
}
```

Diagnostics et généricité

Le message d'erreur peut tenir sur des pages et des pages, retraçant chacune des étapes menant au problème et signalant les causes possibles. C'est pour le moins indigeste, et des programmes (fautifs) dont la compilation génère plus de lignes de messages d'erreurs qu'il n'y a de lignes de code source dans le programme.

Pour être utile, il faudrait au contraire qu'un message d'erreur nous indique que l'algorithme générique sollicité ne peut être appliqué au type souhaité faute d'exposer un certain nombre d'opérations, et devrait même indiquer lesquelles : ici, quelque chose comme *`Incomparable n'est pas Ordonnancable`*, par exemple. En d'autres mots, on voudrait un moyen de déterminer le contrat que devra respecter un type pour qu'on puisse lui appliquer un algorithme générique donné.

Ces contrats sont déjà exprimés dans la documentation du langage et des bibliothèques. Par exemple, on dira des types susceptibles d'être insérés dans un conteneur standard qu'ils doivent être *Assignable*, *CopyConstructible* et *LessThanComparable*. En d'autres termes, un tel type doit offrir un opérateur d'affectation, un constructeur par copie et un opérateur `<` booléen et `const` permettant de déterminer un ordre entre deux instances.

Ce sont là des noms donnés à des ensembles de règles implicites, qu'on demande au programmeur de respecter. Tout type se conformant à ces règles sera insérable dans un conteneur standard; si l'une de ces règles n'est pas respectée, alors la compilation générera des erreurs. Le souhait de tous et chacun est de faire en sorte que le contrat déterminé par ces règles soit plus explicite et qu'un compilateur puisse le valider plus clairement.

La stratégie manuelle

Il est possible de mettre en place un contrat plus formel par une saine discipline de programmation. Par exemple, si les contraintes d'un type `T` donné sont d'être *Assignable*, *CopyConstructible* et *LessThanComparable*, dans l'optique où l'on veut pouvoir insérer `T` dans un conteneur standard, on pourrait définir une classe `ContainerInsertible` comme suit :

```
template <class T>
   class ContainerInsertible {
      static void contraintes(T a) { // constructeur par copie supporté
         T b = a;
         a = b; // opérateur d'affectation supporté
         if (a < b); // comparaison avec < supportée
      }
   public:
      ContainerInsertible() {
         //
         // Obtention de l'adresse d'une méthode de classe dans laquelle sont
         // vérifiées les contraintes. Ceci n'appelle pas la fonction mais
         // force sa génération.
         // Si une partie du contrat n'est pas respectée, alors une erreur de
         // compilation sera générée à la ligne appropriée de cette méthode.
         //
         void (*p)(T) = contraintes;
      }
   };
// Présumant que ChicConteneur soit un conteneur respectant les règles des
// conteneurs standard, on voudra que le type T de ses éléments soit conforme
// à la règle définie par ContainerInsertible.
template <class T>
   class ChicConteneur {
      // Ce membre privé a pour rôle de forcer l'instanciation des contraintes
      // et, au besoin, de stopper la compilation. Si on rédigeait une fonction,
      // on aurait simplement déclaré un ContainerInsertible<T>() sans nom
      ContainerInsertible<T> nePasToucher_;
      // reste du conteneur
   };
```

Cette stratégie fonctionne somme toute assez bien, permet d'obtenir de meilleurs messages d'erreur, et formalise par une technique le respect du contrat d'un modèle générique.

À titre d'exercice, examinez le bon comportement d'un type muni d'opérateurs tels que décrit par **Alex Stepanov** sur [StepDobb] et imaginez une classe assurant le respect de ces contraintes.

Stratégie manuelle (raffinement)

Herb Sutter, dans [MExcCpp], met en relief une vision encore plus charmante de cette stratégie. Cette version a été proposée par **Bjarne Stroustrup** avec contributions d'**Alex Stepanov** et de **Jeremy Siek**.

La nuance de cette proposition est de reconnaître que dans un cas comme ceux de `ChicConteneur<T>` et de `ContainerInsertible<T>`, on pourrait affirmer que `ChicConteneur<T>` « contient » intrinsèquement, et pas seulement par composition, la contrainte `ContainerInsertible<T>`.

On donc fait face à un beau cas d'héritage privé (mais pas d'héritage public: il n'y a pas lieu de regrouper ensembles les `ContainerInsertible<T>` ou encore de les manipuler de manière polymorphique).

Profitant de cette intuition, le code devient :

```
template <class T>
   class ContainerInsertible {
      static void contraintes(T a) { // constructeur par copie supporté
         T b = a;
         a = b; // opérateur d'affectation supporté
         if (a < b); // comparaison avec < supportée
      }
   public:
      ContainerInsertible() {
         void (*p)(T) = contraintes;
      }
   };
template <class T>
   class ChicConteneur : ContainerInsertible<T> {
      // reste du conteneur
   };
```

C'est beaucoup plus élégant comme cela, n'est-ce pas? La contrainte sur `T` est intégrée structurellement à même `ChicConteneur<T>` sans que cela ne nécessite l'ajout d'un attribut nommé de manière inélégante, la mécanique devient implicite et l'effet est précisément le même.

Concept de… concept

Un mécanisme pour déterminer des contrats sur des types génériques à même le langage a été à l'étude a été adopté en 2015 sous la forme d'une spécification technique en vue d'une intégration dans C++ 17. Ce mécanisme est ce qu'on nomme les **concepts**.

Dans les mots d'**Alexander Stepanov**, tirés d'une entrevue de 2015[82] :

> « *Concepts are requirements on types; preconditions are requirements on values. A concept might indicate that a type of a value is some kind of integer. A precondition might state that a value is a prime number* »

Ainsi, on souhaite pouvoir remplacer le code ci-dessous…

```
template <class It, class T>
   It trouver(It debut, It fin, T elem) {
      for (; debut != fin; ++debut)
         if (*debut == elem) return debut;
      return fin;
   }
```

…par quelque chose qui ressemblerait au code ci-dessous

```
template <ForwardIterator It, Comparable T>
   It trouver(It debut, It fin, T elem) {
      for (; debut != fin; ++debut)
         if (*debut == elem) return debut;
      return fin;
   }
```

Remarquez les qualifications remplaçant le mot générique `class` dans la spécification des types auxquels s'applique l'algorithme générique. Ce ne sont pas des types mais des concepts, donc des règles pouvant être appliquées à des types qui ne sont pas nécessairement apparentés.

On vise donc introduire le mot clé `concept` dans le langage C++ à partir de C++ 20. Ce mot aura un usage semblable à celui du mot `class`, et sera paramétrique au sens des *templates*, mais permettra d'indiquer les opérations que doit supporter tout type respectant le contrat défini par le concept.

En identifiant clairement les attendus des divers types, l'un des principaux objectifs de la démarche sous-jacente aux concepts était de mener à une meilleure génération de messages d'erreurs (très complexes à bien générer avec les *templates*, de par leur nature).

[82] http://www.informit.com/articles/article.aspx?p=2314360

Les concepts des spécifications techniques de C++ 17

Les concepts sont passés de spécification technique en vue de C++ 17, pour fins d'expérimentation, à une inclusion (partielle, au moment d'écrire ceci) dans le texte du standard en préparation pour C++ 20. Ce qui suit demeure donc instable, mais donne un aperçu de ce qu'il est raisonnable d'attendre pour le langage sous peu.

Après plusieurs bouleversements et tergiversations, nous avons maintenant une idée assez précise de ce que seront formellement les concepts dans C++ lorsqu'ils seront officiellement adoptés.

Utiliser un concept

Pour illustrer le propos, examinons tout d'abord comment utiliser un concept un peu bidon que nous nommerons `PrefixIncrementable` et qui s'avèrera pour tout type `T` tel que si `e` est de type `T`, alors `++e` sera légal – notez immédiatement que c'est un concept d'une utilité plus que discutable, mais je l'utiliserai à titre illustratif.

Nous montrerons d'abord comment utiliser ce concept, puis nous verrons comment le définir.

Imaginons maintenant une fonction générique `prochain(T)` telle que celle proposée à droite. Cette fonction est simple, pour ne pas dire simpliste, mais sollicite le constructeur de copie de `T`, son destructeur, de même que son opérateur `++` préfixé (je n'ai pas utilisé ici le fait que `++p` devrait normalement être de type `T&`, mais on aurait bien sûr pu le faire).

```
template <class T>
   T prochain(T p) {
      return ++p;
   }
```

Si l'on suppose les concepts susmentionnés, alors on pourrait écrire la même fonction comme proposé à droite. Ici, la clause `requires` pourrait composer plusieurs concepts sous une forme d'expressions logiques, le tout étant évalué à la compilation, sur la base des types. Cette forme est verbeuse mais permet d'exprimer des idées complexes sur la base de concepts composites.

```
template <class T>
   requires PrefixIncrementable<T>
T prochain(T p) {
   return ++p;
}
```

Une autre écriture, plus concise mais tout à fait équivalente dans ce cas-ci, serait celle proposée à droite. Elle ne se prête pas à la composition de concepts mais est plus directe.

```
template <PrefixIncrementable T>
   T prochain(T p) {
      return ++p;
   }
```

La version à droite est encore plus directe, et permet de constater qu'il est possible de surcharger des fonctions sur la base de concepts.

```
PrefixIncrementable
   prochain(PrefixIncrementable p) {
   return ++p;
}
```

Notez que cette syntaxe est controversée et, au moment d'écrire ces lignes, ne semble pas destinée à être standardisée dès le début.

Notez le type de retour de cette dernière version, qui est lui-même un concept. On pourrait appeler cette fonction dans une autre fonction telle que :

```
template <class T>
   void g(T p) {
      auto q0 = prochain(p); // Ok; auto est le concept le plus général
      PrefixIncrementable q = prochain(p); // Ok; constrained auto
   }
```

Les concepts permettent de contraindre ce qu'il est possible de faire sur le plan opératoire à partir des types retournés par les fonctions, mais de manière non-intrusive.

Définir un concept

Un concept est une constante booléenne générique; une version sous forme de prédicat a été discutée pour fins de standardisation mais a été écartée en cours de route, semblant à la majorité quelque peu redondante.

Des concepts simples et souvent utilisés pour fins d'illustration sont :

```
concept C0 = true;  // tautologie
concept C1 = false; // contradiction
```

Pour `PrefixIncrementable`, nous souhaiterions sans doute un prédicat testant le support, par le type visé, de l'opérateur `++` préfixé. L'écriture serait alors :

```
template <class T>
   concept PrefixIncrementable = requires(T e) {
      ++e;
   };
```

La clause `requires` pourrait exprimer plusieurs contraintes opératoires, qui ne sont pas évaluées ici (il n'y a pas vraiment d'objet `e`, et le concept n'est pas une fonction). Si nous avions voulu exprimer que `++e` doive retourner un `T&` ou quelque chose de convertible en `T&`, nous aurions aussi pu écrire :

```
template <class T>
   concept PrefixIncrementable = requires(T e) {
      { ++e } -> T&;
   };
```

Pour un concept plus simple, comme `PasTropGros`, qui s'avèrerait quand un `T` occupe au plus deux *bytes* en mémoire, nous choix seraient les suivants :

Passer par une clause `requires`.

```
template <class T>
   concept PasTropGros = requires() {
      sizeof(T) <= 2;
   };
```

Passer par la notation de format « constante ».

```
template <class T>
   concept PasTropGros = sizeof(T) <= 2;
```

Ces notations sont complémentaires, et le choix en est surtout un de confort. Un concept peut s'exprimer sur la base de plus d'un type, de constantes entières, de variadiques… tout comme l'ensemble des *templates*. Par exemple :

```
template <class ... Ts>
   concept SontPasTropGros = PasTropGros<Ts>...;
```

Une notation concise pour tenir compte des concepts est ce qu'on appelle l'**introduction de paramètres**, proposée par souci de concision. Avec cette notation, les deux écritures ci-dessous sont équivalentes :

```
template <class T, class U>
    concept bool ZeConcept = true;
template <class A, class B>
   requires ZeConcept<A,B>
   struct ZeStruct
   {
      // ...
   };
```

```
template <class T, class U>
    concept bool ZeConcept = true;
ZeConcept{A,B} struct ZeStruct
{
   // ...
};
```

…et il en irait de même pour les suivantes :

```
template <class T, int N, class ...Ts>
    concept bool ZeConcept = true;
template <class A, class ... B>
   requires ZeConcept<A,3,B...>
   struct ZeStruct {
      // ...
   };
```

```
template <class T, int N, class ...Ts>
    concept bool ZeConcept = true;
ZeConcept{A,3,...B} struct ZeStruct {
   // ...
};
```

Cette notation inhabituelle est toutefois controversée, et pourrait ne pas faire partie de la première mouture des concepts.

Sérialisation et persistance

Cet article est général et ne fait qu'effleurer la surface d'un sujet très complexe. Des documents ultérieurs offrent beaucoup d'espace à des manières d'implémenter sainement ce qui suit.

Un sujet important dont nous devons discuter un peu en ces pages est celui de la ***persistance des objets***, associé de près à celui de la ***sérialisation***.

⇒ La **persistance d'un objet** est la capacité d'entreposer cet objet sur un média de masse, pour conservation à long terme et dans le but de pouvoir le récupérer ultérieurement de ce média pour qu'il résume sa vie utile.

⇒ Conséquemment, un **objet persistant** est un objet dont la durée de vie potentielle peut excéder celle du programme qui l'a créé.

Un objet persistant peut être entreposé dans une base de données OO comme sur un flux de données ou dans un simple fichier. On peut même envisager des zones de mémoire persistantes, dont le contenu persisterait spontanément sur disque lors d'une panne de courant.

Pourquoi vouloir des objets persistants

Lorsque le développement informatique à grande échelle prend une approche *vraiment* OO, les objets en viennent à se multiplier en mémoire, et on finit par définir le système lui-même de par la toile faite des objets qui le composent et des interactions entre ces objets.

Le système et ses objets deviennent une seule et même chose, mais examinée à partir de deux points de vue distincts :

- l'interface personne/ machine devient un objet;
- la base de données devient un objet;
- les entités de calcul et les entités organisationnelles deviennent toutes des objets;
- le système, considéré dans son ensemble, devient un objet;
- l'usager devient, au sens du système, un objet;
- chaque entité servant à des fins de représentation devient un objet; *etc*.

Tous ces objets sont, par définition du système dans lequel ils interagissent, en relation et en interaction, directe ou indirecte, les uns avec les autres.

L'usager comme objet?

Considérer l'usager comme un objet parmi d'autres est un aboutissement normal de l'approche objet, où les objets représentant telle ou telle chose se multiplient.

L'usager est connu du système de par les périphériques d'entrée auxquels il est lié, au sens du système. Ainsi, l'usager devient abstrait et prend la forme d'un objet parmi d'autres.

Loin d'être une déshumanisation de l'individu, c'est là une abstraction des plus utiles, une fusion de la pensée et de l'apparaître, par laquelle on peut plus aisément se représenter l'usager (au sens de ses besoins, préférences, aptitudes) et mieux subvenir à ses besoins.

Le domaine de l'intelligence artificielle peut tirer profit de cette optique et noter les états de l'*objet usager* comme ceux des autres objets du système, pour apprendre de l'usager, s'en construire un modèle cognitif, apprendre à l'assister dans ses tâches, prévoir ses actions, *etc*.

On peut, en tout temps, vouloir entreposer cette toile implicite sur un média, en tout ou en partie, conservant ainsi une image du système entier ou d'une portion du système à un moment précis de son évolution.

On peut vouloir agir ainsi :

- en vue de faire une copie de sauvegarde du système;
- pour des fins expérimentales, par exemple dans un contexte de simulation, pour avoir sous forme persistante un état initial à partir duquel on pourrait expérimenter plusieurs scénarios possibles, en variant certains éléments de la simulation, ou en modifiant légèrement les conditions initiales;
- pour faciliter des opérations de type annuler/ refaire (*Undo/ Redo*);
- pour survivre plus aisément aux pannes;
- pour faciliter la diffusion d'un logiciel. En effet, si les objets qui composent le logiciel constituent le logiciel en soi, cela simplifie son déploiement, qu'on le livre en un seul bloc ou en pièces détachées (pour faire payer la clientèle que pour les objets utilisés);
- et ainsi de suite (il existe des milliers de raisons pour vouloir des objets persistants).

Il est à noter qu'on est en mesure, depuis longtemps (depuis les tous débuts?), d'entreposer les *attributs* d'objets sur des médias de masse. Le problème à adresser dans le dossier de la persistance des objets est celui d'entreposer (et de récupérer) les objets *en tant qu'objets*.

On le verra, il y a un problème intéressant sur le plan conceptuel à l'application d'une approche objet au problème de la persistance. Ce problème n'est pas tant lié à l'entreposage des données qu'à leur éventuelle récupération.

Ce qu'est la persistance

La **persistance** est la capacité d'entreposer un objet puis de le retrouver plus tard dans le même état. C'est un problème complexe qui demande entre autres :

- qu'on soit capable d'**identifier** non seulement chaque classe mais aussi **chaque instance** non transitoire d'une classe donnée (pour être en mesure de distinguer *égalité de valeur* et *identité*);
- que le code d'entreposage d'un objet sur un média persistant et de reconstruction d'un objet à partir d'un média persistant soit en mesure de survivre au passage du temps et aux changements de versions; et même, idéalement
- que les objets entreposés sur un média persistant à l'aide d'une technologie donnée puissent être récupérés à l'aide d'une autre technologie.

La persistance est un objectif plus large que la sérialisation (ci-dessous). Un programme peut utiliser des objets alloués dynamiquement dans une zone de mémoire persistante de manière telle que leur état puisse en tout temps être récupéré, même suite à une panne de courant.

Ce qu'est la sérialisation

La mécanique de la **sérialisation** est une mécanique servant de projeter sur un flux une représentation d'un objet (la *désérialisation*, quant à elle, est l'opération de reconstruction d'un objet à partir des données prises sur un flux; une sérialisation utile est, évidemment, réversible). Certaines plateformes (p. ex. : `.NET` et *OpenDocument* de *OpenOffice* avec des formes sérialisées respectant les règles de `XML`) tendent vers des formes sérialisées plus ouvertes, mais la sérialisation demeure habituellement un problème local.

La sérialisation peut-être subjective : un objet existant peut se projeter lui-même sur un flux. La *désérialisation* est une opération objective du fait qu'elle ne peut être faite par l'objet qui sera reconstitué, ceci parce que l'objet à reconstituer, par définition, n'existe pas encore.

Habituellement, les moteurs facilitant la sérialisation et la désérialisation utilisent des relations collaboratives entre les types et des schémas de conception connus. Un cas type est celui du schéma de conception **fabrique** [hdPatt] :

- il est possible d'avoir une classe qui sait comment reconstruire un objet d'un flux. Il lui suffit de consommer un en-tête descriptif de l'objet (incluant habituellement un identifiant unique au type d'objet) puis de reconstruire cet objet et de le retourner sous une forme générale, que le code client devra ensuite convertir de force en un type qu'il sera en mesure de manipuler;
- il est aussi possible (et plus souple) d'avoir une fabrique générale, à laquelle s'accolent plusieurs ***microfabriques*** [hdMicroFab]. Chaque microfabrique annonce qu'elle sait décoder un certain type d'objet (chaque type étant identifié de manière unique) puis la fabrique générale, plutôt que d'implémenter elle-même toutes les stratégies de reconstruction, déléguera à la microfabrique appropriée (selon l'identifiant du type sérialisé) la tâche de désérialiser chaque objet.

Il est donc possible de mettre au point une stratégie générale de sérialisation et de désérialisation d'une séquence d'objets ***si chaque type d'objet est identifié de manière unique*** (peu importe comment du moment que les identifiants sont accessibles et comparables). Règle générale, la désérialisation sera séquentielle, respectant l'ordre de la sérialisation originale.

La sérialisation est un modèle d'entreposage primitif. Les données n'y sont pas structurées autrement que par le simple concept de séquence; impossible à ce stade d'accéder, par exemple, à tous les objets dérivant d'un même parent pour en invoquer une méthode polymorphique à moins de consommer la totalité du flux sur lequel la sérialisation a été opérée puis de filtrer les objets ainsi consommés (opération coûteuse s'il en est une).

Nous couvrirons la sérialisation de manière détaillée un peu plus loin.

SGBD et modèle OO

Le modèle OO comprend, à la base, plusieurs mécaniques. En particulier :

- l'**encapsulation**, par laquelle l'objet est responsable de son intégrité;
- l'**héritage**, qui décrit des relations structurelles entre les classes;
- le **polymorphisme**, qui permet une spécialisation des comportements;
- l'**abstraction**, à l'aide d'interfaces et de classes abstraites, qui permet de cacher des familles entières d'objets derrière une devanture opérationnelle;
- l'**instanciation**, qui décrit les règles de construction d'un cas particulier d'une classe donnée;
- et bien d'autres encore.

Il n'y a pas de consensus universel quant à ce que signifie vraiment être OO. J'ai ma vision des choses (voir la section ***Être OO***, plus haut), d'autres défendent des visions qui, sans être disjointes de la mienne, ne s'y superposent pas nécessairement. Loin de moi (et, je l'espère, de celles et ceux qui ont une vision différente de ma mienne) la prétention d'avoir la vérité absolue en ce sens.

Nous sommes toutes et tous conscient(e)s que le modèle OO est, nettement, le modèle de programmation et de conception dominant sur le plan commercial aujourd'hui. Toute approche à la persistance doit, éventuellement, permettre d'y intégrer naturellement des objets.

Être ou ne pas être objet

La vision fondamentale de ce qu'est ou non un objet est, en soi, une question à laquelle il existe plusieurs réponses possibles. Peu importe l'optique choisie, certains concepts comme les attributs de classe et la représentation des opérations posent problème.

En effet, si une BDOO ne sert qu'à un seul langage, alors il importe peu que les attributs de classe (ceux qualifiés `static` en C++, Java et C# ou qualifiés `Shared` en VB.NET) y soient entreposés ou non.

Par contre, si des objets doivent être stockés à partir d'un langage et récupérés puis reconstruits dans un autre langage, alors la cohérence des attributs de classe devient ardue à garantir. En retour, si un attribut de classe est constant, l'entreposer pour le récupérer est-il raisonnable?

De même, un objet est, fondamentalement, un regroupement sous un même nom de données et d'opérations sur ces données : le concept de type, intégré de manière fortement cohésive. N'entreposer que les données dans une BDOO est-il, dans ce cas, une approche gagnante?

Catégories d'héritage

L'un des thématiques variant d'un langage OO à un autre est le support accordé à l'héritage au sens large (d'interface comme d'implémentation).

Certains langages (Java, les langages .NET, Smalltalk) privilégient l'héritage simple (du moins l'héritage simple d'implémentation) de manière exclusive alors que d'autres (dont C++ et Eiffel) supportent l'héritage multiple sous toutes ses formes.

Il est possible de simuler l'héritage multiple d'implémentation par une combinaison d'héritage d'interfaces, de composition et de délégation (voir la section ***Concepts ou techniques***, plus loin). Cela dit, pour entreposer un objet C++ dans une BDOO puis le récupérer en Java par la suite, il faut qu'un mécanisme prenne en charge la mise en place d'un pont entre les deux approches.

Les représentations des classes, d'un langage à l'autre, doivent être prises en charge par les programmes[83]. Cependant, la correspondance entre les données dans la BDOO et les classes à instancier dans un programme donné doit avoir une forme OO pour éviter que la BDOO et le programme ne doivent transiger des données brutes et, du même coup, briser l'encapsulation.

Une approche choisie par plusieurs moteurs de BDOO est d'imposer des contraintes sur les types d'objets susceptibles d'être entreposés dans la BDOO en question (par exemple, implémenter une interface précise). Cela pose un problème de refactorisation important (et potentiellement coûteux) pour qui désirerait intégrer une BDOO à un système existant.

[83] L'alternative est de faire reposer la reconstruction des objets sur des mécanismes de réflexion, ce qui entraînera des programmes très lents et difficiles à concevoir.

Qualifications particulières

Certaines qualifications de sécurité sont tellement répandues qu'on les prend pour acquis. La qualification `public` a habituellement le même sens d'un langage à l'autre et il en va de même pour la qualification `private`.

Il y a par contre d'autres qualifications possibles dans certains modèles `OO` et il y a des glissements sémantiques d'un langage à l'autre (le mot `protected`, par exemple, a un sens légèrement différent en Java et en C++).

Certaines qualifications sont aussi locales à un modèle ou l'autre (pensons à `package private` en Java et *privé pour un assemblage* dans un langage `.NET`). Pour respecter l'encapsulation et les qualifications de sécurité sur le plan conceptuel, un `SGBDOO` visera à limiter la compromission des règles applicables au modèle original (celui d'où viennent les objets) dans le programme qui extraira les objets.

Il n'est pas clair qu'il existe une solution universelle à ce problème.

Concepts non transférables

Tous les langages `OO` ont leur spécificité et leurs caractéristiques peu (ou non) transférables à d'autres langages et à d'autres modèles `OO`. Par exemple :

- les objets constants sur une base instance de C++ ne sont pas représentables tels quels en Java ou dans un langage `.NET`;
- la méthode `finalize()` d'une classe Java n'est pas nécessairement (en fait, n'est souvent pas) invoquée dans les programmes Java alors que les destructeurs C# sont invoqués de manière non déterministe et ceux de C++ sont invoqués de manière déterministe. Comment tracer un pont entre le concept et la pratique dans ce cas?
- que faire de pratiques autodocumentaires comme les traits en C++ ou les métadonnées à même le code de Java et des langages `.NET`?

Tous ces détails posent, chacun à sa façon, le problème du lien entre une idée, une philosophie ou une pratique dans un langage `OO` donné et le concept plus général de modèle `OO` (si tel concept existe vraiment en tant qu'entité).

Approches possibles

Plusieurs approches sont possibles pour entreposer des objets afin d'obtenir une forme pratique de persistance. Nous explorerons quelques-unes des plus importantes, qui sont toutes liées de près les unes aux autres.

Approche OO : objets persistants et flux

Le standard C++ par lequel on lit quelque chose d'un flux en entrée via son opérateur >>, et où on écrit quelque chose dans un flux en sortie à l'aide de son opérateur <<, a fait ses preuves.

Nous avons déjà examiné, dans notre étude de la surcharge des opérateurs, la possibilité de surcharger la fonction `ostream &operator<<(ostream&,const T&)` (et sa contrepartie en lecture) pour tout type `T`.

Cela signifie que, présumant les opérateurs en question définis pour le type `ChicType`, la persistance sur un flux est, en C++, aussi simple que le sont les accès aux périphériques d'entrée/sortie standard avec les objets `cin` et `cout`, deux mécanismes que nous utilisons déjà depuis fort longtemps.

```
#include <sstream>
#include <fstream>
using namespace std;
// ...
stringstream sstream;
ChicType a, b;
sstream << a;
sstream >> b;
// ...
{
   ofstream ofs{"fich.dat"};
   ofs << a;
}
// ...
{
   ifstream ifs("Fich.dat");
   if (ifs)
      ifs >> b;
}
```

Notez que les objets ainsi entreposés peuvent l'être sous forme texte, qui serait lisible par un humain, mais ne le sont pas nécessairement. On s'attend à ce que les opérateurs << et >> sur les flux offrent des fonctionnalités **symétriques** pour une classe donnée, et à ce qu'ils soient responsables à la fois de l'entreposage et de la reconstitution de l'objet[84].

Pour qui estime l'approche `OO` comme reposant sur des opérations subjectives, les stratégies d'écriture d'objets sur des flux et de consommation d'objets pris dans des flux peut ne pas sembler `OO`. Après tout, elle repose sur un programme appliquant une opération à des objets existants plutôt que sur des objets actifs en tant qu'eux-mêmes.

Dans une approche par **sérialisation** à l'aide d'opérateurs sur des flux, le code client a aussi la particularité que l'objet doit exister *a priori* si une opération doit lui être appliquée. L'extraction d'un objet à partir d'un flux, en particulier, présume un objet existant (donc déjà construit) qui sera modifié par l'extraction d'un flux d'une représentation sérialisée qui lui sera appliquée. Cela implique au moins un mécanisme de construction applicable à l'objet puis une initialisation supplémentaire à partir des données lues : une initialisation en deux temps.

Il est possible de définir des stratégies d'extraction de données à partir de flux qui ne reposent que sur une construction paramétrique privée et une construction par copie publique. Voyez-vous comment?

[84] Le problème de récupération des objets persistants d'un support donné, sur lequel nous reviendrons plus loin, est déjà visible ici. Le percevez-vous?

Notez que l'objet n'est pas écrit sur un flux. C'est bel et bien une représentation de l'objet, suffisante pour le reconstruire, qui y sera projetée. Sérialiser, c'est décrire. *Désérialiser*, c'est reconstituer. Ceci implique entre autres que la sérialisation permet, dans la mesure où il y a un accord de format entre les diverses technologies (ce qui est *beaucoup* plus complexe qu'il n'y paraît!), de transiger des objets d'un langage à l'autre, d'une plateforme à l'autre et d'une version à l'autre d'une même technologie.

Les objets avant sérialisation et après *désérialisation* ne sont pas un seul et même objet, même si les technologies avant et après sont identiques. Transférer une vulgaire chaîne de caractères entre C++, Java et un langage `.NET`, par exemple, résultera en trois objets fondamentalement différents mais susceptibles de servir aux mêmes fins et d'être construits à partir de données de formats semblables.

Il devrait être clair à vos yeux qu'implémenter une projection subjective d'un objet sur un flux est possible, et même simple : il suffit d'exposer une méthode d'instance, probablement publique, prenant en paramètre un flux sur lequel l'objet écrira une description de lui-même, description suffisante pour, éventuellement, reconstruire un objet qui lui sera identique.

Il devrait aussi vous apparaître qu'implémenter une extraction subjective d'un objet à partir d'un flux est problématique : comment un objet peut-être s'extraire lui-même d'un flux sans exister au préalable? N'est-ce pas un problème d'œuf et de poule?

Dans la plupart des langages, on solutionnera ce problème en divisant les tâches en deux : une classe offrira des services (souvent une méthode de classe; en C++, un opérateur représenté sous forme de fonction globale pour étendre la gamme d'opérations applicables aux flux) capables de prendre une instance et de la représenter sur un flux ou d'extraire des données d'un flux et d'instancier un objet à partir de celles-ci, alors qu'un autre objet sera responsable de faire le lien entre le flux et la classe dont il faut y projeter ou à extraire des instances.

Ceci met en relief que, pour certaines opérations, même fondamentales, l'approche `OO` est plus riche qu'il n'y paraît et s'exprime mieux par collaboration entre plusieurs objets que par une tentative, par une classe donnée, de tout prendre en charge.

Les objets sont ici responsables d'entreposer (et de reconstruire) eux-mêmes leurs relations. La problématique peut être très complexe lorsque les objets se réfèrent les uns aux autres par des pointeurs et n'ont pas d'identité unique propre[85]: comment reconnaître, *après reconstruction*, les objets avec lesquels nous étions en relation avant l'entreposage?

Notez que le problème de reconstruire les relations entre les objets est un problème extrêmement important. Si un objet prend son sens de par ses associations avec d'autres objets, alors il faut, en entreposant un objet sous forme persistante, décrire non seulement l'objet mais aussi *tous ceux auxquels il est associé*. Cela implique intégrer des schèmes d'identification systématiques des objets, un problème en soi. Dans la plupart des langages, ce ne sont pas les objets qui s'identifient eux-mêmes mais bien l'infrastructure en soi ou encore l'objet chargé de faire le lien entre les objets à sérialiser et les flux eux-mêmes.

[85] Par exemple, lorsque chaque objet ne se voit pas apposer, de manière intrinsèque, un identifiant unique dans le système tout entier.

Approche 01 : objets persistants et bases de données relationnelles

Du côté des standards de persistance, d'entreposage et de recherche, le modèle dominant demeure le modèle relationnel. Ce modèle demande qu'on décompose un objet avant de l'entreposer sur un média persistant : un bris d'encapsulation.

Le modèle relationnel est efficace à plusieurs points de vue mais colle de près au modèle de développement et de conception procédurale des programmes. La BD y est une combinaison de structures et d'algorithmes opérant sur des données.

Les données entreposées dans une BD relationnelle ont des relations connues de l'extérieur. L'objet entreposé dans une BD perd son intégrité structurelle et son identité en tant qu'objet.

Grossièrement[86], une base de données est une structure de données riche, entreposée sur un média de masse[87], qui présente une interface de requêtes puissante et efficace et offre une forme de sécurité et de contrôle de l'intégrité transactionnelle. On y entrepose les données sous forme de *tables*, composées de *colonnes* (*champs*) et de *lignes* (*enregistrements*). Chaque champ a un type et un ensemble de propriétés. Chaque enregistrement d'une table est constitué d'une occurrence de chaque champ.

Requêtes de type SQL

Le **langage SQL** (pour *Structured Query Language*) sert de protocole standard (malgré ses très nombreuses variations locales) pour effectuer des requêtes à une base de données relationnelle.

À l'aide d'énoncés se rapprochant de l'anglais, on peut consulter ou modifier une base de données de manière flexible. Des moteurs et des pilotes[88] complètent le portrait et permettent généralement de faire, par programmation, ce qui est difficile (ou impossible) avec une interface reposant strictement sur SQL.

Le modèle de BD dominant étant relationnel et le modèle conceptuel dominant étant OO, la majorité des projets visent une jonction entre les deux : les objets se décomposent pour s'inscrire dans une BD et peuvent être reconstitués par du code tiers. Les programmes accédant aux SGBD sont responsables d'assurer le respect et l'implémentation des concepts OO tels que le polymorphisme ou l'abstraction, et la vie continue.

---

[86] Ceci est une description très simplificatrice, qui ne rend pas justice à la richesse du concept. Elle ne nous servira qu'à illustrer notre propos, qui est la persistance des objets à travers une base de données relationnelle.

[87] … ou qu'on répartit sur plusieurs médias distincts, tout en accédant comme s'il s'agissait d'une seule et même entité complexe.

[88] On pense par exemple aux moteurs du modèle UDA (*Universal Data Access*) comme ADO, et aux pilotes comme ceux des modèles JDBC et ODBC.

Le modèle objet-relationnel (OR)

Il existe, entre autres chez de gros vendeurs comme Oracle ou mises en place par des technologies ouvertes comme Hibernate, des modèles de bases de données relationnelles munies d'une strate spécifiquement vouée à faciliter une approche objet. On parle alors d'une **base de données objet-relationnelle**. Dans une telle base de données, l'entreposage se fait quand même sous forme relationnelle.

Les **SGBD objets-relationnels** (**SGBDOR**) constituent le pont entre les modèles OO et relationnel le plus fréquemment rencontré dans les applications commerciales.

L'idée est simple : laisser les deux modèles exister et chercher à faciliter le lien entre eux à l'aide d'une stratégie hybride. L'approche OR se veut d'abord pragmatique.

Prudence : ceci n'est pas une prise de position en faveur un modèle ou l'autre. Les SGBDOR sont la manière privilégiée de réaliser des ponts entre les SGBDR et le monde OO aujourd'hui; la situation changera peut-être éventuellement.

Toute approche a ses qualités et ses défauts. Comme tous les hybrides pragmatiques, les SGBDOR peuvent être critiqués sur le plan de l'élégance mais sont suffisamment appréciés par l'industrie pour être considérés comme une solution réaliste au problème de l'intégration des deux modèles dominants dont nous discutons ici.

Dans un SGBDOR, on trouve habituellement des objets *lignes* (des instances) et des objets *colonnes* (des attributs). Les objets lignes sont identifiés de manière individuelle et unique de manière à permettre de référer de manière non ambiguë à n'importe quel objet de la BDOR et de manière à faciliter la mise en place de diverses relations entre objets, principalement des relations d'association, de composition ou d'agrégation.

Les SGBDOR intègrent plus harmonieusement que les SGBDR le concept de type, cher au monde OO, dans un modèle relationnel. Cette intégration se fait en partie par un raffinement (controversé) de SQL (norme SQL:1999) et par l'identification systématique de chaque objet.

Un exemple comparatif entre SGBDR et SGBDOR, pris sur Wikipedia mais semblable à ce qu'on retrouve dans les manuels des divers vendeurs de SGBDOR sur le marché, serait celui-ci-dessous, où une table Clients représenterait des… clients.

Le *Object Data Management Group*, ou ODMG[89], a proposé à la fois une stratégie de correspondance entre langages OO et BDOO et une extension OO (nommée OQL[90]) au langage SQL. L'idée est pertinente mais le tout n'a jamais vraiment pris son envol.

Définir un standard est une chose très difficile; souvent, les standards naissent plus souvent des usages et pratiques en vigueur que du souhait d'un organisme – dans la majorité des cas, les organismes de standardisation sont plus efficaces pour codifier l'existant que pour introduire un standard tout neuf.

La création d'une table dans un SGBDR implique une description des champs par leur nom et par leur type. Ici, une entrée est créée pour un identifiant de client, un nom de famille, un prénom et une date de naissance, chacun étant défini de manière détaillée et organique.

[89] Voir http://en.wikipedia.org/wiki/ODMG

[90] Voir http://en.wikipedia.org/wiki/Object_Query_Language

La création d'une table équivalente dans un SGBDOR utiliserait des types plus précis (tout en étant plus abstraits) pour représenter une identification de client et un nom. Ceci implique une indirection dans la table Clients d'un SGBDOR – selon toute probabilité, ce qui y sera entreposé sera non pas la donnée brute mais bien un lien vers celle-ci dans une autre table.

SGBDR

```
CREATE TABLE Clients (
  Id              CHAR(12) NOT NULL PRIMARY KEY,
  Nom             VARCHAR(32) NOT NULL,
  Prenom          VARCHAR(32) NOT NULL,
  DateNaissance   DATE        NOT NULL
);
```

SGBDOR

```
CREATE TABLE Clients (
  Id              IdClient NOT NULL PRIMARY KEY,
  Nom             TypeNom  NOT NULL,
  DateNaissance DATE       NOT NULL
);
```

Pour réaliser une requête permettant de trouver les clients dont c'est aujourd'hui l'anniversaire, un SGBDR pourrait chercher les champs Id dans la table Clients pour lesquels la procédure Birthday() (œuvrant à partir du Id d'un client ce qui, il faut le souligner, est contre-intuitif pour un développeur OO) retourne une valeur correspondant à celle retournée par la procédure Today().

Dans un SGBDOR, la recherche se fera par client (chaque client était ici identifié par la variable C pour la durée de la requête). Le lient entre nom et date de naissance passe par le concept de client, ce qui est plus près d'une vision OO, et la procédure Birthday() retourne probablement un identifiant d'objet comparable avec celui de l'objet TODAY.

SGBDR

```
SELECT Formal(Id)
  FROM Clients
  WHERE Birthday(Id) = Today()
```

SGBDOR

```
SELECT Formal( C.Nom )
  FROM Clients C
  WHERE BirthDay ( C.DateNaissance ) = TODAY;
```

Un peu comme dans le cas des SGBDOO, c'est dans la réduction de l'importance des jointures qu'on apprécie le plus les SGBDOR.

La requête à un SGBDR à droite, par exemple, repose sur une jointure entre des adresses et des clients sur la base d'un identifiant de client.

SGBDR

```
SELECT InitCap(C.Nom) || ', ' || InitCap(C.Prenom), A.Ville
   FROM Clients C, Adresses A
   WHERE A.IdClient=C.Id -- la jointure
     AND A.Ville="Longueuil"
```

Dans la version reposant sur un SGBDOR, à droite, le recours à une jointure explicite dans la requête devient caduque du fait que naviguer les relations à partir d'un client suffit pour retrouver les noms des clients habitant Longueuil. C'est plus simple.

SGBDOR

```
SELECT Formal( C.Nom )
   FROM Clients C
   WHERE C.Adresse.Ville="Longueuil"
```

Notons toutefois que les SGBDOR ne mettent pas implicitement en place une structure respectant le principe d'encapsulation. Les données demeurent entreposées dans des tables qu'il est possible de consulter à l'aide de requêtes SQL conventionnelles[91] et il n'est pas (en général) possible d'interroger un SGBDOR sur la base du résultat de l'invocation d'une méthode (donc de supporter pleinement le polymorphisme)[92].

Pour résumer grossièrement, un SGBDOR représente adéquatement un objet dans une forme relationnelle mais n'est pas nécessairement conscient des particularités de cet objet en tant qu'il est un objet.

[91] Évidemment, il est possible de mettre en place des mécanismes de sécurisation de la BD qui y assureront manuellement le respect de l'encapsulation, comme il est d'ailleurs possible par programmation d'n faire autant dans la plupart des langages de programmation qui ne sont pas OO… Ce qui distingue les outils OO des autres dans ce cas est le support *intrinsèque* et *naturel* offert aux concepts OO.

[92] Les extensions SQL des BDOR sont disponibles (version presque finale – la finale est disponible, mais n'est pas gratuite) sur [BDORSQL].

La technologie LINQ

Les concepteurs de Java et de C# se font, au moment d'écrire ces lignes, compétition pour mettre au point la technologie dite de *Language Integrated Query*, ou LINQ (acronyme proposé par *Microsoft*). Par cette technologie, les requêtes SQL apparaissent comme intégrées au langage, de manière naturelle ne nécessitant (en apparence du moins) pas d'interaction avec divers pilotes d'accès aux bases de données.

C# a une longueur d'avance sur Java de ce côté puisque LINQ fasse partie, avec les expressions λ [POOv02], des innovations propres à la version 3.0 de ce langage.

La technologie LINQ ne règle pas le problème de la persistance des objets mais facilite l'accès à un SGBD relationnel à l'aide d'un langage OO. La requête SQL apparaît comme une structure objet du langage, et le résultat apparaît comme une collection convenablement remplie d'objets génériques devant être convertis explicitement dans les types appropriés aux champs effectivement obtenus. C'est un très élégant mélange de langage fonctionnel et de langage objet, en lien avec le monde des BD et, plus encore, avec le modèle des collections de l'infrastructure .NET toute entière (on peut réaliser des requêtes LINQ sur un conteneur .NET aussi simplement que sur un SGBD).

Entreposage d'objets et SGBD relationnel

Insérer un objet dans une base de données relationnelle demande qu'on en décompose la description en champs et qu'on enregistre ces champs individuellement[93]. Cet entreposage peut donc être réalisé de manière OO, au sens où il peut être fait par un objet spécialisé à l'aide d'une stratégie subjective; à la limite, une BD pourrait être représentée par un flux pour les fins de l'insertion de valeurs à partir d'un objet.

Insérer des objets dans un SGBD relationnel présume souvent l'existence d'une table par classe d'objet persistant dans la base de données, ou d'une représentation structurelle où l'héritage se mêle à des hiérarchies de tables et où chaque objet décomposé risque d'être éparpillé en au moins autant de tables qu'il a d'ancêtres.

Prenant l'exemple d'une classe Voiture, on entreposerait une instance dans un garage (représentant la base de données) :

- selon une approche structurée, en la démontant avant de l'y rentrer; ou
- selon une approche objet, en lui demandant de se démonter puis de s'y entrer elle-même.

Visiblement, la dichotomie entre insérer un objet dans une base de données, qui peut être une opération subjective de l'objet à entreposer, et extraire un objet d'une base de données, qui ne peut pas être une opération subjective de l'objet à extraire, apparaît avec les bases de données comme avec la sérialisation.

Une nuance s'impose ici : quand j'écris que l'extraction ne peut être une opération subjective du point de vue de l'objet à extraire, j'entends que l'objet, s'il existe avant extraction, aura une identité avant extraction distincte de son identité après extraction.

[93] On voit tout de suite une similitude stratégique avec l'entreposage par opérateurs sur des flux, abordée plus haut. Malgré les différences superficielles, les deux approches sont, d'un point de vue OO, équivalentes.

Représenter l'encapsulation et le polymorphisme

Certains moteurs de bases de données objets-relationnels permettent de représenter des méthodes sous forme de procédures stockées, et implantent l'encapsulation à travers des mécanismes de bases de données (particulièrement des vues et procédures stockées). Ces moteurs souffrent en retour d'une différence structurelle de fond entre le modèle relationnel, normalisé selon des règles précises et formelles, et les modèle objet, fait de relations entre les entités qui ne sont pas soumises à ces règles.

Il reste qu'au sens strict, l'encapsulation et le polymorphisme sont difficiles à représenter dans une base de données relationnelle. Les enregistrements y sont faits de données tabulaires, visibles par SQL, ce qui brise d'office l'encapsulation stricte lors de l'insertion dans la base de données.

Le polymorphisme tend, de son côté, à reposer sur le code associé aux objets, et à ne pas être représenté en soi dans la base de données, du moins pas de manière indépendante des langages. Certains moteurs de bases de données permettent d'invoquer des méthodes (des procédures stockées) des objets à partir de requêtes à la base de données, mais entreposer un objet décrit par un langage de programmation donné dans une base de données puis l'extraire à l'aide d'un autre langage de programmation tend à détruire (ou, du moins, à modifier) la capacité opératoire de l'objet. Nous sommes meilleurs pour décrire les données que pour décrire les opérations.

Pour représenter des concepts OO dynamiques comme le polymorphisme sur un support physique ou logique qui n'est pas en soi OO, il faut concevoir un formalisme de représentation de plus haut niveau et décrire les dynamiques en question dans ce formalisme. Ceci n'est pas une mince tâche.

La plupart du temps, l'entreposage d'un objet dans une base de données relationnelle se limite à l'entreposage de ses attributs. On présume que le programme qui extraira éventuellement l'objet pour s'en servir sera conscient *a priori* de tous les détails structurels et dynamiques complexes qui n'auront pas été décrits sous forme tabulaire dans la base de données. Le problème de la persistance de l'objet est alors réglé, dans une certaine mesure, mais au prix de mettre de côté ce qui donne à l'approche OO toute sa force.

Approche 02 : objets persistants et bases de données OO

Un SGBDOO se veut une alternative raisonnable aux SGBD traditionnels et une alternative avantageuse à la sérialisation brute. En effet :

- il peut être utile d'entreposer le ***modèle de données*** entier d'une application à la fin d'une session de travail pour récupérer de modèle lors du prochain démarrage de cette application;
- il peut aussi être utile de ne récupérer que certaines portions du modèle de données sans avoir à tout recharger en mémoire;
- de plus, la sérialisation permet la persistance mais ne permet pas automatiquement de respecter des contraintes d'intégrité transactionnelle (les classiques opérations *Rollback* et *Commit*), ce qui peut être fâcheux lorsqu'un programme plante.

On vise évidemment le respect des contraintes ACID bien connues (traduites littéralement : Atomicité, Conséquence[94], Isolation, Durabilité). La capacité de faire des *Rollback* (annulation d'une transaction avant qu'elle ne soit effectuée) est une garantie forte qu'il est pénible d'implémenter à la pièce avec de la sérialisation.

Les bases de données OO sont, à ce jour, beaucoup moins répandues que les bases de données relationnelles. Leur mandat officiel est d'être à la fois un SGBD et un système OO, donc un système qui, en autant que faire se peut, sera conceptuellement de niveau avec les principaux langages OO.

Les SGBDOO tendent à conserver sur un média persistant une représentation arborescente des objets et de leurs relations plutôt qu'une version tabulaire. Cette représentation permet en générale beaucoup plus rapidement de reconstruire un programme ou des structures de données utiles à la programmation, et tend à être très efficace pour permettre de retrouver de l'information en fonction de laquelle les objets entreposés auront été pensés (pour laquelle les accesseurs offrent un chemin pertinent). C'est compréhensible, la représentation sur média de masse étant déjà conforme aux besoins du programme.

Les SGBDOO tendent par contre à être beaucoup moins performants pour ce qui est de réaliser des recherches arbitraires d'information. C'est raisonnable : pour réaliser une recherche arbitraire, il faut réaliser des bris d'encapsulation. De même, une base de données non normalisée n'est pas nécessairement organisée pour faciliter des recherches quelconques.

Pour qu'un SGBDOO mérite le titre de SGBD, celui-ci devrait :

- offrir un support pour la persistance des objets;
- proposer une gestion efficace de l'espace d'entreposage;
- offrir du support en vue d'accès concurrents;
- avoir la capacité de récupérer suite à une panne ou à un bris; et
- offrir la possibilité de lui soumettre des requêtes *ad hoc*.

[94] L'anglais *Consistency* ne se traduit pas par « consistance » en français dans ce cas-ci.

Pour qu'un SGBDOO mérite le qualificatif OO, celui-ci devrait :

- des objets complexes (des composés, des agrégats, des collections);
- les relations entre objets (associations diverses, héritage, amitié);
- l'identité des objets;
- l'encapsulation;
- les idées de type et de classe;
- la surcharge de méthodes;
- le polymorphisme;
- la capacité d'étendre le système de types existant;
- offrir des facilités de calcul complètes[95]; *etc*.

Plusieurs éléments pourraient être ajoutés à la liste, sans être considérés essentiels au moment d'écrire ces lignes : la possibilité d'inférer dynamiquement le type d'un objet persistant, le support d'héritage multiple, la gestion des versions d'objets; *etc*.

Modèle de données

Les modèles de données relationnels font le travail avec des données sur lesquelles on impose des relations. Ils sont moins efficaces pour représenter les relations naturelles entre objets dans un système complexe (les toiles d'associations, relations d'héritage, d'agrégation et, pour les langages qui les supportent, de composition).

Dans un programme OO, les liens entre les objets sont naturellement représentés par la capacité qu'ont des objets de s'envoyer des messages (donc la capacité qu'a un objet d'invoquer une méthode d'un autre objet).

Que ces relations soient intrinsèques (comme dans le cas de l'héritage) ou construit par des pointeurs ou des références (parfois même les deux), le fait est que les relations existent et devaient pouvoir être capturées et exploitées à fond.

Avec certains SGBDOO, les classes pourront être annotées à l'aide d'un outil externe au compilateur pour faciliter la prise en charge de l'entreposage des objets dans la BD et la reconnaissance des relations entre les objets. C'est une bonne chose; certains moteurs OO exigent que les classes dérivent toutes d'un parent commun, ce qui complique l'existence de ceux qui utilisent un langage à héritage simple seulement comme Java et les langages .NET.

[95] Ceci est une exigence assez forte, puisqu'elle n'est pas couverte au sens strict par SQL. La plupart des gens situent ce critère au niveau des souhaits, des vœux pieux, plus qu'à celui d'exigence absolue.

Stratégies

Il existe plusieurs approches pour représenter les objets et les relations entre objets de manière neutre. Il est important de le faire, d'ailleurs, peu importe que le moteur soit en mesure de déduire ou non les relations existantes lors de l'entreposage.

Parmi les approches possibles, on compte :

- le recours à des descriptions neutres (langage `ODL` pour les objets, langage `IDL` ou `WSDL` pour les interfaces). Il faut alors distinguer la représentation opérationnelle (les abstractions, les interfaces) de l'implémentation dans un langage donné (ce qu'on nomme souvent le *Language Binding*), en espérant que le pont soit possible;
- les `SGBD` relationnels supportent typiquement les accès à travers des requêtes `SQL`, qui retournent des descriptions structurées mais qui révèlent directement les données, ce qui convient au modèle relationnel mais va à l'encontre de certaines bases de l'approche `OO`. Une manière d'interroger une `BDOO` sans contrevenir aux règles de l'encapsulation est souhaitable (la réflexivité, supportée d'office par Java et par les langages `.NET`, est un pas dans cette direction);
- dans cette veine, il faut s'interroger quant à la nature de ce qui sera retourné par une requête à un `BDOO` : obtiendra-t-on une description des données semblable à un `ResultSet` (ou à un `RecordSet`, ou à un `DataSet`, ou …)? Obtiendra-t-on l'objet à proprement dit (et, dans ce cas, devra-t-on imposer une racine commune à tous les objets entreposés dans la `BD`)? Obtiendra-t-on des simulacres (des *proxies*) opérationnels qui permettront d'utiliser, de manière opaque, un équivalent opérationnel de l'objet original?

SGBD relationnels et SGBDOO

Les SGBD relationnels, ou SGBDR (pour être bref) ont (et auront sans doute longtemps encore) une plus grande part du marché des bases de données que n'en ont les SGBDOO, et ce pour plusieurs raisons, dont :

- l'**ubiquité** : les SGBDR sont partout. La mouvance requise pour remplacer un produit ayant atteint ce niveau de déploiement et d'acceptation par les milieux technologies est considérable, et bien qu'il soit possible qu'une telle mouvance en vienne à se créer, je doute que qui que ce soit la perçoive dans le moment;
- la **fiabilité** : les SGBDR fonctionnent. Ce bête constat frappe dur d'un point de vue pragmatique chez tous les tenants de l'adage anglais *If it ain't broke, don't fix it!*[96];
- la **clarté** : les SGBDR sont codifiés. J'entends par cela qu'il s'agit d'une des technologies les plus matures du monde de l'informatique, fortifiée d'un vocabulaire et de pratiques bien définies[97]. Ceci favorise un enseignement du modèle, donc sa pérennité.

Cette liste n'est, évidemment, pas exhaustive. En retour, un gouffre (le mot est peut-être fort, mais il est consacré) se creuse entre le monde du développement informatique (fortement engrangé dans la mouvance OO) et le monde de l'entreposage et de l'organisation des données à long terme (fortement relationnel).

Les intérêts des tenants des deux approches divergent souvent sur des points de fond. Quelques points de divergence parmi d'autres :

- là où une BD relationnelle est souvent modélisée de manière à réduire la redondance des données, le monde OO manipule fréquemment diverses formes de collections. Les modèles OO dynamiques ne respectent pas souvent la 1ière forme normale et, conséquemment, la transposition d'un ensemble d'objets dans une BD relationnelle tend à faire de même;
- les SGBDR travaillent au niveau des données alors que la modélisation OO privilégie une abstraction des données en faveur d'un point de vue axé sur les opérations. Une des idées maîtresses du modèle OO est de favoriser un raffinement de l'implémentation tout en assurant une stabilité au niveau des interfaces. Visiblement, les objectifs de l'un et de l'autre, sans être incompatibles *a priori*, sont en partie disjoints;
- les objets se réclament du principe d'encapsulation. Les SGBDR étalent les données sous forme de tables;
- les objets travaillent beaucoup au niveau des types (héritage, polymorphisme, associations, abstractions). Une multiplicité de types dans un système informatique OO est chose normale (et souvent souhaitable!) alors qu'une transposition imprudente du modèle OO d'un programme à un SGBDR pourrait résulter en une multiplicité de tables ne contenant que peu ou pas de données.

---

[96] En québécois : si ça fait le travail, n'y touche pas!

[97] En fait, le monde des BD est bien codifié et celui des SGBDR semble être le mieux défini d'entre tous. Pensons seulement à SQL!

Il y a aussi des éléments qui rapprochent les SGBDR du modèle OO. On n'a qu'à penser à la question de l'identité des objets, qui peut se représenter à l'aide de clés dans une BD, et à la formalisation des relations dans un SGBDR (la grande force de ce modèle, évidemment) qui peut souvent être directement appliquée à une multitude de relations entre objets (en particulier la relation d'association, qui repose sur un couplage moins fort que des relations comme l'héritage).

Un passage à grande échelle aux SGBDOO se fera peut-être un jour, donc, mais nous n'en sommes pas là. Il existe des SGBDOO solides et réellement utilisés en entreprise (pensons à l'immense BD du *Stanford Linear Accelerator Center*[98]) et certains SGBDOO semblent offrir des performances de très haut niveau[99], mais le marché de masse demeure relationnel : le modèle est connu, stable et fait le travail auquel on s'attend.

L'existence de pointeurs ou de références entre les objets qui doivent se connaître réduit, dans les BDOO, le recours aux jointures et tend à accélérer les requêtes conformes au modèle de données du programme entreposé dans la BD. On dit des SGBDOO qu'ils sont *pensés pour la navigation*.

En retour, et c'est là une critique fréquemment faite des SGBDOO, appliquer une requête arbitraire sur les données (plutôt qu'une requête respectant le modèle du programme) tend à être plus lent que dans le modèle relationnel. On dit des SGBDR qu'ils sont *pensés de manière déclarative*.

Il est donc nécessaire de réfléchir à des ponts entre le monde OO et le monde relationnel. Cette réflexivité est un marché en soi.

[98] Voir http://www-db.cs.wisc.edu/cidr/cidr2005/papers/P06.pdf

[99] Voir http://www.objectivity.com/WhitePapers/High_Performance_DB.pdf

Requêtes de type OQL

Il est probable que les SGBDOO, pour être en mesure d'offrir des facilités de requêtes *ad hoc* (comme le permet SQL pour l'approche structurée et pour les SGBD relationnels), doive trouver un point mitoyen entre permettre l'expression de requêtes selon une approche objet[100] et causer un bris total d'encapsulation[101].

Le langage **OQL** se veut une telle alternative à SQL, un compromis entre une approche OO puriste et une approche vouée à être utilisée par un large éventail d'utilisatrices et d'utilisateurs.

Le langage OQL est un standard qui fut définit par le défunt *Object Data Management Group*, ou ODMG[102], et dont la syntaxe et la connectivité avec Java et C++ sont disponibles dans Internet[103]. La syntaxe de OQL rappelle beaucoup celle de SQL et c'est, on le comprendra sûrement, volontaire de la part de ses concepteurs.

Note :	bien que nous mentionnons ici OQL dans le contexte des SGBDOO, rien n'empêche d'autres systèmes d'accès à une BD d'implanter une strate interface OQL malgré une infrastructure sous-jacente qui s'y prêterait moins directement. Ainsi, des SGBD relationnels offrent parfois des interfaces de requête OQL pour faciliter la tâche aux usagers du SGBD privilégiant une approche OO. Ces interfaces sont habituellement restreintes, vu l'absence dans un SGBD relationnel de certains mécanismes OO nécessaires.

[100] … donc subjective, pour laquelle l'objet offre implicitement un support de par ses méthodes.

[101] … pour permettre d'exprimer des requêtes pour lesquelles les objets en place n'offrent peut-être pas de support a priori. Il se peut, en effet, que les utilisatrices et utilisateurs d'un SGBDOO désirent tirer des objets qui y apparaissent des données pour lesquelles ils n'ont pas, *a priori*, été pensés. Et pourquoi pas? Ne s'agit-il pas là l'un des intérêts que comporte l'utilisation d'un SGBD, ou même d'un entrepôt de données?

[102] http://www.odmg.org/

[103] Voir http://www.odmg.org/oqlg.zip et http://www.cis.upenn.edu/~cis550/oql.pdf

Réaliser la persistance dans un SGBDOO

La persistance dans un SGBDOO cherche à conserver l'intégrité des objets entreposés, qu'ils soient simples ou complexes. Elle cherche aussi à y maintenir l'intégrité des relations entre les objets, de même qu'à permettre une échange naturel (à travers des méthodes, à toutes fins pratiques) avec les objets qui s'y retrouvent.

Un problème de base de données auquel font face les traditionalistes du domaine, lorsqu'ils visent à mettre au point une approche objet, est que les objets ne sont pas, règle générale, exprimés dans ce qu'on appelle la *première forme normale* (ou mieux). L'expression habituelle des modèles objets diffère fondamentalement des manières conventionnelles d'exprimer des données structurées dans une base de données.

Un autre problème est que le modèle objet n'est pas un modèle reposant strictement sur les données, mais bien un modèle *mixte* mêlant données et opérations, avec idées d'abstraction, de polymorphisme, de composition, d'agrégation, et ainsi de suite. Le modèle objet est appliqué d'abord dans une optique dynamique, programmée, et tend à être lié, peut-être plus intimement qu'il ne le faudrait, aux langages de programmation avec lesquels il est appliqué.

Ceci mène parfois à des différences de fond d'un point de vue conceptuel entre penseurs de SGBD et penseurs OO, et entraîne parfois des dérives forçant les SGBDOO à penser des extensions (ou à être pensées complètement) en fonction d'un langage spécifique ou d'un groupe de langages particuliers.

Ces disparités ne sont pas souhaitables pour qui veut voir les SGBDOO prendre leur envol.

Normalisation du modèle objet : langage ODL[104] ou autre

Pour en arriver à une représentation d'un modèle objet qui soit indépendante de tout langage de programmation, donc à une manière *neutre* de représenter les classes, les instances et leurs interrelations, le groupe de standardisation ODMG s'est entendu sur le principe d'une norme nommée le *Object Definition Language*, ou ODL[105].

On peut, à l'aide d'énoncés ODL, concevoir des modèles objets qui soient ensuite portables aux différents langages de programmation orientés objet répandus sur le marché, dans les limites de ce que chaque langage permet de représenter, bien entendu. En théorie, on peut aussi constituer une forme ODL des modèles objet conçus dans chacun de ces langages.

L'espoir des SGBDOO, c'est que des modèles ODL puissent être utilisés comme lien entre les approches programmées et les SGBDOO eux-mêmes, éliminant ainsi les différences conceptuelles susmentionnées. C'est une approche prometteuse à plusieurs égards, et qui transcende la question de la persistance.

C'est aussi une approche qui est bousculée et engluée dans une multitude d'impératifs commerciaux et philosophiques. Tout vendeur d'une solution OO voit presque par définition l'avènement d'un véritable ODL comme un cauchemar : implémenter la persistance à travers des modèles ODL impliquerait, pour tout langage OO, la mise en place de mécanismes techniques pour compenser les divers atouts supportés dans ODL mais pas dans le langage OO du vendeur en question.

Au fond, les options pour un véritable ODL sont (a) de ratisser aussi large que possible le modèle OO et offrir la représentation la plus complète qui soit du domaine, ou (b) être par essence incomplet et favoriser un langage ou un autre.

La section ***Concepts ou techniques***, plus loin, cherche à montrer comment certains concepts de certains langages peuvent être transposés en techniques dans d'autres langages. Il peut cependant exister des concepts qui ne sont pas transposables (il n'est pas clair, en particulier, comment on pourrait représenter efficacement les traits en Java ou dans un langage .NET).

Idéalement, il y aurait un équivalent technique dans un langage pour tout concept d'un autre langage. Cela permettrait d'avoir une représentation neutre (ODL ou autre) et d'envisager une véritable forme persistante universelle.

---

[104] Pour en savoir plus, vous pouvez aller consulter le site Web disponible à l'adresse suivante : http://wwwinfo.cern.ch/asd/cernlib/rd45/DRDCproposal/drdc_17.html. Si vous faites des recherches sur Internet, soyez prudent(e) car l'acronyme ODL sert à une multitude de standards, plusieurs d'entre eux étant porteurs du sens *Object Definition Language* mais ayant tout de même un sens différent.

[105] Il existe aussi diverses normes IDL, pour *Interface Definition Language*, permettant de décrire des interfaces au sens OO de manière neutre et indépendante des langages de programmation. La disponibilité d'un IDL est essentielle dans une approche client/ serveur par composants. L'équivalent usuel d'un IDL pour services Web est le langage WSDL (pour *Web Services Definition Language*)

Objets persistants et sérialisation

Dans les approches vues plus haut, la persistance n'est pas tant implantée *sous forme objet* que *pour des objets*. Ce sont dans chaque cas des procédés extérieurs aux objets, donc suivant une approche structurée, en appliquant des opérations sur des objets plutôt qu'en responsabilisant les objets eux-mêmes, qui réalisent l'entreposage des objets persistants.

On peut toutefois penser à une stratégie d'entreposage `OO` qui soit valable pour tous les objets d'un système si certains critères sont respectés.

Le cas des insertions peut être fait implicitement de manière objet dans la mesure où :

- tous les objets à entreposer[106] ont un parent commun, une *racine polymorphique*[107], ce qui est le cas en Smalltalk ou en Java avec les classes `Object` et `Class`, mais pas en C++ par exemple[108], par lequel on peut réclamer une opération subjective d'entreposage; et
- il existe une représentation neutre (`ODL` ou autre) implicite à tout objet qui puisse être utilisée pour que l'entreposage soit fait naturellement et sans douleur pour tout objet.

Si ces deux conditions sont réunies, alors on peut implanter une opération de sérialisation à même la classe racine, qui s'appliquera[109] alors pour tout objet qui soit au moins de la classe racine.

---

[106] On voit ici qu'il est important de considérer une classe comme étant un objet, si on veut devenir capables d'entreposer les classes comme les instances de manière persistante.

[107] Qui pourrait fort bien être une interface au sens strict.

[108] En C++, l'approche est de rendre tous les objets aussi simples à utiliser que le sont les données de types primitifs, et d'accorder un support plus puissant aux objets qu'aux types primitifs. En Java comme en C#, on distingue fondamentalement les types primitifs et les objets, et offrant à chacun un support très différent; la sérialisation des types primitifs se fait en les transformant en objets (explicitement ou par *Boxing*). En *Smalltalk*, il n'y a que des objets. Cela dit, on peut toujours appliquer la sérialisation implicite dans un langage `OO` dans la mesure où on définit une classe racine pour l'opération de sérialisation. En Java ou en *Smalltalk*, les classes `Object` et `Class` (`Object` et `Type` en C#) permettent implicitement cela à travers une racine prédéfinie pour tout objet. En C++, il faut choisir *délibérément* de définir une classe racine pour fins de sérialisation si on désire appliquer le principe, et encore là, *il ne s'appliquera qu'à cette classe et à ses descendants*. Une autre alternative est évidemment d'y aller par programmation générique.

[109] En général par polymorphisme, quoique si le modèle de représentation est assez solide pour le permettre on pourrait penser implanter une méthode générique à la racine qui entrepose la représentation neutre de l'objet pour fins de persistance sans nécessiter d'aide de ses descendants, du moins pour les descriptions qui ne relèvent pas directement d'attributs d'instance. Une autre approche (beaucoup moins `OO`) est d'appliquer la réflexion et de disséquer les objets à sérialiser et à *désérialiser* pour que la persistance soit pleinement prise en charge par un moteur servant d'assistant à cette mécanique.

Principes de sérialisation

L'idée de sérialisation est de définir une méthode qui permettra de représenter, de manière facile à entreposer (et, éventuellement, à reconstituer), l'objet à entreposer, à *sérialiser*. Tel qu'indiqué à la section précédente, cette représentation pourrait suivre la norme ODL, mais elle pourrait aussi être autre chose (comme une forme XML textuelle, ressemblant par exemple au populaire protocole SOAP).

Il est à souhaiter que le standard utilisé soit le plus portable et le plus ouvert possible, pour faciliter les interactions directes sans égard aux langages de programmation utilisés. Ceci explique en partie la popularité des notations XML pour les représentations d'objets au cours des dernières années. Cela dit, je doute qu'on souhaite qu'afficher 3 à la console donne <integer>3</integer> ce qui implique que même XML ne soit pas le format universel souhaitable.

Pour la sérialisation, subjective, le format appliqué par un objet pour s'écrire se projeter sur un flux importe peu, mais la désérialisation n'est pas un processus subjectif et le moteur utilisé pour cette tâche doit être capable de reconnaître au minimum le début et la fin de la description sérialisée de chaque objet s'il veut être en mesure d'accomplir sa tâche.

Selon cette approche, pour entreposer un objet sur un flux, on lui demandera de se sérialiser sur ce flux. ***Se sérialiser*** signifie encoder sous une forme appropriée sa propre représentation, incluant l'appartenance à une classe et l'identité des objets avec lesquels on interagit.

Pour un objet, *se sérialiser* ressemble (pour l'entreposage, du moins) à *construire sa représentation* : tout objet, pour se sérialiser, sérialisera d'abord ses parents, puis sérialisera ce qui le distingue, ce qui le rend spécial. Si le langage utilise implicitement une représentation prête à être sérialisée des objets, comme le font Java et .NET, alors l'entreposage devient presque une tâche banale.

Ce qui sera sérialisé

Les langages proposant une sérialisation intégrée (*Built-In*), comme Java et les langages .NET, le font parce que la structure de chaque classe et de chaque objet est une donnée disponible à même le langage, et manipulable comme telle[110].

Dans ce cas, tout objet peut être sérialisé dans la mesure où ses attributs peuvent être sérialisés eux aussi.

Le langage Java offre une manière d'empêcher un attribut d'être sérialisé tout en permettant à l'objet qui le contient d'être sérialisé quand même. Pour ce faire, il faut spécifier l'attribut à ne pas sérialiser comme étant transient. Ceci peut être utile lorsqu'un objet contient des données confidentielles, comme un mot de passe ou un numéro de carte de crédit par exemple.

Le moteur .NET fait de même avec la métadonnée [Transient].

La sérialisation intrinsèque comporte un immense avantage : les détails structurels propres au langage, comme par exemple l'appartenance à une classe et la représentation des méthodes, sont *de facto* sérialisées de manière standard.

Ce qu'on entreposera lors d'une sérialisation sera :

- la structure descriptive de la classe à laquelle appartient l'instance en cours de sérialisation (au moins une fois pour l'ensemble des instances, au pire une fois par instance);
- les attributs et leur valeur au moment de la sérialisation; et
- une manière de distinguer les objets les uns des autres, une sorte d'identifiant unique.

[110] … ce qui est aussi une condition d'implantation de la réflexion au sens OO, couverte plus haut.

Habituellement, les constantes de classe ne sont pas sérialisées par ces moteurs, du fait qu'ils présument que le programme qui *désérialisera* les objets possède cette information au préalable. Ceci pose un problème d'interopérabilité dans le cas où un programme écrit dans un autre langage que le langage d'origine voudrait se saisir des objets sérialisés.

Lorsque la mécanique du langage le permet, une représentation des relations entre les objets sera aussi entreposée, question de faciliter la reconstruction éventuelle de chaque objet et de sa manière de coexister avec les autres objets du système[111].

Procéder ainsi est plus simple quand chaque objet possède une identité propre, un identifiant unique qui permet de le distinguer d'autres objets possédant les mêmes attributs.

Lorsque la mécanique du langage n'inclut pas spécifiquement de représentation tangible de chaque classe et de chaque instance en mémoire, comme c'est le cas en C++, on peut implanter la sérialisation par polymorphisme (via une interface au sens strict, dévoilant une méthode de sérialisation) ou via des opérateurs d'écriture sur un flux.

Un code de *hashage* ne suffirait pas ici pour déterminer l'unicité de l'identité d'un objet, car si deux objets identiques devraient avoir des codes de *hashage* identiques, il est aussi possible que deux objets différents aient à l'occasion le même code de *hashage* (une collision; ça fait partie des affres du hashage).

Dans le monde C++, attribuer de manière intrusive un identifiant unique pour tout objet du langage est une charge philosophiquement inacceptable. La norme y est, après tout, qu'un programme ne doit pas payer pour ce dont il ne se sert pas. Une tentative de sérialisation de réseaux d'objets pourrait toutefois y attribuer, au moment de la sérialisation, des identifiants uniques sur la base de leurs adresses. Cela dit, quand un objet peut exposer plusieurs visages (dans un cas d'héritage multiple, par exemple) et quand divers objets pointent sur diverses interfaces d'un même objet, la problématique peut devenir complexe.

Dans ce cas, chaque objet est responsable de savoir s'écrire lui-même sur le flux; ce n'est pas le langage qui prend en charge la représentation des objets.

Problème de la désérialisation

Réexaminons le petit programme sérialisant, via des opérateurs d'écriture et de lecture sur un flux, des instances de `ChicType`.

L'entreposage est une étape délibérée. Lorsqu'on écrit un objet sur un flux, cet objet existe déjà, et on lui demande, à toutes fins pratiques, de se projeter lui-même sur un flux. Ce n'est pas le flux qui agit sur l'objet, mais bien l'objet sur le flux, même si l'opérande de gauche est le flux.

```
#include <fstream>
#include <sstream>
// ...using, etc.
stringstream sstr;
ChicType ct0, ct1;
sstr << ct0;
sstr >> ct1;
ofstream{"f.dat"} << ct0;
ifstream{"f.dat"} >> ct1;
```

La reconstruction d'un objet est, malgré les apparences, d'un autre acabit. Pour reconstruire un objet pris d'un flux, il faut connaître d'avance le type de l'objet à reconstruire, ou encore avoir entreposé un identifiant sur le flux qui permet de reconnaître le type d'objet à reconstruire, et encore là, on ne peut reconstruire que les objets dont on connaît *a priori* l'existence, desquels on connaît la classe ou desquels on peut obtenir la classe.

---

[111] Pour voir comment Java procède, voir le site Web disponible à l'adresse suivante: http://developer.java.sun.com/developer/technicalArticles/RMI/ObjectPersist/. Pour voir comment des solutions plus neutres, plus dissociées d'un langage de programmation spécifique, sont possibles, voir aussi http://www.xml.com/pub/a/1999/09/serialization/. L'implémentation choisie pour une bibliothèque commerciale bien connue (la bibliothèque `MFC`) est expliquée à l'adresse suivante : http://staff.develop.com/onion/Articles/cpprep1295.htm.

Dans notre exemple, tout fonctionne simplement parce que le type de chaque objet entreposé est connu du code servant à la reconstruction.

Aussi, remarquez que si, au moment de l'écriture sur un flux, les surcharges et le polymorphisme, selon les mécaniques utilisées, rendent transparente l'opération de sérialisation, ces mécanismes ne sont d'aucun secours lors de la reconstruction des objets.

Si le flux contenait un identifiant propre au type de l'objet entreposé, il faudrait y aller d'une série d'alternatives (ou d'une sélective) pour identifier le type de reconstruction à appliquer. *Le polymorphisme à l'envers, quoi*.

L'autre problème est la reconstruction des relations entre les objets. Doit-on entreposer, pour chaque objet, son adresse en mémoire au moment de l'entreposage[112]? La mécanique d'un langage donné doit-elle suppléer intrinsèquement un identifiant unique à même chaque objet pouvant être sérialisé?

Problème plus complexe encore : comment garantir qu'aucun objet déjà en mémoire dans le programme procédant à une *désérialisation* n'ait le même identifiant unique qu'un objet déjà sérialisé, entraînant ainsi un conflit relationnel lors d'une éventuelle reconstruction?

On le voit clairement, il suffit (par exemple) qu'un même programme entrepose un objet sur disque puis le récupère dans une autre variable pour que, si une dose suffisante de prudence n'est pas appliquée, deux objets apparaissent en mémoire munis du même identifiant. Leur unicité détruite, le maintien de leur place (et, peut-être, de leur rôle) dans une toile de relations devient beaucoup plus risqué.

Reconstruire les objets n'est pas banal. C'est un problème auquel il est raisonnable d'apposer une solution locale, mais très difficile de mettre au point une solution universelle, surtout si cette solution doit être efficace et élégante.

[112] … en présumant cette adresse unique, donc nonobstant toute stratégie de mémoire virtuelle par laquelle des objets auraient pu être déplacés de la mémoire au disque et inversement.

Désérialisation et bris d'encapsulation : Java et `instanceof()`

La sérialisation intrinsèque de langages comme Java facilite en partie la reconstruction d'un objet du bon type.

En Java, on a droit à l'opérateur `instanceof()`, qui permet de comparer le type d'une instance sérialisée avec tout autre type, chaque classe ayant une représentation objet à même le système.

En récupérant un objet d'un flux et en comparant cet objet avec la liste des types possibles (une séquence d'alternatives y allant de comparaisons utilisant `instanceof()`), on peut donc reconstituer chaque objet d'un flux donné, présumant qu'on connaisse au préalable la liste exhaustive des classes possibles.

> En Java, `a` est au moins une instance de la classe `c` si l'expression `a instanceof(c)` est vraie. C# offre l'opérateur `is` pour le même effet, alors que `VB.NET`, de son côté, offre `Is`.
>
> Tout comme l'opérateur `dynamic_cast` du langage C++, `instanceOf()` ne permet pas de savoir si `c` est la classe terminale ou l'un de ses parents.
>
> Il ne faut pas abuser des comparaisons explicites de types comme celles engendrées par l'utilisation de `instanceof()`, qui sont en fait des bris d'encapsulation. La plupart des raisons possibles pour utiliser ce mécanisme auraient une meilleure solution, au sens de l'élégance comme au sens de l'efficacité, par polymorphisme

Le résultat sera plus précis si les tests vont du plus spécifique au plus général, surtout si on considère que Java ne supporte que l'héritage simple, mais la plupart du temps le programme procédant à la *désérialisation* ne testera que les cas qui sont pertinents à l'utilisation qu'il compte faire de l'objet (peut-être ne testera-t-il qu'une série d'interfaces dont il compte se servir).

Le support intrinsèque de la sérialisation par Java et la présence dans ce langage d'un mécanisme complet de réflexivité a ce bon côté que les outils pour lire un objet d'un flux sont capables d'y reconnaître le début et la fin de l'objet, de sorte qu'il ne reste plus au processus reconstruisant un objet qu'à en vérifier la classe effective et à l'utiliser comme tel via une conversion explicite de type. Ceci ne peut pas nécessairement être garanti par des mécanismes implantés explicitement.

Remarquez que la reconstitution des relations demeure problématique. Il ne s'agit pas d'un problème élémentaire et pour lequel il existe, dans le moment, une solution générale.

Désérialisation et bris d'encapsulation : C++ avec `RTTI` et `dynamic_cast`

En langage C++, on peut implémenter un mécanisme polymorphique de sérialisation/ *désérialisation* dans la mesure où tous les objets sérialisés dérivent d'un même ancêtre, déterminé par le système à sérialiser. L'identité de la racine de sérialisation doit être connue du module procédant à l'entreposage comme de celui qui le reconstruira.

On peut appliquer, avec C++, la même stratégie qu'avec Java, dans la mesure où on a pris soin d'inclure à la compilation ce qu'on appelle le *Run-Time Type Information*, ou `RTTI`. Utiliser `RTTI` accroît la taille des modules compilés en accroissant la taille de chaque objet, ce qui explique que ce mécanisme soit optionnel.

Utiliser `RTTI` en C++ ajoute au système des instances d'une classe système nommée `std::type_info`, qu'on peut accéder directement en incluant le fichier d'en-tête standard `<typeinfo>`. Cette classe permet de dévoiler sous forme textuelle brute (`const char*`) le nom de la classe à laquelle appartient une instance donnée.

Les outils d'écriture et de lecture sur un flux de C++ ne permettent pas de reconstruire un objet au sens abstrait en mémoire, la représentation des types n'étant pas une structure objet en soi.

Le langage C++ ne permet l'inclusion de `RTTI` que sur les instances de classes pour lesquelles on retrouve au moins une méthode polymorphique (au moins une méthode virtuelle).

On peut comparer la classe de deux instances en utilisant l'opérateur `typeid()`. Ainsi, `a` et `b` sont des instances d'une même classe si l'expression `typeid(a) == typeid(b)` est vrai.

En C++, on préférera en général l'utilisation de `dynamic_cast` à celle de `typeid()` pour évaluer si on peut considérer un objet comme une instance d'une classe donnée. Cela dit, en situation de reconstruction suite à une sérialisation, la meilleure stratégie de comparaison dépendra des stratagèmes d'entreposage utilisés.

De toute manière, si le polymorphisme a bien été implanté au sein d'une hiérarchie de classes, le type exact d'une instance donnée nous importe peu; savoir les opérations supportées par cette instance est bien plus important. Et sur le plan opérationnel, répondre à la question *cet objet peut-il faire ceci?* équivaut à répondre à la question *cet objet supporte-t-il cette interface?* ou, de manière équivalente, *cet objet est-il au moins un dérivé de cette classe?* Toutes des questions auxquelles répond plus adéquatement `dynamic_cast` que `typeid()`.

La mise en garde quant aux dangers des abus de `instanceof()` en Java s'appliquent aussi pour l'utilisation de `RTTI` en C++. La plupart du temps, s'il existe une solution tenant du polymorphisme à un problème pour lequel on pourrait utiliser le `RTTI`, celle où on utilise le polymorphisme sera pratiquement toujours nettement meilleure

Il est possible en C++ de tirer de l'information sur un type donné au moment de la compilation, ce qui permet de générer des algorithmes génériques plus efficaces (par exemple, il est possible de vérifier si un type `T` est `const`; s'il s'agit d'une référence; s'il est `volatile`; *etc.*).

Persistance et réflexivité

La sérialisation est facilitée chez les moteurs OO qui supportent le concept de réflexivité, donc la capacité pour les objets de se décrire structurellement par des objets et d'offrir une méthode qui révèle un objet représentant leur classe, leurs ancêtres, les interfaces qu'ils exposent, leurs constructeurs, leurs autres méthodes, leurs attributs, *etc*.

La réflexivité ne donne pas la valeur des attributs d'un objet, et ne permet donc pas à elle seule de sérialiser diverses instances sur un flux. Dans une optique OO stricte, la seule entité jugée capable de produire une représentation d'un objet qui puisse être sérialisée devrait être l'objet lui-même, suivant le principe d'encapsulation.

En retour, la réflexivité décrit la structure d'une classe en termes d'objets eux-mêmes *sérialisables*, ce qui permet à un moteur connaissant au moins des objets d'offrir un support plus complet de la reconstitution des objets lors de leur *désérialisation*.

Le support de la réflexivité dans un moteur OO inclut habituellement la capacité de charger la représentation OO des méthodes à partir de leur nom et d'invoquer ces méthodes à partir de leur représentation. Cela permet entre autres d'invoquer des constructeurs et de compléter l'automatisation de la *désérialisation*.

Persistance et contrôle des versions

Toute mécanique de persistance rencontrera éventuellement un problème presque inéluctable : *le passage du temps*. Les langages évoluent, les structures internes des objets aussi. Un objet peut rester utilisable sur une longue période par d'autres objets si son design est solide, si son encapsulation est stricte, et si sa barrière d'abstraction—la signature de ses méthodes publiques—demeure stable. La sérialisation est une bête d'un autre ordre que celui de l'utilisation, toutefois : c'est une projection sur un flux de données d'une séquence de *bytes* décrivant la structure interne de l'objet, les valeurs de ses attributs, la nature de ses relations, *etc*.

L'objet sérialisé peut rester sur un flux, un fichier, une BD… pour une durée arbitrairement longue. Rien ne garantit que les tentatives de désérialisation d'un objet sérialisé soient faites avec une vision cohérente de cet objet. Tout changement de représentation interne de l'objet risque de briser la capacité de récupérer d'un flux des versions antérieures du même objet. *La persistance pose un problème de cohérence qui outrepasse les protections érigées par la mise en application d'une encapsulation stricte*.

Des objets se voulant persistants doivent se faire un point d'honneur de mettre sur pied un schème interne de numéro de version. Une implémentation supportant la version v_i d'une classe peut ne pas être capable de *désérialiser* une instance de la même classe si celle-ci a été entreposée à l'aide d'un outil supportant la version v_j si $i>j$. Une classe voulant supporter la *désérialisation* devrait prudemment garder à portée de main le code de *désérialisation* de versions antérieures[113], juste au cas, et prendre des dispositions pour ajuster l'objet *désérialisé* de manière à faire face aux exigences de la nouvelle structure.

[113] … et souhaiter que toutes les classes en relation avec elle, de manière transitive, fassent de même!

Autres problèmes liés à la persistance des objets

Plusieurs autres petits (et grands) problèmes liés à la persistance existent, sans égard aux langages de programmation et aux approches.

Certains sont en partie solubles par l'emploi d'une norme neutre de représentation des objets. Ces problèmes tendent à être liés à la structure interne des objets, et incluent une résolution partielle du problème de l'identité des objets, de la représentation de la structure d'une instance comme de celle d'une classe, de la représentation des méthodes, et de la représentation des idées d'héritage simple ou multiple comme de polymorphisme.

D'autres sont en partie solubles par l'emploi d'une norme neutre telle qu'UML pour représenter des relations entre les objets et de représentation de l'existence d'objets. Ces problèmes tendent à être ceux qui outrepassent l'existence de l'objet en tant qu'objet et touchent plus son existence comme partie d'un système. Des trucs comme représenter une toile de relations ou un rôle dans un schéma, par exemple.

La persistance est un idéal, dont l'atteinte sera possible pour qui parvient aussi à une solution standard de problèmes comme les suivants :

- lors de la copie d'un objet, que faire des relations qu'il entretient avec d'autres objets? Copiera-t-on en cascade la toile de ses relations (copie en profondeur, ou *Deep Copy*), ou ne copiera-t-on plutôt que les relations elles-mêmes (copie en surface, ou *Shallow Copy*), évitant ainsi une duplication des objets avec lesquels l'objet copié interagit? Les deux situations sont possibles, et on souhaite pouvoir décider de manière standard, à la compilation ou à l'exécution, entre les deux schèmes, peut-être pour chaque relation prise individuellement;
- comment empêcher de manière stricte la duplication d'un singleton dans un système suite à la reconstruction d'un objet sérialisé?
- lors de la suppression d'un objet, que faire de ses relations? La problématique s'apparente ici à celle ayant lieu lors de copies d'objets, et est sensiblement du même niveau de complexité. Les habitués de systèmes client/ serveur le savent: il y a des situations pour lesquelles seul un objet lui-même peut indiquer qu'il peut (ou doit) être supprimé;
- quel est le sens d'un objet une foi celui-ci extrait de son environnement d'origine? Cette question permet-elle-même une réponse?

Un modèle uniforme de persistance sera sans doute fait de compromis. On n'a qu'à penser à la définition moderne du mot objet, véhiculée par la norme UML, qui est plus restrictive que la définition traditionnelle. On peut penser aussi à l'héritage tel que proposé par les dévots du modèle à héritage simple seulement ou à celui proposé par les amateurs d'héritage multiple, virtuel ou non. Est-il même possible d'en arriver à une représentation neutre, indépendante du langage, mais qui réconcilie les divers points de vue?

La persistance est un idéal lié à plusieurs questions de standardisation du modèle objet, et qui se bute conséquemment à des considérations philosophiques; tous les langages sont porteurs de philosophie, après tout. Cet idéal ratisse large, voulant en arriver à une modélisation aussi inclusive que possible. Au moment d'écrire ces lignes, la persistance des objets au sens large est, au sens globalisant du terme, un problème ouvert.

Concepts ou techniques

Dans le monde de l'informatique, les produits sont nombreux. Il en va de même pour les langages, donc pour les philosophies et, presque inévitablement, les dogmes.

On peut, à mon avis, affirmer sans ambages que la plupart des langages de programmation à vocation OO récents[114] sont en partie proposés comme outils et en partie comme écoles de pensée. Même en faisant exception des catégories commerciales pour lesquelles chacun est foncièrement à son avantage[115], et en examinant chaque langage pour son offre conceptuelle, une chose fondamentale transparaît :

⇒ ces langages, pris en tant que langages, ne se distinguent pas tant par ce qu'ils permettent d'exprimer que par la facilité avec laquelle ils permettent de le faire.

Cette affirmation pourrait se traduire en un slogan plus compact : *les concepts de l'un sont les techniques de l'autre*. En restreignant notre regard aux langages en tant que langages, on peut prétendre que ce que l'un propose sous forme de concept apparaît chez l'autre sous forme de technique, et qu'on pourrait, avec effort, aplanir par des techniques de programmation bien des différences de fond entre eux.

> Cette capacité de représenter par des techniques dans un langage donné ce qu'un autre exprime par des concepts est à la base de la volonté de persistance pleinement portable d'un langage OO à l'autre.

Ces diverses techniques représentent avec une élégance variable les concepts des langages compétiteurs. Le choix esthétique pour les qualités des langages en tant que langages devient une question d'évaluer quelles contorsions nous semblent les plus élégantes, les plus acceptables pour offrir, au besoin, les fonctionnalités qui apparaissent en tant que concepts dans d'autres langages.

Je doute qu'on puisse tracer un portrait complet des concepts de chaque langage et qu'on puisse cartographier un éventail complet et rigoureux des techniques qui permettent d'en arriver à une équivalence fonctionnelle ailleurs. Ainsi, prenez cette section comme une amorce de réflexivité sur le sujet, et comme un exposé de quelques pistes pouvant mener à des pratiques compensatoires pour des éléments conceptuels que vous appréciez peut-être *par ici* et regrettez de ne pas rencontrer *par là*.

---

[114] Pensons à C++, à Java, aux langages .NET, et sûrement à d'autres (Python et Ruby par exemple), bien que cet échantillon suffise pour la démonstration proposée ici.

[115] Pour illustrer *grossièrement*, pensons à la puissance et à la polyvalence avec C++, à la portabilité avec Java, et à la qualité des outils de développement avec les produits .NET. On pourrait prétendre à une forme de prise de position objective dans chaque cas, et encore – il y aurait malgré tout des tensions philosophiques justifiables entre les *aficionados* de chaque technologie.

Constantes et immuabilité

C++ offre un soutien aux instances constantes. Pour une même classe, deux instances distinctes peuvent être l'une modifiable, l'autre non modifiable.

Par exemple, dans l'exemple proposé à droite, `INVITE` est qualifiée `const`, ce qui signifie qu'entre la fin de sa construction et le début de sa destruction, le compilateur prendra des dispositions pour empêcher, de manière statique, toute opération susceptible d'en modifier les états. L'objet `s`, lui, est variable et peut être modifié à loisir.

```
#include <string>
#include <iostream>
int main() {
   // ...using...
   const string INVITE="Allo!";
   cout << INVITE;
   string s;
   if (cin >> s)
      cout << s;
}
```

Java (`final`) et C# (`const`) offrent tous deux un support à la constance des primitifs et des références, mais pas à celle des objets. C# supporte des attributs `readonly`, ce qui se rapproche de l'idée d'attribut constant, mais leur utilisation impacte fortement la vitesse d'exécution des programmes.

Les langages ne supportant pas la constance des instances sur une base individuelle peuvent pallier cette déficience en partie, du fait qu'il est possible (si des concepts comme celui des membres privés existent dans le langage, ce qui exclut Python par exemple) de définir des classe immuables, donc n'offrant aucun service permettant de modifier leurs instances une fois ces dernières construites.

Ce faisant, on trouve parfois deux classes cousines, l'une modifiable (et rapide, mais peu appropriée pour la surface publique des objets si ceux-ci sont tous manipulés indirectement) et l'autre immuable, plus lente d'utilisation mais appropriée pour révéler des états d'un objet.

La classe Java `Entier` proposée à droite est immuable. Si elle offrait un mutateur, elle ne le serait plus.

```
public class Entier {
   private int valeur;
   public Entier(int valeur) {
      this.valeur = valeur;
   }
   public int getValeur() {
      return valeur_;
   }
}
```

En Java, la plupart des classes correspondant à des types primitifs (par exemple, la classe `Integer`) sont immuables. Le duo le plus célèbre de classes, l'une immuable et l'autre non, est fait des classes `String` (immuable) et `StringBuffer` (modifiable); dans le monde `.NET`, on retrouve l'équivalent avec `String` et `StringBuilder`.

Pour faciliter le passage d'une classe immuable à sa contrepartie modifiable (et inversement), ces classes organisées par paires exposent typiquement chacune un constructeur prenant en paramètre une instance de l'autre.

Nous y reviendrons sous ***Objets constants***, plus loin.

Encapsulation de premier niveau des attributs sans propriétés

Ce cas est sûrement le cas le plus simple de tous ceux que je me propose de couvrir ici. Tel que mentionné dans [POOv00], dans les langages qui supportent les propriétés (VB6, Delphi, les langages .NET), une propriété sert à associer implicitement un mutateur, un accesseur ou les deux à un attribut. Plus précisément, une propriété offre une barrière d'abstraction qui apparaît comme modifiable par affectation et consultable sous forme de valeur, qu'il y ait un attribut ou non d'associé à la propriété.

Visiblement, une propriété est une formalisation simple d'un couple accesseur/ mutateur (par nécessairement de premier ordre) à valeur unique. C'est ce qu'on nomme souvent du sucre syntaxique[116]: On ne pourrait pas, par exemple, remplacer un `Set` à deux paramètres par une propriété[117]. L'emploi de propriétés est donc une extension esthétique mais limitée du concept d'encapsulation primitif véhiculé par les couples `Get/ Set` les plus simples.

Là où l'emploi de méthodes préfixées par `Set` et `Get` (ou par `set` et `get`, selon la norme Java) pour couvrir d'une strate primitive un attribut[118] est une convention, une discipline de programmation au même sens que l'utilisation de majuscules pour nommer les constantes ou au même sens que l'emploi de verbes comme noms de sous-programmes, les propriétés sont, dans les langages qui les offrent, des concepts du langage de programmation. En conséquence, le compilateur peut leur réserver un traitement spécial.

Dans les langages .NET, s'inscrivant dans la foulée du développement Delphi ou du développement C++ avec la bibliothèque MFC, on utilise les propriétés comme interface privilégiée pour le développement de composants. La thématique des composants se rapporte surtout aux systèmes client/ serveur (SCS), mais signalons qu'un composant est une entité logique qui se comporte comme un objet, et qui repose à la fois sur une encapsulation stricte et sur une utilisation exhaustive du polymorphisme.

Les composants se veulent interopérables sur le plan binaire : on peut rédiger un composant dans un langage, le compiler puis l'utiliser dans un autre langage.

---

[116] Le terme vient du concepteur de Delphi et de C#, **Anders Hejlsberg**, qui prétend (et il n'a pas tort) dans une entrevue officielle [HejlMS] que bien des éléments de la POO sont, en fait, du sucre syntaxique. On pourrait aller jusqu'à dire que le propos de cette section est, du début à la fin, une discussion de ce qu'on peut faire en l'absence de sucre syntaxique.

[117] Notez que C# offre un enrobage spécial pour la notation indicée des tableaux; on nomme d'ailleurs cet enrobage un *indexeur*. Nous y reviendrons plus loin. C# est sûrement l'un des langages commerciaux dans lesquels les programmeuses et les programmeurs sont les plus dépendants de support spécial à des cas particuliers et de sucre syntaxique de la part du compilateur.

[118] Lorsque la paire `Set/ Get` encapsule un attribut, bien sûr, ce qui repose aussi entièrement sur une discipline de programmation.

Cette interopérabilité primitive a servi, dans le monde *Microsoft*, à la construction d'éditeur visuels d'interfaces personne/ machine où des composants compilés au préalable sont insérés dans une fenêtre puis personnalisés en insérant des valeurs dans ce qu'on nomme une fenêtre des propriétés du composant : une table de paires nom/ valeur déduites de la liste des propriétés de l'objet.

Notez qu'en l'absence de propriétés en tant que concept, réaliser un mécanisme comme la fenêtre des propriétés d'un composant demande presque la présence d'un mécanisme de réflexivité permettant de déduire la structure d'un objet une fois celui-ci chargé en mémoire.

L'absence de propriétés dans un langage ne signifie pas qu'il soit impossible de mettre en place de telles techniques de développement. Les *Beans* de Java font en fait précisément la même chose, c'est-à-dire représenter des composants par des instances de classes respectant certaines contraintes et une encapsulation primitive à partir d'une paire de méthodes `set/ get`.

La fonctionnalité exposée par les paires accesseur/ mutateur classiques est supérieure à celle des propriétés. Elle est plus complète, plus homogène avec le reste du système, ce qui évite de segmenter certaines méthodes d'instance d'un objet en les plaçant dans une catégorie somme toute artificielle.

Les propriétés formalisent une partie des paires accesseur/ mutateur. Ce concept est donc très facilement représentable et réalisable avec des produits compétiteurs. Le contraire est moins vrai : des accesseurs exigeant un ou plusieurs paramètres à l'appel et des mutateurs à plus d'un paramètre[119] ne peuvent être directement représentés par des propriétés. Conséquemment, les langages offrant les propriétés doivent supporter deux familles distinctes de contrôles primitifs d'accès aux attributs.

Les propriétés peuvent être agréables pour qui en aime la syntaxe, mais représentent une strate conceptuelle supplémentaire à assimiler puisque, dans le cas général, elles ne suffisent pas à couvrir tous les cas.

Pour couvrir une partie des cas (pas tous!) que les propriétés ne couvrent pas, les langages `.NET` offrent d'autres strates de sucre syntaxique[120] qui apparaissent comme des concepts sous `.NET` et qui apparaissent comme des techniques dans d'autres langages. Nous examinerons quelques-uns de ces cas plus bas.

Voir [hdProp] pour un article expliquant comment exprimer des propriétés sophistiquées en C++.

[119] …comme on en voit par exemple si on désire contrôler l'accès élément par élément à un attribut indexé comme un tableau interne à un objet, ou lorsqu'on veut forcer l'emploi d'une forme de certificat d'authentification lors d'appels à des méthodes (un `Get` sécurisé par mot de passe, par exemple).

[120] Les *indexeurs*, par exemple, tel que mentionné précédemment, ou la construction implicite de listes de paramètres prise en charge par le langage, ce sur quoi reposent entre autres les entrées/ sorties standard à la console.

Indexeurs

L'idée d'indexeur, chère à C#, est de permettre d'indexer un objet de la même façon qu'on indexerait un tableau. En C#, d'ailleurs, les indexeurs ont un support syntaxique particulier au niveau des propriétés.

Un exemple d'indexeur C# pour une classe `Tableau` irait comme suit :

```
using System;
namespace peuimporte {
   class Tableau {
      // C# ne permet que des constantes de classe
      private const int TAILLE = 100;
      private int[] tableau;
      public Tableau() {
         tableau = new int[TAILLE];
      }
      // propriété indexeur (remarquez le this: c'est l'instance de Tableau
      // elle-même qui peut être indicée
      public int this[int n] {
         get {
            if (n < 0 || n >= TAILLE) throw new Exception("Hors bornes");
            return tableau[n];
         }
         set {
            if (n < 0 || n >= TAILLE) throw new Exception("Hors bornes");
            tableau[n] = value;
         }
      }
   }
   public class Test {
      public static void Main() {
         Tableau t = new Tableau();
         // utilisation de l'indexeur en écriture
         for (int i = 0; i < 20; i++)
            t[i] = i + 1;
         // utilisation de l'indexeur en lecture
         for (int i = 0; i < 20; i++)
            System.Console.WriteLine("Élément {0}: {1}", i, t[i]);
      }
   }
}
```

On peut indicer un indexeur avec d'autres types que des entiers.

En C++, on obtient le même résultat (et plus) avec l'opérateur `[]` :

```
class Tableau {
   enum { TAILLE = 100 };
   int tableau[TAILLE] {};
public:
   Tableau() = default;
   class HorsBornes {};
   // lecture seule
   int operator[](int n) const {
      if (n < 0 || n >= TAILLE) throw HorsBornes{};
      return tableau[n];
   }
   // lecture et écriture
   int& operator[](int n) {
      if (n < 0 || n >= TAILLE) throw HorsBornes{};
      return tableau[n];
   }
};
int main() {
   Tableau t;
   // accès en écriture
   for (int i = 0; i < 20; i++)
      t[i] = i + 1;
   // accès en lecture
   for (int i = 0; i < 20; i++)
      cout << "Element " << i << ": " << t[i] << endl;
}
```

On peut, ici aussi, utiliser autre chose qu'un entier pour servir d'index.

En Java, il est difficile d'utiliser l'équivalent d'un indexeur sauf à l'aide de méthodes spéciales. Comme en C#, aucun support pour l'opérateur `[]` n'existe, et comme en C++, il n'y a pas de sucre syntaxique fait spécifiquement pour le simuler. Il ne reste qu'à exprimer une méthode de recherche d'éléments à partir d'un indice (au sens large : entier, chaîne de caractères ou autre) pour arriver au même résultat en simulant l'opération.

Remarquez que la plupart des langages mettent aujourd'hui l'accent sur les itérateurs, et ce pour la majorité des opérations sur des conteneurs. Le recours à la notation indicée est utile en programmation contemporaine, mais moins qu'elle l'a déjà été.

Listes de paramètres

Certains langages (C, qui n'est pas OO du tout, mais aussi VB6 et les langages .NET) supportent le concept de liste de paramètres, au sens de sous-programmes acceptant un nombre variable de paramètres.

Pour exploiter ce mécanisme en VB.NET, on doit utiliser un paramètre de type Object() (tableau de Object()) préfixé par le mot clé ParamArray, alors qu'on privilégiera en C# un object[] préfixé du mot clé params. En langage C, on peut faire de même avec ce qu'on nomme des ellipses (la séquence ...), et ces fonctions sont dites *variadiques*[121]. Depuis C++ 11, C++ supporte quant à lui les *templates* variadiques.

Il est fréquent, dans un cas comme dans l'autre, qu'un paramètre du genre soit précédé d'une description des paramètres (souvent une chaîne de caractères), pour permettre au sous-programme appelé de comprendre la nature des paramètres reçus.

Le fait que les langages .NET utilisent comme tableaux des objets à part entière, conscients de leur propre taille, permet d'éviter des débordements accidentels comme on en voit en C.

Cela dit, la dépendance envers une chaîne de caractères (ou un équivalent) pour dicter le format des objets à traiter, empêche le compilateur de prendre en charge les types et augmente les risques de comportement incorrect à l'exécution.

```
const short NO_ANNEXE = 18;
string auteur = "un chic prof";
Console.WriteLine
   ("Ceci est l'annexe {0}; auteur: {1}",
    NO_ANNEXE, auteur);
```

```
const short NO_ANNEXE = 18;
string auteur = "un chic prof";
Console.WriteLine
   ("Ceci est l'annexe {0}; auteur: {1}",
    NO_ANNEXE); // boum!
```

Une des choses qui rendent possible ce type de construction est que C# impose (comme Java) que toutes les classes dérivent directement ou indirectement de System.Object, et que toute instance de Object peut être convertie (par polymorphisme) en chaîne de caractères. Java exploite une stratégie similaire mais sans l'emploi de chaînes de formatage ou de paramètres en nombre variable.

On peut donc, sans que le compilateur ne puisse nous assister, passer un nombre trop élevé de paramètres. C'est là la principale raison pour laquelle C++ prend ses distances de C pour cette stratégie et y va par opérateurs binaires plutôt que par chaînes de caractères (comme le faisaient les fonctions printf() et scanf() de C, et comme le font plusieurs produits *Microsoft* incluant les entrées/ sorties de la plate-forme .NET): cela permet au compilateur de nous aider.

[121] En langage C (et en C++), ce mécanisme est l'ellipse et est un mécanisme dangereux à plusieurs égards (en plus d'être dysfonctionnel avec des types autres que les types primitifs) mais qui sert aujourd'hui à certaines manœuvres subtiles de métaprogrammation. Nous disposons aujourd'hui de techniques sophistiquées qui nous permettent d'éviter les ellipses dans l'immense majorité des cas.

Par exemple, ceci ne générera pas d'erreur à l'exécution mais peut être considéré comme une erreur de logique : l'utilisateur passe deux paramètres suite à la chaîne de formatage, et a sans doute des attentes qui ne sont pas rencontrées lors de l'exécution du programme.

```
const short NO_ANNEXE = 18;
string auteur = "un chic prof";
// Pas d'erreur à l'exécution, mais une erreur
// de logique qui passe inaperçu...
Console.WriteLine
   ("Ceci est l'annexe {0}", NO_ANNEXE, auteur);
```

L'inverse (passer un nombre insuffisant de paramètre), par contre, générera une erreur à l'exécution. Le compilateur n'est d'aucune utilité dans de tels cas.

```
const short NO_ANNEXE = 18;
string auteur = "un chic prof";
// paramètre manquant... boum!
Console.WriteLine
   ("Ceci est l'annexe {0}; auteur: {1}", NO_ANNEXE);
```

C++ privilégie l'enchaînement d'opérateurs binaires retournant, par convention, l'opérande de gauche pour arriver aux mêmes fins.

Pensez au comportement de `std::cout` pour en avoir un exemple. Ceci permet au compilateur de valider chaque paramètre sur une base individuelle, rapportant l'essentiel du travail à la compilation.

```
const short NO_ANNEXE = 18;
const string AUTEUR = "un chic prof";
cout << "Ceci est l'annexe " << NO_ANNEXE
     << "; auteur: " << AUTEUR;
     << endl;
```

Un lieu où C++ `03` avait besoin de pouvoir considérer des listes de valeurs comme telle est dans l'initialisation de certains objets.

Il est par exemple élégant, en C++, d'initialiser un tableau comme ceci.

```
char *tab[]{ "J'aime", "mon", "prof" };
for (auto s : tab)
   cout << s << ' ';
```

Pour en arriver au même résultat avec un vecteur, il fallait autrefois malheureusement procéder de manière beaucoup moins élégante.

```
vector<string> v;
v.push_back("J'aime");
v.push_back("mon");
v.push_back("prof");
for (const auto & s : v)
   cout << s << ' ';
```

Autre stratégie (gênante car elle demande de passer par un tableau pour initialiser un vecteur).

```
char *tab[] { "J'aime", "mon", "prof" };
vector<string> v{begin(tab), end(tab)};
for (const auto & s : v)
   cout << s << ' ';
```

C'est l'une des raisons pour lesquelles C++ `11` inclut une notation pour les séquences d'initialisation d'objets arbitraires, représentée par le type `std::initializer_list`, un type générique dont les éléments peuvent être parcourus par une séquence à demi ouverte $[begin..end[$ conforme au standard. Ainsi, la forme $\{e_0, e_1, \cdots, e_n\}$ peut y être utilisée pour tous les conteneurs, sans exception.

À titre d'exemple :

```
#include <initializer_list>
#include <vector>
// ...using...
class Tableau {
   vector<int> v;
public:
   Tableau(initializer_list<int> lst) : v{begin(lst), end(lst)} { // ou v{ lst }
   }
   // ...
};
int main() {
   Tableau tab { 1, 2, 3, 4, 5 }; // Tableau::Tableau(initializer_list<int>)
   // ...
}
```

Le type générique `initializer_list` permet aux conteneurs standards et maison d'obtenir le même niveau de support syntaxique que les agrégats du langage C tels que les `struct` et les tableaux bruts.

Il est aussi possible en C++ d'utiliser des *templates* variadiques, ou *templates* acceptant un nombre arbitrairement grand de paramètres. Ce faisant, une fonction peut être définie récursivement pour traiter le premier paramètre, puis les autres, ce qui donne à C++ une alternative robuste aux fonctions telles que `std::printf()`. Cette thématique est explorée dans des volumes ultérieurs de cette série[122].

[122] Voir [hdVarTmpl] si vous êtes trop curieux.

Objets constants

Les langages Java et .NET n'offrent pas de support conceptuel réel pour les objets constants. La raison fondamentale pour ce manque est simple : les objets sont toujours utilisés de manière indirecte (par référence) dans ces langages, ce qui signifie qu'apposer une mention de constance rendrait la *référence* constante, pas le *référé*[123].

Le concept associé au mot const en C++ ne trouve donc pas d'équivalent *pour les objets* en Java ou dans les langages .NET, et ce dans tous les cas. En Java, le problème se présente à peu près comme suit :

- un objet Java peut être créé avec new puis affecté à une référence spécifiée final, mais c'est alors la référence qui sera constante, pas l'objet;
- on pourra utiliser des mutateurs sur l'objet malgré tout, et on pourra utiliser une référence qui ne soit pas final vers un objet auquel pointe aussi une référence final.

```
public class Z
{
   private int valeur;
   public Z() {
      setValeur(0);
   }
   public void setValeur(int valeur) {
      this.valeur = valeur;
   }
   public static void main(String [] args) {
      final Z z0 = new Z();
      // Ok, même si z0 est «final»
      z0.setValeur(3);
      Z z1 = z0; // copie de référence. Ok
      z1.setValeur(6); // Ok
   }
}
```

[123] À vrai dire, il serait *facilement* possible de contourner ce problème sans créer de contorsion conceptuelle majeure, par exemple en utilisant un mot clé différent pour indiquer la constance du *référé* plutôt que celle de la *référence*.

Une situation semblable s'applique en C#, et pour les mêmes raisons. Notez que le mot `const` est utilisé ici, comme en C++.

C# a la gentillesse de nous informer que l'instruction où on tente d'initialiser un `const Z` (dans la méthode `Main()`) est tout simplement illégale[124].

Notez qu'un attribut d'instance ou de classe en C# (le langage les nomme des champs) peut être qualifié `readonly`, ce qui le rend non modifiable ailleurs qu'à sa déclaration ou que dans un constructeur, mais cela ne permet pas de créer des instances constantes autrement que par design de la classe toute entière.

Le problème se pose donc comme suit : comment en arriver à avoir des objets *opérationnellement constants* (donc dont les états sont fixés dès leur création) sans que le langage ne supporte le concept de constante opérationnelle.

```
class Z
{
   public int Valeur
   {
      get; private set;
   }
   public Z()
   {
      Valeur = 0;
   }
}
class Démarrage
{
   static void Main(string[] args)
   {
      const Z z0 = new Z(); // illégal!
   }
}
```

Comment garantir, idéalement à la compilation, le caractère non modifiable d'un objet donné suite à sa création?

La réponse à cette question a été le développement d'une technique : celle par laquelle on emploi des ***objets immuables***.

⇒ Un objet est dit **immuable** s'il n'offre aucune méthode qui permette d'en modifier le contenu une fois qu'il aura été créé[125].

Une image grossière d'un objet immuable serait celle d'un objet ayant des constructeurs, des accesseurs mais aucun mutateur, aucune propriété `set` (si disponible) et aucun opérateur d'affectation (pour les langages qui supportent ce concept).

[124] Et non, créer un `new const Z()` n'est pas permis.

[125] Il est intéressant de comparer la technique d'immutabilité avec l'idée derrière le mot clé `mutable` de C++, qui rend un attribut insensible au qualificatif `const`.

Évidemment, pour qu'un objet soit immuable, il faut s'assurer qu'aucune de ses méthodes ne permette d'en modifier quelque attribut que ce soit. Règle générale, on s'assurera surtout que les services publics et protégés d'une instance ne permettent pas d'en modifier les états, que ce soit de manière directe ou indirecte.

L'exemple à droite est une classe C++ nommée `Entier` représentant un entier de manière immuable (des constructeurs, un accesseur mais aucune méthode permettant de modifier la valeur d'un `Entier` une fois celui-ci créé).

Notez que j'ai qualifié `valeur()` avec `const` dans cet exemple, par hygiène, mais ceci serait impossible en Java ou en C#.

```
// classe immuable en C++
class Entier {
   int val {};
public:
   Entier() = default;
   Entier(int val) : val{val} {
   }
   int valeur() const {
      return val;
   }
};
```

Si j'avais omis la qualification `const` dans `valeur()`, nous n'aurions pas pu déclarer un `const Entier` puis en vérifier la valeur. Évidemment, les langages où on doit avoir recours à la conception d'objets immuables sont habituellement des langages où les objets constants n'ont pas de support du langage, alors ce n'y serait pas un enjeu.

Une addition entre deux instances immuables d'`Entier` se ferait comme proposé à droite. La caractéristique principale ici est qu'on doit, pour contenir le fruit de chaque nouveau calcul, créer un nouvel objet, car il est impossible de modifier les objets existants, ceux-ci étant immuables!

```
Entier e0 =3,
       e1 =2;
Entier somme = e0.valeur() + e1.valeur();
// ou Entier somme = e0 + e1; s'il est
// possible de surcharger l'opérateur +
// ou de définir operator int ()
```

Les langages Java et C# exploitent massivement les constructeurs et la collecte automatique d'ordures (car tous les objets y sont créés avec `new`) pour tous les objets immuables. Les cas d'objets immuables dans les langages comme Java et C# sont nombreux. Les plus connus sont sans doute les classes `String`[126] d'un côté comme de l'autre : il n'existe aucun moyen de modifier le contenu d'une instance de `String` une fois celle-ci instanciée.

En Java, les classes offrant une contrepartie `OO` des types primitifs (la classe `Integer` pour le type `int`, par exemple) sont aussi des classes dont les instances sont immuables.

Une caractéristique importante de la technique d'immuabilité est qu'on se retrouve avec une immuabilité *par classe*, alors que la spécification de constance habituelle exprime une constance *par instance*. En effet, toute instance de `String` en Java est immuable, alors qu'il est possible en C++ d'avoir certaines `std::string` qui soient `const` et d'autres qui ne le soient pas.

[126] La classe `java.lang.String` en Java et la classe `System.String` en C#. Bien qu'il s'agisse de deux classes distinctes, les deux partagent la particularité d'être conçues selon l'approche d'immutabilité décrite ici.

Cela signifie qu'en C# et en Java, quand on a besoin de vitesse et quand on sent que la construction spontanée et fréquente de nouvelles instances de `String` coûte trop cher en ressources (mémoire, temps de processeur), on a recours à une contrepartie modifiable de cette classe (`StringBuffer` en Java, `StringBuilder` en C#). Ces classes offrent des instances modifiables et offrent un constructeur paramétrique prenant une instance de `String` en paramètre (et `String` fait de même de son côté avec ses contreparties modifiables). Cela rend le modèle complexe, mais utilisable.

Constructeurs de classe et membres partagés

Dans les langages `.NET`, il est possible d'offrir des ***constructeurs de classe***, qui sont, syntaxiquement parlant, des constructeurs précédés du mot clé `static` (en C#) ou `Shared` (en `VB.NET`).

⇒ Un **constructeur de classe** est une méthode appelée au plus tard juste avant la construction d'une première instance de la classe en question.

L'exemple proposé à droite montre la syntaxe utilisée en C# pour les constructeurs de classe.

```
class X
{
   static X
   {
      // constructeur de classe
   }
   public X()
   {
     // constructeur d'instance
   }
}
```

En C++, ce concept n'existe pas. Les constructeurs sont des méthodes d'instance, purement et simplement. Lorsqu'on désire exécuter un sous-programme avant que ne soit faite la première instanciation d'une classe donnée, on y va d'une technique qui combine les idées de classe interne et de singleton.

La principale différence entre les deux approches est que le support de niveau langage des langages `.NET` pour les constructeurs de classe fait en sorte que, pour une classe donnée, le constructeur de classe soit appelé juste avant la première instanciation.

```
class X {
   struct Interne {
      // «constructeur de classe» pour X
      Interne();
   };
   // ... singleton sera construit
   // avant que le code n'y accède
   static Interne singleton;
   // ...
};
```

En C++, si on laisse la mécanique gérer le singleton tel que proposé ici, le singleton jouant un rôle semblable à celui du constructeur de classe sera construit avant même le début du programme principal.

En Java, le constructeur de classe d'une classe donnée est un bloc `static` anonyme déclaré dans la classe.

```
class X {
   static {
      // constructeur de classe
   }
   public X() {
     // constructeur d'instance
   }
}
```

Délégués

Dans les langages .NET et en Delphi[127], on mousse fort le concept de ***délégué***.

⇒ Un **délégué** (`delegate`) est une classe déguisée en sous-programme offrant le polymorphisme sur la base de sa signature.

En gros, un délégué respectant un certain prototype peut être utilisé de manière polymorphique pour remplacer n'importe quelle méthode ayant la même signature.

En Java, on privilégiera l'emploi d'*interfaces* pour arriver aux mêmes fins. Ceci implique de *nommer les concepts* (nommer l'interface racine du polymorphisme), mais gagne en généralité face aux délégués.

Ce qui permet à Java de rester souple malgré cette obligation de passer par des interfaces pour mettre en place des comportements polymorphiques est la capacité qu'offre ce langage de définir des classes anonymes.

Les pointeurs de fonctions globales de C++ ne sont pas un véritable remplacement pour les délégués, contrairement à ce qu'on pourrait penser.

Prenons à titre d'exemple le code à droite. Le type nommé `pf` représente un pointeur sur une fonction retournant un `int` et prenant en paramètre un `const int`.

La fonction `f()` est une telle fonction. La méthode d'instance `m()` de `X` *semble* aussi être une telle fonction, or (comme l'indique la compilation de `main()`) tel n'est pas le cas.

La raison pour cet état de fait est qu'une méthode d'instance reçoit toujours un paramètre supplémentaire et presque invisible au programmeur : le pointeur `this` sur l'instance active.

Ainsi, la signature de la méthode `m()` de `X` diffère de celle de la fonction `f()`.

```
using pf = int (*)(int);
int f(int n) {
   return n + 1;
}
struct X {
   int m(int n) const {
      return n * 2;
   }
};
int main() {
   // légal: f() respecte la
   // signature du type pf
   pf p0 = &f;
   // illégal, malgré les apparences
   X x;
   pf p1 = &(x.m);
}
```

Les méthodes d'un objet sont toutes, au fond, des fonctions globales déguisées. Chaque méthode de classe est (techniquement) une fonction globale munie de certains privilèges d'accès aux données de la classe à laquelle elle appartient; chaque méthode d'instance est aussi une fonction globale mais qui doit connaître (à travers `this`) l'instance à laquelle elle appartient. Le code d'une méthode d'instance est *techniquement* global, comme celui des fonctions, même s'il est *conceptuellement* local à l'objet.

[127] … cela découle en grande partie du fait que le concepteur de Delphi est aussi l'architecte de C#.

En C++, toutefois, il est possible par programmation générique d'exprimer des délégués arbitrairement flexibles (plus que ceux de C#) et très rapides. La bibliothèque *Boost* en offre d'ailleurs quelques exemples que vous pouvez examiner si vous en avez envie [BoostFct], et la classe `std::function` de C++ 11 le fait aussi très bien (c'est d'ailleurs sans doute le choix optimal si vous souhaitez des délégués dans ce langage; voir [POOv02] pour des détails). Vous pouvez même les écrire vous-mêmes si vous êtes en forme : c'est un exercice divertissant mais un peu croustillant de programmation générique (nous couvrirons la chose dans un document ultérieur).

Héritage multiple d'implémentation

Comment peut-on simuler l'héritage multiple d'implémentation dans les langages qui ont choisi ne pas le supporter?

Présumant que le langage auquel manque l'héritage multiple d'implémentation (comme Java ou C#) supporte à tout le moins l'héritage multiple d'interfaces, voici une recette pour le simuler manuellement au prix d'un effort raisonnable.

Nous utiliserons un cas simple pour illustrer le tout, mais sachez que vous devrez y aller un peu au cas par cas si vous rencontrez ce type de situation. La *refactorisation* qui suit sera exprimée en C++ (où elle n'est pas nécessaire, évidemment) mais peut être faite sans grande douleur en Java ou sans un langage `.NET`.

Présumons que l'on désire dériver la classe `D` des classes `B0` et `B1`. La classe `B0` implémentera la méthode `m0()`, alors que la classe `B1` implémentera la méthode `m1()`.

Dans notre cas, ces méthodes seront banales, mais imaginons que ce ne soit pas le cas et que réécrire ces méthodes dans la classe `B` ne soit pas une option raisonnable.

On veut donc que `D` ait les méthodes `m0()` et `m1()` sans devoir les implémenter lui-même en entier, et qu'un `D` soit (au sens du polymorphisme) à la fois un `B0` et un `B1`. On veut aussi pouvoir traiter un `D` comme un `B0` ou comme un `B1` au besoin, par polymorphisme.

```
struct B0 {
   int m0() const {
      return 3;
   }
};
struct B1 {
   int m1() const {
      return 4;
   }
};
struct D : B0, B1 {
};
```

Pour s'en sortir sans héritage multiple, il faut penser `B0` et `B1` comme des interfaces pures, donc sans implémentation de `m0()` ou de `m1()`, ce qui permet d'utiliser `B0` et `B1` comme des racines polymorphiques, tout comme si l'on aurait eu accès à l'héritage multiple à part entière.

On veut aussi que tous ceux qui dériveront de `B0` puissent avoir accès à l'implémentation de `m0()` sans avoir à la réécrire. Le même souhait s'applique à `B1` et `m1()`.

Un truc qui fonctionne bien est de considérer une implémentation (nommons-la `B0Impl`) des méthodes de `B0`, et de faire en sorte que `B0Impl` dérive de `B0`. Appliquons la même technique avec `B1`.

⇒ Les classes `B0Impl` et `B1Impl` seront les ***implémentations partielles*** de `B0` et de `B1`.

Ensuite, notre `D` sera un dérivé de `B0` et de `B1` *par héritage d'interfaces seulement*. En termes Java ou C#, `D` *implémentera* `B0` et `B1`, mais n'en dérivera pas.

Pour éviter d'implémenter lui-même `m0()` et `m1()`, `D` aura comme attributs (par composition) une instance de `B0Impl` et une instance de `B1Impl`.

Ainsi, il ne restera plus à `D` qu'à écrire des méthodes `m0()` et `m1()` bidon qui *délègueront* leur tâche de traitement aux attributs appropriés.

La clé, donc, est en quatre temps:

- interfaces (pour conserver le polymorphisme via un héritage d'interfaces);
- implémentations partielles (pour limiter la redondance dans les implémentations);
- composition (pour cacher les implémentations partielles dans l'objet); et
- délégation (pour que les appels de méthodes soient acheminés aux implémentations partielles les plus appropriées).

```
// interfaces
struct B0 { // membres publics
   virtual int m0() const = 0;
   virtual ~B0() = default;
};
struct B1 { // membres publics
   virtual int m1() const = 0;
   virtual ~B1() = default;
};
// implémentations partielles
struct B0Impl : B0 {
   int m0() const {
      return 3;
   }
};
struct B1Impl : B1 {
   int m1() const {
      return 4;
   }
};
// héritage d'interfaces
class D : public B0, public B1 {
   // composition
   B0Impl b0;
   B1Impl b1;
public:
   // délégation
   int m0() const {
      return b0.m0();
   }
   int m1() const {
      return b1.m1();
   }
};
```

Avec un bon compilateur, l'implémentation résultante sera plus complexe à entretenir que la version exprimée (de manière plus naturelle) par héritage multiple, mais ne devrait pas vraiment être plus lente à l'exécution puisque la délégation d'une méthode de `D` vers une méthode d'un de ses attributs devrait être facile à éliminer à la compilation.

Notez que Java 8, et bientôt C# 8, permettent d'insérer une implémentation par défaut des méthodes dans une interface. Les langages convergent…

Héritage privé

L'héritage privé au sens de C++ se simule raisonnablement bien dans d'autres langages en combinant composition et délégation, comme dans l'exemple de simulation d'héritage multiple ci-dessus.

La principale différence est qu'on ne veut pas, dans une situation d'héritage privé, que le polymorphisme soit possible sur l'enfant en passant par une référence ou par un pointeur au parent. Ainsi, on répétera la technique avec `B0`, `B0Impl`, `B1`, `B1Impl` et `D`, mais sans dériver `D` et `B0` et de `B1`, tout simplement.

Classes anonymes

Tel que mentionné dans [POOv02], on peut simuler des classes anonymes à l'aide d'interfaces et de classes locales à des fonctions si ces deux concepts sont applicables dans le langage de programmation choisi. En C++, nous avons vu quelques exemples de cette manœuvre (voir la section sur les *mixin*, en particulier). Les expressions λ de C++ `11` sont, en quelque sorte, des classes anonymes.

En Java, les classes anonymes sont supportées par le langage. En C#, la manière privilégiée de réaliser les opérations faites en Java par des classes anonymes serait d'utiliser des délégués.

Foncteurs

Un foncteur est à la fois un objet et une fonction [POOv02]. Les foncteurs, au sens puriste du terme, sont possibles si on peut profiter à la fois des avantages d'une fonction et ceux des objets, donc si on peut les utiliser en tant que fonctions *instanciables* capables de conserver des états.

Certains langages se prêtent très bien à la conception de foncteurs, comme C++, avec l'opérateur `()`, et `LISP` et ses dialectes qui peuvent simuler des objets dans un langage purement fonctionnel. Notez que les λ de `C++ 11` sont aussi des foncteurs.

Les langages comme Java et C# partagent un problème conceptuel pour ce qui est de la représentation de foncteurs: ces langages ne permettent pas de rédiger des fonctions, au sens de fonctions globales évidemment.

En C#, on peut obtenir une partie des avantages des foncteurs en utilisant un délégué respectant la syntaxe de la méthode à utiliser pour réaliser l'opération du foncteur. L'exemple à droite montre comment on peut créer une classe `Afficher`, exprimant comment afficher un entier sur un flux, et comment un délégué (`op`, de type `Op`) peut remplacer la méthode `Operation()` de la classe `Afficher`.

On remarquera que la stratégie est un peu manuelle, par contre, et qu'on a recours à deux classes (la classe `Afficher` et la classe `Op`).

```
using System;
using System.IO;
namespace z {
   class Afficher {
      private TextWriter flux;
      public Afficher(TextWriter flux) {
         this.flux = flux;
      }
      public void Operation(int val) {
         flux.WriteLine(val);
      }
   }
   delegate void Op(int val);
   class Démarrage {
      static void Main(string[] args) {
         var aff = new Afficher(Console.Out);
         Op op = new Op(aff.Operation);
         int [] tab = { 1, 2, 3, 4, 5 };
         foreach (int elem in tab)
            op(elem);
      }
   }
}
```

En Java, la seule réelle alternative est de représenter le concept d'objet opérable par une interface et d'utiliser cette interface pour opérer sur l'objet. Cette stratégie peut être utilisée en C++ et en C# aussi, évidemment. C'est une application simple et usuelle du polymorphisme, mais pas vraiment ce qu'on entend par foncteur.

Types internes et publics

Ni Java, ni les langages `.NET` ne permettent de définir des types qui sont des *alias* pour d'autres types, ce qui réduit fortement la capacité qu'ont ces langages d'exprimer des abstractions complètes en programmation générique.

Évidemment, tous les langages `OO` pragmatiques supportent les classes internes, ce qui permet de contourner cette limitation par l'expression de types internes qui encapsulent manuellement d'autres types. Il est donc possible, mais parfois pénible, d'en arriver dans ces langages au même résultat qu'avec une définition de type à la `typedef` ou à la `using`.

Amitié

L'amitié au sens du mot clé `friend` de C++ n'est pas possible à proprement dit en Java ou dans un langage `.NET`. Cependant, des palliatifs partiels existent de par des qualifications d'accès spécifiques à ces langages :

- dans une classe C#, un membre qualifié `internal` est directement accessible aux membres du même assemblage et un membre qualifié `protected internal` est directement accessible aux dérivés de la classe comme aux classes du même assemblage;
- dans une classe `VB.NET`, un membre qualifié `Friend` est directement accessible aux membres du même assemblage et un membre qualifié `Protected Friend` est directement accessible aux dérivés de la classe comme aux classes du même assemblage;
- dans une classe Java, un membre qui n'est pas explicitement qualifié est considéré *Package Protected*, ce qui signifie qu'il est directement accessible aux classes situées dans le même paquetage.

Remarquez que la brèche d'encapsulation provoquée par ces manœuvres est à la fois beaucoup plus forte et beaucoup plus stricte que celle exposée par le mot clé `friend` de C++ :

- en C++, une classe peut offrir un privilège d'accès spécial sur tous ses membres à un ami clairement ciblé; alors que
- en Java et dans les langages `.NET`, une classe peut offrir un privilège d'accès spécial à un groupe arbitrairement grand sur un membre précis.

Il serait possible d'en arriver à reproduire l'amitié de C++ dans les langages `.NET` et en Java en concevant de très petits assemblages et de très petits paquetages, faits dans chaque cas strictement de classes considérées comme des amis privilégiés.

Traits

Les traits, qui permettent entre autres de qualifier et de documents des types, même primitifs, à partir de considérations connues *a posteriori* et d'écrire des types et des algorithmes génériques sur la base de ces qualifications, ne peuvent véritablement être exprimés dans un langage où la programmation générique n'est possible que sur les classes.

Une partie des traits repose sur la capacité qu'ont les classes d'exposer des types internes et publics. Conséquemment, les limites imposées sur cette fonctionnalité dans un langage donné sont aussi des limites quant à la capacité qu'a ce langage d'exprimer des traits.

Des propositions ont été faites pour ajouter à Java une adaptation locale du concept de trait[128]. En `.NET`, des annotations écrites dans un métalangage (qu'on y nomme des attributs `.NET`) peuvent aussi jouer un rôle semblable[129].

---

[128] Voir par exemple les articles http://www.artima.com/weblogs/viewpost.jsp?thread=220916 et http://web.cecs.pdx.edu/~pq/papers/techReport-CSE-04-005.pdf

[129] Pour un didacticiel bref à leur sujet, voir http://www.codersource.net/csharp_tutorial_attributes.html

Collecte automatique d'ordures

La collecte automatique d'ordures, intrinsèque à Java et aux langages .NET, teinte la philosophie des programmes et des classes offertes par les infrastructures de ces langages. En C++, dû à la maxime à coût zéro (MC0), forcer les programmes à dépendre d'un moteur de collecte automatique d'ordures n'est pas souhaitable[130].

Le standard de C++ ne force pas les implémentations des divers compilateurs et des bibliothèques à offrir la possibilité de livrer un moteur de collecte automatique d'ordures. Le standard n'explicite pas non plus la possibilité qu'ont les compilateurs et les bibliothèques d'offrir un tel moteur, parce qu'il faudrait alors expliciter ce qui est entendu par un moteur de collecte automatique d'ordures; dans le cas contraire, il y aurait un réel risque d'obtenir du code C++ à la fois conforme au standard et non portable. Une API primitive de collecte d'ordures existe pour C++ 11 mais je ne sais pas quels compilateurs l'implémentent.

Définir les politiques propres à un moteur de collecte automatique d'ordures n'est pas chose simple : même ceux offerts par Java et par les langages .NET n'appliquent pas les mêmes stratégies et ne visent pas les mêmes objectifs. Par exemple, le moteur de .NET cherche à collecter rapidement les objets déclarés localement à une méthode de déplace, en mémoire, les objets qui tendent à vivre plus longtemps. En retour, Java collecte le moins souvent possible (et pas du tout, s'il y arrive). Chaque stratégie a ses avantages, ses inconvénients et ses *aficionados*.

En C++, il existe des dialectes non-conformes au standard (pensez à C++/CLI, le dialecte de C++ sur plateforme .NET) qui implémentent un moteur de collecte automatique d'ordures pour C++, mais c'est une avenue déplorable : historiquement, le standard de C++ a cherché activement à éviter l'apparition de dialectes, puisque ceux-ci fragmentent et endommagent sa communauté de développeurs.

Il existe aussi diverses implémentations de moteurs de collecte automatique d'ordures implémentées à même le langage. Par exemple :

- l'article http://www.codeproject.com/cpp/garbage_collect.asp propose une infrastructure de collecte automatique d'ordures reposant sur les mécanismes plus sophistiqués de contrôle de la gestion de la mémoire allouée dynamiquement vus dans la section ***Gestion avancée de la mémoire***, plus haut;
- pour la suite, voir http://www.codeproject.com/cpp/garbage_collect2.asp;
- l'article http://www.codeproject.com/cpp/libgc.asp présente une autre solution au même problème;
- l'article http://www.hpl.hp.com/personal/Hans_Boehm/gc/ propose un moteur de collecte automatique d'ordures utilisé dans plusieurs projets. L'article est de ***Hans Boehm***, un éminent spécialiste de la multiprogrammation.

Vous trouverez beaucoup plus d'informations à ce sujet sur [hdGC].

[130] L'article http://www.open-std.org/jtc1/sc22/wg21/docs/papers/1996/N0932.pdf qui fut publié en 1996 démontre toutefois qu'il est tout à fait possible de mettre en place un tel mécanisme en C++.

Enfin, les pointeurs intelligents permettent d'implémenter des politiques efficaces de gestion de la mémoire allouée dynamiquement. Les types `unique_ptr` et `shared_ptr`, de la bibliothèque standard (en-tête `<memory>`), en sont deux exemples très connus. Écrire un tel pointeur est très possible (nous en verrons quelques-uns dans des documents à venir) mais il s'agit de code subtil, de *code de grande personne*. En général, un pointeur intelligent est facile à utiliser mais demande beaucoup de prudence au moment de la conception.

Notez que la propagande selon laquelle la collecte automatique d'ordures est un avantage pour un langage donné est, effectivement, de la propagande.

Cette position reflète une vision (pas *la* vision) de ce qu'est la programmation, et entraîne son propre lot de difficultés :

- destructeurs s'exécutant à des moments semi-imprévisibles;
- libération manuelle des ressources externes;
- schémas de conception implémentés manuellement pour reproduire certains automatismes de C++ (interfaces `IDisposable` des langages `.NET`);
- complexification du langage par des blocs `finally`, des ajouts spéciaux pour injecter un peu de déterminisme dans les langages (blocs `using` de C#, blocs *try-with* de Java) et tours de magie du compilateur, incluant la libération automatique de ressources connues *a priori*, comme les mutex de `.NET` en C#; *etc*.

Un langage dépendant fortement de l'allocation dynamique de mémoire bénéficiera nettement d'une collecte automatique d'ordures, mais il s'agit alors d'une solution locale à un problème local. Si d'autres options de programmation sont possibles, alors la collecte automatique d'ordures est intéressante quand le problème s'y prête.

N'oubliez pas que collecter la mémoire ne signifie pas nécessairement finaliser les objets. Les deux problèmes sont reliés mais distincts l'un de l'autre.

Programmation orientée aspect (POA)[131]

Là où la POO offre un paradigme pour regrouper sous un même nom des données (*attributs*) et des opérations sur ces données (*méthodes*), la ***programmation orientée aspect*** (**POA**) vise à regrouper en modules des préoccupations transverses communes à plusieurs classes.

Certains diront qu'il s'agit d'un nouveau paradigme, alors que d'autres y verront une technique de programmation s'appuyant sur la POO. Il est possible que le langage de prédilection des gens influence leur perception de la POA comme novatrice ou complémentaire.

Le rôle que joue l'***aspect*** en POA est analogue à celui de la classe en POO. Ainsi, un **aspect** permet de définir :

- un **point d'action** (en anglais : *Pointcut*), qui est l'endroit où se produira l'interaction entre objets distincts; et
- une **condition d'exécution** (en anglais : *Advice*), qui définit quant à lui le moment où se produira l'interaction.

Souvent, les tenants de la POA comme approche novatrice mettront de l'avant qu'une condition d'exécution pourrait être *avant l'appel d'une méthode* ou *après l'appel d'une méthode* et permettait d'exprimer des *préconditions* ou des *postconditions* à l'exécution de ces méthodes.

Des classes spéciales, nommées **classes de tramage**, permettent d'exprimer des relations entre deux classes distinctes avec un couplage minimal. La classe de tramage exprime ce qu'on nomme l'**aspect** de la relation entre deux classes distinctes, et ne nécessite idéalement pas de connaissance du couplage de la part des objets qu'elle associe. L'aspect est une relation apposée entre deux classes sans que celles-ci n'en soient informées.

La plupart des langages utilisés en POA aujourd'hui sont en fait des métalangages (des espèces de préprocesseurs évolués) au-dessus d'un langage OO commercial (souvent Java, mais il existe des versions C# et C++), ce qui est raisonnable car l'approche OA se veut un complément à ce qui est perçu par ses partisans comme étant des *carences* du modèle OO.

Le couplage minimal résultant de l'apposition d'une classe de tramage pour exprimer un aspect de la relation entre deux classes se veut porteur de plusieurs vertus :

- facilité accrue à maintenir la relation entre l'aspect et les classes;
- facilité de réutilisation des classes liées par une classe de tramage, du fait que les classes liées par un aspect ne savent rien de l'environnement relationnel ainsi exprimé; et
- facilité d'exprimer de nombreux aspects sans couplage entre eux, ce qui implique une possibilité de répartition efficace du travail.

Notez qu'à une certaine époque, les *préconditions* et les *postconditions* lors d'appels de méthodes faisaient partie intégrante du langage C++, mais sont disparues.

Ces méthodes spéciales n'auraient de toute manière pas permis l'application de la POA, n'exprimant pas des concepts transverses entre plusieurs classes.

On utilise souvent le mot ***greffe*** pour exprimer l'insertion d'un aspect dans une relation entre deux objets.

[131] Pour en savoir plus sur la POA, voir l'essai de maîtrise de **Lamine Sy** déposé à l'université de Sherbrooke en 2005 et titré *Mise en place de services Web avec la programmation orientée aspect*.

Applications de la POA

La POA est intéressante lorsque des considérations apparaissent dans un programme OO de manière telle qu'elles impliquent un ensemble de classes plutôt qu'une classe en particulier.

Les présentations typiques de la POA parleront entre autres de l'écriture dans un journal (un *Log*) de certains états lors d'appels de méthodes. Il est possible qu'on veuille que certaines classes d'un programme, en période de test, écrivent les valeurs de certaines variables dans un fichier ou dans une BD avant ou après chaque appel de méthode, souvent pour faciliter la tâche de dépistage de certains *bugs*.

Typiquement, en C et en C++, on aura recours à des directives de compilation conditionnelle (à des blocs `#ifdef ... #endif`) pour circonscrire des blocs d'opérations insérés de manière calculée au début ou à la fin des méthodes visées dans les classes identifiées comme étant à risque. Ces blocs seront inclus ou non selon la valeur ou même selon la définition de certains symboles au niveau du préprocesseur[132].

Cette technique a fait ses preuves, mais elle a beaucoup de défauts :

- elle est fortement artisanale, ce qui implique qu'elle est particulièrement sujette à souffrir d'erreurs d'inattention;
- elle nécessite des changements dans le code de plusieurs classes et de plusieurs méthodes, de par l'insertion stratégique des opérations d'écriture dans un fichier, dénaturant ainsi le code des méthodes de manière parfois difficilement repérable et nettoyable ultérieurement;
- elle repose sur la présence d'un préprocesseur, donc d'un outil qui transforme le code source avant la compilation. On peut souvent procéder de la même manière en examinant la valeur d'un fanion au moment de l'exécution, mais cela risque de générer du code plus lent (et plus gros), selon la qualité des modules d'optimisation des compilateurs utilisés.

L'emploi de classes de tramage exprimant une relation entre plusieurs classes peut permettre d'exprimer une relation telle qu'*avant l'appel de toute méthode des classes ainsi tramées, pose telle action*.

On peut imaginer beaucoup d'autres applications *méta* de la POA :

- prendre des mesures autour des appels de méthodes pour tirer des statistiques d'utilisation et de performance (ce qu'on nomme le **profilage**);
- automatiser certaines validations répétitives de paramètres;
- pour des fins d'internationalisation des programmes, traduire certaines chaînes de caractères passées en paramètre à des méthodes pour les transformer automatiquement en chaînes respectant les standards linguistiques locaux; *etc*.

[132] Le symbole DEBUG, souvent, sera défini en période de tests et ne le sera pas quand on produira la version officielle du programme

Les tisseurs

Tel qu'indiqué plus haut, les métalangages utilisés pour appliquer des aspects par-dessus des programmes OO sont souvent des préprocesseurs évolués, parfois nommés tisseurs car ils tissent des relations entre des classes a priori disjointes.

Pour être utile, un tisseur doit permettre d'exprimer un vaste éventail de conditions d'action. Pour être efficace, il doit être intimement lié au langage OO sous-jacent pour insérer les points d'action aux bons endroits et pour permettre d'exprimer les lieux propices pour ces insertions.

Certains moteurs transactionnels client/ serveur (COM+ de *Microsoft* et EJB de *Sun* en sont de bons exemples) permettent d'exprimer certaines relations de type aspect en interceptant des appels entre objets et en insérant discrètement des opérations entre l'émission d'un message par un composant et la réception de ce message par son destinataire.

Les deux modèles cités en exemple ci-dessus ont le défaut de ne pouvoir exprimer que des aspects tirés d'une liste construite *a priori*, et à laquelle on ne peut ajouter de nouveaux aspects.

Quelques bémols

Il faut tout de même exprimer, au passage, quelques bémols face à l'introduction d'aspects comme complément à la POO.

En effet, à moins que le moteur supportant les aspects ne soit *très* prudent, la mise en place d'un aspect devant être appliqué juste avant ou juste après toutes les méthodes d'une certaine signature permettra à un individu malfaisant (ou simplement curieux) de suivre à la trace l'invocation de certaines méthodes privées. En particulier, dans un langage comme C++ où, contrairement à Java ou aux langages .NET, il est possible d'exposer des interfaces véritablement opaques, ceci pourrait mener à la révélation non désirée de détails d'implémentation.

Notons aussi que l'une des applications de la POA, du moins pour les moteurs qui supportent pleinement la réflexivité, est ce qu'on nomme l'**introduction** ou l'**intercession**. Il existe des moteurs OO dans lesquels la réflexivité permet même d'insérer des attributs et des méthodes dans les classes ainsi représentées.

Certains produits (comme JBoss pour Java, un produit à code ouvert) permettent l'introduction au sens d'ajouter une interface à une classe et d'implémenter les méthodes de cette interface, pour permettre à un aspect d'ajouter une fonctionnalité à une classe (par exemple celle d'être *sérialisable*).

L'introduction permet à un aspect de modifier la structure de l'objet sur lequel il opère, en y ajoutant des membres. On le voit tout de suite, il s'agit d'un concept puissant… et d'un très sérieux bris d'encapsulation structurelle. La question se pose alors : le jeu en vaut-il la chandelle? Peut-être bien que oui, convenons-en, mais la question en est pour le moment au stade de l'étude et de la recherche.

Internationalisation

Avec l'ubiquité du réseau Internet, le village global est véritablement devenu, pour les *technoriches* du moins, une réalité.

La plupart des outils de programmation récents, en particulier Java et C#, tiennent compte à même leur design de plusieurs considérations propres à l'internationalisation, en particulier dans leur gestion des chaînes de caractère, ayant tous deux adopté le standard Unicode comme système d'encodage par défaut du texte. Depuis la version `1.6` de Java, en particulier, ce langage fait des efforts marqués en direction de l'internationalisation des unités de mesure [JavaInM].

De son côté, C++ reste à bien des égards un dérivé de C, et son support le plus naturel reste, aux yeux de la majorité, offert au texte encodé sur huit bits ou moins comme le sont les standards `ASCII` et `ANSI`. Ce problème en est principalement un de perception, il faut le dire, puisqu'il existe à toutes fins pratiques, avec la bibliothèque standard de C++, le même support pour les séquences de caractères `8` et `16` bits :

- les algorithmes standards (`copy()`,`for_each()`,`generate()`, *etc.*), opèrent tous sur des séquences quelconques;
- les entrées/ sorties à la console sont supportées aussi naturellement sur des `wchar_t` (`wcin`, `wcout`, `wclog` et `wcerr`) que sur des `char` (`cin`, `cout`, `clog` et `cerr`); et bien sûr
- il existe des classes de support fonctionnellement identiques pour les chaînes de caractères applicables à des `char` (`string`) et celles applicables à des `wchar_t` (`wstring`).

En fait, le plus grand obstacle à l'internationalisation avec C++ est l'habitude que la plupart des gens ont d'utiliser des `char` (et les opérations et objets qui leur sont apparentés) plutôt que des `wchar_t`. Une impulsion historique bien plus que technique.

Notez que C++ `11` va bien plus loin en offrant un support uniforme des littéraux caractères Unicode, mais je n'ai pas, au moment d'écrire ces lignes, accès à un compilateur permettant de tester ces nouveaux outils.

Cette impulsion qui nous maintient dans nos habitudes d'écrire du code simple mais qui ne fonctionne que dans un environnement *américanocentré* nous fait négliger un aspect très puissant de la bibliothèque standard de C++ qui est son support très complet de l'internationalisation, allant bien au-delà de considérations alphabétiques.

Cette section se veut une introduction à l'internationalisation en C++, incluant le puissant concept client/ serveur (`CS`) de ***facettes***, et donne une démonstration non seulement de l'applicabilité du modèle standard, mais aussi de la facilité par laquelle il est possible d'étendre les facettes existantes du modèle d'internationalisation standard.

Représenter les règles locales d'encodage

La première chose à relever quand on adresse la question de l'internationalisation est qu'il s'agit, fondamentalement, d'un problème ouvert. Ce qui constitue la liste de considérations propres à un endroit donné dépend de la culture et des préoccupations propres à ce lieu.

Certains aspects sont reconnus comme plus globaux que d'autres: représentation de la monnaie, des nombres, des dates, *etc*. D'autres sont moins répandus mais peuvent devoir être pris en considération dans certaines applications ou certains pays. Pensez par exemple :

- à la manière d'identifier la taille des souliers, qui diffère en Europe et en Amérique du Nord;
- aux systèmes de mesure (métrique, impériale, autre);
- aux structures de pouvoir (république, fédération, royaume, *etc*.); et ainsi de suite.

Dans un souci d'uniformité conceptuelle, la bibliothèque standard de C++ a été réfléchie de manière à supporter, de façon homogène, les *facettes* jugées d'utilité globale et les facettes qu'on qualifierait de plus spécifiques.

⇒ La représentation d'un groupe d'aspects propres à un lieu ou à une culture se fait dans une instance de **`std::locale`**. Pour y avoir accès, il faut inclure **`<locale>`**.

On peut construire une instance de `std::locale` pour un lieu spécifique en passant une chaîne descriptive dudit lieu en paramètre à son constructeur[133]. Le constructeur par défaut d'un `std::locale` crée, sans surprise, une représentation jugée correcte pour la plateforme; avec *Visual Studio*, la présomption est (sans surprise) que nous sommes américains.

[133] La liste des noms de lieux supportés sur un ordinateur donné est obtenue par la commande `locale -a` sur les systèmes conformes à la norme `POSIX`, et par la base de registres sous *Microsoft Windows*. On peut obtenir le nom descriptif d'un lieu donné en appelant la méthode `name()` du `std::locale` le représentant.

Imprégner un flux d'un lieu

Les entrées/ sorties sont guidées par le locale courant, et on peut imposer à un programme de respecter les règles propres à un lieu et à une culture donnée. Par exemple, pour appliquer les règles de formatage propres à l'Allemagne dans une sortie à la console sous *Microsoft Windows* :

```
std::locale loc{"German_Germany"};
std::cout.imbue(loc);
// jusqu'à ce qu'on impose un autre lieu, les règles locales de
// l'Allemagne seront appliquées
```

La méthode `imbue()` d'un flux en sortie imprègne, le mot le dit, ce flux d'un lieu. Elle retourne le lieu dont était précédemment imprégné le flux, ce qui permet facilement d'imposer des règles précises pour quelques opérations d'entrée/ sortie puis de revenir au lieu précédent :

```
#include <locale>
#include <iostream>
int main() {
   using namespace std;
   auto &loc = cout.imbue(locale{"french_france"});
   const double PI = 3.14159;
   cout << PI << endl; // affiche 3,14159
   cout.imbue(loc);
   cout << PI << endl; // affiche 3.14159
}
```

Introduction aux facettes

Le formatage des nombres est une facette particulière des règles d'internationalisation possibles, mais ce n'est évidemment pas la seule.

Les facettes représentent une technique de programmation qui transcende les considérations d'internationalisation. Nous y reviendrons.

⇒ Toute facette est une instance d'une classe dérivée de la classe **`std::locale::facet`**, donc d'une classe interne à la classe `std::locale`.

⇒ Chaque facette représente une particularité locale possible (pensez par exemple à la représentation d'une date).

⇒ Chaque instance de `std::locale` peut être vue comme un ***serveur de facettes***, donc comme un objet ayant entre autre pour rôle de fournir des facettes sur demande.

Une facette possible pourrait être la suivante, nommée `FacettePolitique`, dont le rôle serait de représenter les divers aspects du régime politique dans un lieu donné. Il s'agit d'une classe manifestement trop simple pour représenter cet aspect crucial de la vie humaine qu'est la vie politique, mais il ne s'agit que d'une illustration.

Remarquons tout d'abord que, tel qu'indiqué, une facette conforme doit dériver de la classe `locale::facet`.

Ceci permet à `locale` de gérer les facettes par polymorphisme ou à l'aide de *templates*.

Dans cet exemple, j'ai choisi de représenter les différentes catégories par lesquelles `FacettePolitique`, une facette particulière, décrira un lieu à l'aide de constantes énumérées.

Il s'agit là d'un choix purement arbitraire. Des constantes entières, des chaînes de caractères, des abstractions dynamiques comme des pointeurs sur des objets auraient tous pu servir à de telles représentations.

J'ai aussi choisi de représenter les aspects politiques d'un lieu dans cette facette à l'aide de constantes d'instance. Il s'agit, encore une fois, d'un choix personnel, pas d'une règle du modèle.

```
#ifndef FACETTE_POLITIQUE_H
#define FACETTE_POLITIQUE_H
#include <locale>
#include <string>
// ...using ...
class FacettePolitique : public locale::facet {
public:
   FacettePolitique(const FacettePolitique&) = delete;
   FacettePolitique&
     operator=(const Facette Politique&) = delete;
   enum RepartitionPouvoirs {
      Parlementaire, Presidentiel, SemiPresidentiel,
      Dictatorial, Monarchique, Theocratique,
      Totalitaire, Autoritaire
   };
   enum OrganisationTerritoire {
      Federation, Confederation, EtatUnitaire
   };
   enum Autorite {
      Aucun, Dictateur, PremierMinistre,
      President, Tyran
   };
private:
   const string chef;
   const Autorite autorite;
   const OrganisationTerritoire org_terr;
   const RepartitionPouvoirs rep_pouv;
```

Toute facette ***doit*** avoir un attribut de classe public de type **`std::locale::id`** et nommé **`id`**. Cet attribut est utilisé par `std::locale` dans son rôle de serveur de facettes pour identifier les facettes par des index distincts les uns des autres (c'est une belle manœuvre `OO`).

```
public:
   static locale::id id;
```

Une facette n'a habituellement pas de constructeur par défaut.

On s'attend à ce que chaque facette soit initialisée dès sa construction avec les informations descriptives qu'elle est supposée contenir.

```
FacettePolitique(
   const string &chefEtat,
   RepartitionPouvoirs repPouv,
   OrganisationTerritoire orgTerr,
   Autorite autorite
) : chef{chefEtat}, rep_pouv{repPouv},
    org_terr{orgTerr}, autorite{autorite} {
}
```

Une bonne facette offrira habituellement une série d'accesseurs descriptifs de ce que représente la facette. On s'attend d'une facette à ce que ses accesseurs soient constants.

Il devrait être interdit de copier une facette. On y arrive normalement avec la même stratégie que celle appliquée pour empêcher la copie des singletons, par exemple (ici, nous avons dérivé la facette de manière privée de la classe `Incopiable`, vue dans des documents antérieurs de cette série).

```
   string chef_etat() const {
      return chef;
   }
   auto repartition_pouvoirs() const {
      return rep_pouv;
   }
   auto organisation_territoire() const {
      return org_terr;
   }
   Autorite autorite() const {
      return autorite;
   }
};
```

Enfin, sans que ce ne soit nécessaire, il peut être utile d'offrir une manière simplifiée d'obtenir un `std::locale` qui inclut notre (ou nos) facette(s).

```
locale& obtenir_locale();
#endif
```

Personnellement, j'aime bien représenter l'application d'une facette sur un cas particulier du monde, par exemple le système politique canadien, par un singleton.

Je vous propose une telle implémentation, annotée comme le fut la facette en tant que telle.

Il faut évidemment s'assurer d'avoir accès à `std::locale` et à la déclaration de la facette qu'on cherche à représenter.

```
#ifndef SINGLETONCANADIEN_H
#define SINGLETONCANADIEN_H
#include "Incopiable.h"
#include "FacettePolitique.h"
#include <locale>
// ...using...
```

Le singleton sera… un singleton, et en aura la signature et les caractéristiques.

Ce singleton aura un attribut d'instance, soit le `std::locale` représentant la localité visée et augmentée de la facette qu'on veut lui ajouter (remarquez la construction de l'attribut `ici_`, dont le deuxième paramètre est un pointeur de `std::locale::facet`).

```
class SystemeCanadien {
public:
   SystemeCanadien(const SystemeCanadien&) = delete;
   SystemeCanadien&
      operator=(const SystemeCanadien&) = delete;
   locale ici;
   SystemeCanadien()
      : ici{
           locale{""},
           new FacettePolitique{
              "Justin Trudeau",
              FacettePolitique::Parlementaire,
              FacettePolitique::Federation,
              FacettePolitique::PremierMinistre
           }
        }
   {
   }
public:
   locale& obtenir_locale() noexcept {
      return ici;
   }
   static &get() noexcept {
      static SystemeCanadien singleton;
      return singleton;
   }
};
#endif
```

On dira que cela *installe* notre facette dans un `std::locale`. Le reste de cette classe est plutôt banale.

Le fichier source du singleton doit définir le singleton, tout simplement.

Le fichier source de `FacettePolitique`, quant à lui, devrait définir l'attribut de classe `FacettePolitique::id`, ceci étant requis pour toute facette.

Ne reste qu'à y définir la fonction `obtenir_locale()` pour qu'elle retourne une référence sur le `std::locale` à utiliser. Ici, j'ai évidemment utilisé celle décrite par `SystemeCanadien`.

```
#include "FacettePolitique.h"
#include "SystemeCanadien.h"
locale::id FacettePolitique::id;
locale& obtenir_locale() noexcept {
   return SystemeCanadien::get().obtenir_locale();
}
```

Exploiter une facette

Comment se comporterait un programme désireux d'obtenir une facette culturelle d'un lieu donné, par exemple du lieu courant?

Un exemple correct serait celui ci-dessous :

```
#include "FacettePolitique.h"
#include <iostream>
#include <locale>
int main() {
   // ...using...
   locale &ici{obtenir_locale()};
   cout.imbue(ici);
   cout << "Chef d'état : " << use_facet<FacettePolitique>(ici).chef_etat() << endl;
}
```

La fonction `obtenir_locale()` est celle que nous avons définie plus haut. Tel que vu précédemment, on imprègne tout d'abord le flux de sortie visé du lieu dont il doit s'inspirer, à l'aide de sa méthode `imbue()`. Dans ce cas bien précis, que nous l'ayons imprégné ne changera pas grand-chose à son comportement.

On voit que pour exploiter une facette d'un lieu, il suffit d'appliquer l'opération **`std::use_facet`** à un `std::locale` donné, ce qui donne accès à une référence à la facette en question et permet d'en appeler les méthodes.

La beauté de `std::use_facet` est que, étant un *template*, son bon usage est vérifié de manière statique, à la compilation plutôt qu'à l'exécution, et donne donc du code sécuritaire et aussi rapide que possible à l'exécution.

Exemple plus complexe : classe `Date` respectant les standards locaux

Depuis le moment où j'ai écrit cette section, un fichier d'en-tête standard `<chrono>` s'est ajouté à C++ 11, alors au besoin, adaptez les noms choisi pour éviter les conflits. Notez aussi que `<chrono>` devrait comprendre une classe std::date à partir de C++ 20, grâce aux travaux de **Howard Hinnant**, alors ce qui suit ne se veut qu'illustratif.

Présumons que nous désirions écrire le programme ci-dessous et souhaitons que celui-ci fonctionne correctement peu importe les standards locaux d'affichage des dates. La classe `Date` qui y est utilisée est insérée dans un espace nommé (portant le nom `chrono_`) pour éviter des conflits avec d'autres classes `Date`, ce nom étant assez commun[134].

```
#include "Date.h"
using chrono_::Date;
#include <iostream>
#include <locale>
int main() {
   // ...using ...
   cout.imbue(locale{""}); // par principe...
   cout << Date(1972, Date::dec, 27) << endl;
}
```

Notez que la date utilisée dans l'exemple est purement arbitraire, mais que, dû aux choix de représentation internes à la classe, que nous examinerons sous peu, elle doit se situer inclusivement entre minuit le 1er janvier 1970 et 19:14:07 le 18 janvier 2038.

La déclaration de la classe `chrono_::Date` irait comme suit (protections contre les exclusions multiples omises pour fins d'économie).

Le type `time_t` est un type du langage C permettant de représenter et de manipuler une date. Nous réutiliserons ce qui existe et fonctionne déjà bien.

```
#include <iosfwd>
#include <ctime>
// ...using...
namespace chrono_ {
   class Date {
      time_t brute;
   public:
      class Invalide { };
      enum Mois {
         jan, fev,  mar, avr, mai, jun,
         jul, aout, sep, oct, nov, dec
      };
      Date(int annee, Mois mois, int jour);
      void TempsC(tm*) const;
   };
   friend ostream &operator<<(ostream&, const Date&);
}
```

La méthode `TempsC()` a pour rôle d'exposer la date représentée par une instance de `chrono::Date` dans le format traditionnel du langage C pour faciliter l'interopérabilité entre cette classe et le code existant.

[134] Je me suis fortement inspiré de http://www.cantrip.org/locale.html par **Nathan Myers** pour rédiger cet exemple.

On pourrait imaginer plusieurs autres méthodes pertinentes, comme des opérateurs pour ajouter ou soustraire un jour, un mois, un opérateur pour saisir une date d'un flux et ainsi de suite. Nous limiterons toutefois notre exposé à ceci dans le but de garder le tout simple et de rester axés sur les considérations d'internationalisation.

Le constructeur est relativement simple pour qui connaît un peu l'API standard du langage C pour manipuler des dates.

Une instance de `std::tm` est créée, puis tous ses champs sont initialisés à zéro. Certains champs sont ensuite remplis à la pièce, pour représenter la date décrite par les paramètres du constructeur.

```
#include "date.h"
#include <ctime>
#include <ostream>
#include <locale>
// ...using...
namespace chrono_ {
   Date::Date(int annee, Mois mois, int jour) {
      tm moment { };
      moment.tm_mday = jour;
      moment.tm_mon = mois;
      // respecter la norme C
      moment.tm_year = annee - 1900;
      if ((brute = mktime(&moment))==-1)
         throw DateInvalide{};
   }
```

Ensuite, `std::mktime()` est invoquée pour obtenir un `time_t`, représentation compacte de cette date.

La méthode `TempsC()` est banale, encapsulant un appel à une fonction standard C générant en quelque sort l'effet inverse du constructeur ci-dessus.

```
   void Date::TempsC(tm* tempsC) const {
      *tempsC = * gmtime(&brute_);
   }
```

Le gros morceau pour l'internationalisation de la classe `chrono::Date` est lié à son affichage. C'est pourquoi, on le comprendra, l'opérateur `<<` appliqué à un flux en sortie et à une `chrono::Date` sera le point le plus subtil de notre classe, et ce bien qu'il s'agisse au fond d'une fonction globale et non pas d'une méthode.

Tout d'abord, la fonction déclare une sentinelle. Il s'agit d'une technique `OO` ayant pour objectif d'éviter des problèmes de synchronisation lors d'accès à un flux d'entrée/ sortie[135].

Ensuite, on extrait la date en format C standard pour exploiter les outils de formatage de date existants.

Puis, on obtient une facette standard de type `std::time_put`[136], capable de formater une date sur un flux.

Enfin, on indique à cette facette où (et comment) écrire la date à exprimer, et on retourne le flux modifié par cette écriture.

```
ostream& operator<<
   (ostream &os, const Date &date) {
   ostream::sentry cerbere{os};
   if (!cerbere) return os;

   tm tmbuf;
   date.TempsC(&tmbuf);

const time_put<char> &timeFacet =
   use_facet<time_put<char>>(os.getloc());

   if (timeFacet.put(os, os, os.fill(), &tmbuf, 'x').failed())
      os.setstate(os.badbit);
   return os;
}
} // fin du de l'espace nommé chrono
```

La méthode `put()` d'une facette `std::time_put` prend cinq paramètres, mais est moins complexe qu'il n'y paraît :

- le premier paramètre est le flux où sera écrite la date;
- le deuxième est inutilisé (placé là pour usages futurs);
- le troisième indique la sorte de caractère à utiliser pour fins d'espacement lors de l'écriture (ici, on utilise le même caractère que le flux en sortie utiliserait normalement);
- le quatrième est la date en format C standard; et
- le cinquième est un code de formatage, qui correspond à ceux de la fonction `strftime()` du langage C.

Ce qu'il faut retenir est que dans ce cas, la facette `std::time_put` est responsable de décrire de manière standardisée les règles d'écriture d'une date pour un lieu et une culture donnés. Les méthodes de `std::time_put` importent peu, en fait, si on comprend le principe de la facette, puisque chaque facette est (potentiellement) unique en forme et en fonction.

[135] Des problèmes de ce genre surviendront fréquemment dans votre cours de systèmes client/ serveur. C'est une bonne technique mais pas essentiel à notre propos.

[136] …applicable à un `char`, mais elle aurait aussi pu être applicable à un `wchar_t` par exemple.

Dans d'autres langages

En Java, la classe dédiée à l'internationalisation est `java.util.Locale`, qui expose un ensemble de services pertinents mais n'a pas la prétention d'être aussi extensible que le système de facettes de C++ (Java ne supporte pas les techniques de programmation permettant de définir des facettes).

Dans le monde `.NET`, l'espace nommé regroupant les services d'internationalisation se nomme quant à lui `System.Globalization` (sous `.NET`, on parlera de globalisation plutôt que d'internationalisation; deux termes, un seul concept). On y retrouve entre autres une classe nommée `CultureInfo`.

Pour étendre l'ensemble des paramètres régionaux supportés par `.NET`, il faut passer parce que ce *Microsoft* nomme le *Microsoft Locale Builder*. Au moment d'écrire ces lignes, les exemples ne sont pas encore disponibles sur le site officiel de l'entreprise.

Appendice OO – Tester et mesurer dans un contexte objet

Aujourd'hui, on ne peut plus raisonnablement développer et livrer des systèmes informatiques sans les tester et sans être en mesure d'établir, pour ces systèmes, des métriques de qualité (rapidité d'exécution, complexité algorithmique, consommation de ressources, robustesse, *etc.*).

Est-ce que mesurer un programme OO est fondamentalement différent de mesurer un programme qui n'est pas OO? Comment mesurer si une approche OO est bien utilisée? Qu'est-ce qui distingue une stratégie OO efficace d'une qui l'est moins? À partir de quelles optiques peut-on poser un jugement sur la qualité d'un design OO? De manière générale, comment mesurer la qualité d'un design ou d'une implémentation OO? Comment comparer entre elles deux solutions OO distinctes de manière à faire des choix défendables et basés sur des critères objectifs?

Avertissement

Ce qui suit présente le sujet de la mesure de la qualité de systèmes objet, mais n'a pas la prétention d'être exhaustif ou détaillé. Le sujet est important (essentiel!) et est en pleine ébullition, plusieurs volumes y étant dévolus en totalité.

Pour quelques références électroniques, si le sujet vous intéresse, voir [hdMOO]

Peut-on appliquer au fruit de l'approche OO des critères de performance strictement traditionnels, comme mesurer le temps requis pour qu'un programme utilisant un certain objet réalise une tâche donnée, ou doit-on tenir compte des autres apports du modèle OO sur nos vies, comme la facilité avec laquelle on étend le système pour construire de nouveaux objets ou la simplicité de l'entretien des applications qu'il permet de créer?

Les métriques quantitatives et formelles sont importantes dans bien des domaines. L'idée de métriques spécifiquement pensées en fonction des systèmes OO n'est pas neuve, et peut facilement devenir le sujet de plusieurs guerres de clocher, de conflits philosophiques.

L'un des atouts de l'approche OO est qu'elle permet de modéliser les métriques elles-mêmes sous forme d'objets, ce qui rapproche la mesure du mesurable et peut faciliter l'intégration d'outils de mesure, de profilage et de diagnostic à des systèmes complexes.

Les langages OO sont nombreux, et une technique donnant de bons résultats avec l'un peut être beaucoup moins efficace si on cherche à l'appliquer avec l'autre[137].

[137] Les exemples typiques ici sont C++ et *SmallTalk*, souvent comparés l'un à l'autre, tous deux OO, mais parfois sujets à des stratégies de développement diamétralement opposées. Chacun a raison, vu de l'intérieur, parce que de son point de vue s'appliquent des techniques qui sont bonnes pour lui, qui sont appropriées à son outil. Il est difficile de développer des métriques objectives pour départager des points de vue philosophiques.

L'acte de mesurer

Une métrique dans le monde des technologies de l'information peut servir à plusieurs choses :

- déterminer quantitativement le degré de succès d'un produit, d'un individu ou d'un procédé, ce qui signifie se donner la possibilité de comparer les degrés de succès en question;
- déterminer quantitativement le progrès ou la dégradation d'un produit, d'un individu ou d'un procédé, ce qui peut permettre de choisir adéquatement des stratégies d'entretien ou de mise à jour et peut, si des indicateurs prédictifs sont identifiés, mener à la mise en place de stratégies d'entretien préventif;
- identifier les tendances, les modes et développer une vision macro du logiciel;
- justifier une décision administrative ou technique, comme par exemple la réécriture d'un module identifié comme goulot d'étranglement dans un système; *etc*.

Mesurer avant de trancher est une sage maxime. Tout comme on suggère aux programmeuses et aux programmeurs de mesurer rigoureusement[138] leurs programme avant et après chaque modification[139], on recommande aux gestionnaires de technologies de l'information de faire des choix éclairés par des mesures quantitatives avant d'appliquer une stratégie OO plutôt qu'une stratégie procédurale, et de faire de même avant d'appliquer une stratégie OO plutôt qu'une autre.

Mesurer est un acte rationnel. Choisir et mettre en place des métriques est un choix qui peut mener à l'application de changements stratégiques et à une étude rigoureuse des tenants et aboutissants de décisions techniques comme administratives.

Mesurer peut faire mal. C'est un acte en partie introspectif et qui peut mener à une autocritique. Après tout, qui est convaincu d'avoir raison n'a pas besoin de mesurer. Il importe donc de choisir les métriques avec soin, de sorte qu'elles soient appropriées au contexte (produit, technologies, gens, coutumes locales, *etc*.) et d'utiliser les mesures tirées de manière constructive.

Tout mesurer de manière exhaustive n'est pas possible en pratique. Il faut donc savoir choisir ce qui doit être mesuré et testé, tout comme il faut identifier ce qui est susceptible d'être vérifié de manière plus sommaire.

Certaines pratiques saines de documentation peuvent aider dans cette démarche, comme définir clairement, pour chaque sous-programme, les préconditions (*a priori*), les postconditions (*a posteriori*) et la complexité algorithmique en temps et en espace, mais ces pratiques sont surtout appliquées dans le code des bibliothèques et sont souvent des pratiques convenant autant aux fonctions plus traditionnelles et aux méthodes. Elles ne sont pas propres au modèle OO.

[138] Idéalement, à l'aide d'un profileur : les programmeuses et les programmeurs sont notoirement mauvais pour évaluer *de visu* les goulots d'étranglement de leurs programmes, étant trop attaché(e)s au fruit de leur labeur pour y poser un regard véritablement objectif.

[139] Ceci n'est pas seulement vrai d'un point de vue local : certaines optimisations locales peuvent entraîner une dégradation globale de performance!

Catégoriser les métriques[140]

Les métriques applicables au logiciel peuvent être catégorisées de plusieurs manières. Une répartition possible serait de considérer les métriques propres au produit, qui pourraient par exemple reposer sur :

- des considérations *internes*, donc connues de l'équipe de développement, telles que la taille, complexité ou l'adhésion aux standards locaux;
- des considérations *externes*, visibles aux usagers, comme le coût, la vitesse d'exécution ou le nombre de défectuosités;
- des considérations propres au *processus de développement* du logiciel, comme par exemple la rapidité avec laquelle le produit peut être développé, sa simplicité d'entretien, les mécanismes pour assurer le suivi des versions, ceux facilitant le suivi du progrès du développement, *etc.*

On pourrait aussi catégoriser les métriques en trois volets, soit :

- les *produits* : conformité des sources aux standards locaux, complexité algorithmique d'une fonction, taille de l'exécutable résultant, …;
- les *procédés* : facilité de mise à jour, temps requis pour développer un produit complet, manière de superviser la qualité du code, …; et
- les *gens* : productivité mensuelle, satisfaction de la clientèle, conformité du logiciel produit aux attentes, …

Une autre manière de catégoriser les métriques applicables à des projets OO serait orientée vers la qualité logicielle pour elle-même :

- nombre de classes et d'interfaces à entretenir;
- portabilité de l'implémentation;
- recours judicieux aux schémas de conception reconnus;
- impact d'une mise à jour ou d'un ajout;
- complexité algorithmique de certaines méthodes clés;
- couplage relatif des classes entre elles; *etc*.

Peu importe ce qu'on compte mesurer, qu'il s'agisse d'un projet, d'un produit, d'une gamme de produit, d'un procédé de développement, d'un mécanisme de suivi après-vente… Il importe de déterminer ce qu'on souhaite mesurer et de choisir la stratégie de mesure avec soin. Il n'y a pas de marteau doré; chaque problème doit être examiné au cas par cas pour qu'on puisse en tirer de l'information pertinente et utile.

Choisir une métrique appropriée à un domaine ou à une problématique donnée demande un investissement certain en temps et en effort. Il en va de même pour la mise au point d'un outil pour mettre en application une telle métrique.

[140] L'essentiel des idées amenées dans cette section proviennent de **Bertrand Meyer**, *The Role of Object-Oriented Metrics* [RoleOOM] et de Edward V. Berard, *Metrics for Object-Oriented Software Engineering* [MetOOSE].

Bien choisir une métrique est difficile, mais important puisque le retour sur l'investissement baisse lorsque le nombre de métriques devient trop grand[141]. Il peut arriver qu'on applique un grand nombre de métriques en début de parcours, à titre exploratoire, pour découvrir que certaines sont plus expressives que d'autres pour les problèmes investigués et, éventuellement, réduire l'éventail pour ne garder que les métriques reconnues comme vraiment utiles.

Étant donné la complexité des problèmes rencontrés en informatique, la grande majorité des experts recommande la saisie de mesures *fréquente* et *automatisée*, comme par exemple des tests unitaires (voir ***Principes OO et tests unitaires***, plus bas). Si l'information colligée ainsi est dans un format se prêtant à une analyse automatisée, des diagnostics mécaniques pourront même en être tirés et des analystes humains pourront éplucher les données sur une base régulière mais moins fréquente pour voir ce que l'examen automatisé aurait pu manquer.

Approche OO et métriques traditionnelles

L'approche OO diffère de l'approche procédurale à plusieurs égards :

- les objets encapsulent leurs attributs, ce qui entraîne des préoccupations de vitesse (vérifier que l'encapsulation a été faite de manière à éviter des goulots d'étranglement) et de sécurité (un bris d'encapsulation entraîne une difficulté accrue d'entretien à moyen terme);
- les objets regroupent souvent d'autres objets, par agrégation ou par composition. Ils représentent souvent des idées à part entière et peuvent même représenter opérationnellement des fonctions[142]. Caractériser un système à états encapsulés comme le sont les systèmes OO demande des métriques différentes de celles appliquées traditionnellement, surtout si le test se veut non intrusif;
- les objets sont en général pensés de manière à être réutilisés, ce qui implique par exemple qu'on ne voudra pas mesurer la productivité d'un programmeur à partir du nombre de lignes de code rédigées dans un intervalle de temps donné[143];
- l'approche OO vise la généralisation et, de plus en plus, la généricité. On voudra à la fois pouvoir mesurer le degré de généricité des types et algorithmes développés et vérifier à quel point l'application des stratégies génériques sera efficace pour une large gamme de types;
- l'approche OO met des objets en relation les uns avec les autres mais ces relations impliquent un certain degré de politesse (dans le respect de l'encapsulation). La plupart des métriques traditionnelles examinent la qualité du découpage et du regroupement d'un logiciel en fonctions, ce qui les rend difficile à adapter à un système OO. Entre autres, il n'y a pas de relation `1:1` entre objet et fonction puisqu'un objet peut exposer plusieurs méthodes et qu'une méthode peut déterminer une relation entre plusieurs objets;

---

[141] ¸Idéalement, appliquer entre trois et cinq métriques à une problématique donnée est la situation idéale. Passé ce seuil, les métriques coûtent aussi cher à mettre au point mais rapportent moins.

[142] Prudence : les états d'un foncteur peuvent changer pendant son utilisation, ce qui fait que la relation traditionnelle $y = f(x)$ des fonctions au sens mathématique peut ne pas s'appliquer en tout temps.

[143] Notez bien que, même dans une approche procédurale, compter les lignes de code rédigées dans un laps de temps donné est une excellente manière de se débarrasser accidentellement des employés productifs : souvent, une ligne bien écrite vaudra plus que dix lignes moins expressives.

- l'héritage public n'a pas d'équivalent dans une approche procédurale, et nécessite ses propres métriques du fait qu'il entraîne des relations de couplage entre classes, de complexité croissante des objets, et réutilisabilité du code, de qualité d'encapsulation, de profondeur de la dérivation (ce qui influence la complexité de l'objet et l'impact sur lui d'éventuelles mises à jour des parents), de la quantité de parents, de la quantité d'enfants, du travail de spécialisation à réaliser lorsqu'on dérive d'une classe donnée, *etc*.;
- les classes sont des unités susceptibles d'être instanciées. Chaque instance a une vie qui lui est propre, et certaines classes sont plus instanciées que d'autres, peuplant ainsi plus massivement l'espace d'un logiciel donné;
- certaines approches augmentent le temps de compilation pour réduire le temps d'exécution (pensez par exemple à la programmation générique), compliquant l'évaluation de leur impact sur le temps de développement;
- le polymorphisme, l'héritage, les attributs mutables, la POA (plus haut) sont tous des outils susceptibles de permettre l'insertion directement dans les objets de mécanismes pour mesurer les objets et leur environnement;
- enfin, il y a des divergences d'opinion, même (surtout?) parmi les experts, à savoir ce que signifie être OO (voir ***Être OO***, plus haut), ce qui complique la détermination de métriques véritablement utiles et universelles.

Visiblement, un travail de définition et d'implantation de métriques spécifiquement OO ou spécifiquement adaptées au monde OO est nécessaire.

Quelques candidats au titre de métrique OO

Il existe plusieurs candidats au titre de métrique pour évaluer la qualité d'un objet ou d'un système OO. On peut les regrouper en quatre familles :

- les *métriques plus traditionnelles*, applicables au moins en partie aux systèmes développés de manière procédurale;
- les *métriques de formation*, analogues à celles du même acabit dans le modèle procédural;
- les *métriques de production*, servant à mesurer la qualité des méthodes de travail et l'impact de l'approche OO sur la productivité; et
- les *métriques propres au développement*.

Dans tous les cas, il faut appliquer les métriques choisies avec discernement. Appliquer une métrique à l'aveugle équivaut à du gaspillage de ressources, et peut mener à des conclusions erronées, donc coûteuses et dangereuses.

Le terme ***regroupement*** est utilisé ci-dessous pour un ensemble de stratégies de regroupement d'objets (paquetages, espaces nommés, modules, assemblages, bibliothèques, *etc*.) dans le but d'alléger le propos.

Métriques plus traditionnelles

Taille du code source. On peut vouloir mesurer la taille, comptabilisée en nombre d'instructions, du code source. Dans la majorité des cas, on estimera que la complexité du programme croît avec la taille du code source ainsi mesuré, et on privilégiera les petites méthodes.

Comment : par classe, par méthode, par regroupement. On examine souvent des moyennes (taille moyenne des classes dans un regroupement; taille moyenne des méthodes dans une classe; taille moyenne des méthodes par classe; *etc*.).

Contre : les petits accesseurs/ mutateurs peuvent influencer les mesures et noyer des méthodes volumineuses. Donne une impression de complexité très locale. Il est possible de fausser le test par de mauvaises pratiques, comme en écrivant plusieurs méthodes de petite taille mais non pertinentes Une méthode très simple mais réalisant un accès JDBC à une base de données peut impliquer une dizaine de lignes de code très simple, et il en va de même pour un constructeur ou un opérateur d'affectation.

Proportion de commentaires. On peut vouloir comparer le rapport nombre de commentaires/ nombre de lignes de code. On estime habituellement qu'une plus haute proportion de commentaires par ligne de code signifie du code plus lisible.

Comment : par classe, par méthode, par regroupement.

Contre : découper le code en regroupements, classes, méthodes et ainsi de suite constitue un incitatif fort à la production de code autodocumenté, qui nécessite moins de commentaires que du code découpé en gros sous-programmes. Une application contre-productive serait d'insérer des commentaires superflus, ce qui peut nuire à la lisibilité si le code et les commentaires dérivent tous deux dans des directions différentes. Mesurer le nombre ou la proportion des commentaires ne dit rien sur leur pertinence ou sur le respect des standards d'entreprise. Que faire avec les commentaires générés automatiquement? Ne dit rien sur la justesse des commentaires, qui doivent être tenus à jour au même rythme que le code.

Taille du produit compilé. Un plus petit produit est plus facile à déployer, peut coûter moins cher à déployer et peut être diffusé par Internet.

Comment : par produit, par regroupement.

Contre : ne tient pas compte de considérations architecturales de plus haut niveau, par exemple d'un *Framework*. Dans le cas d'un produit voué à diffusion par Internet, les binaires voués à des machines virtuelles sont souvent préférables (très petits, leur code réel se trouve presque complètement logé dans le moteur de la machine virtuelle à laquelle ils sont destinés). Pour un système où les ressources sont plus rares (p. ex. : un système embarqué), charger une machine virtuelle en mémoire peut entraîner un coût prohibitif.

Métriques de formation

Temps requis pour apprendre l'approche OO en général[144]. Coût fixe (on n'a à apprendre le tout une seule fois), suivi de coûts récurrents plus faibles formation continue (mise à jour des acquis).

Métrique à examiner surtout si une entreprise doit former de nouveaux employés en vue de l'approche OO ou si elle doit assister ses employés dans leur mise à jour technologique.

Comment : formation à la pièce. Cours collégiaux ou universitaires. Formation sur place. Mentorat. Accès à de la documentation pertinente.

La métrique à évaluer tient à la qualité de la formation, au contenu dispensé et au temps requis pour arriver au stade souhaité de compétence.

Contre : prudence dans le choix des formateurs. Les erreurs sont coûteuses ici car elles se répercutent sur plusieurs personnes en même temps, tout en nuisant à la confiance envers l'approche OO et la compagnie. Prendre soin de comprendre la philosophie avant d'apprendre des outils spécifiques, du moins lors du premier contact. Éviter d'offrir une formation initiale puis aucun moyen de la mettre en pratique (gaspillage de ressources). Éviter de laisser un projet OO strictement aux mains de novices.

Temps requis pour apprendre une technologie OO donnée. Voir ***Temps requis pour apprendre l'approche OO en général***, à ceci près que les concepts doivent avoir été assimilés *a priori*. Le choix de la technologie est important, et il importe d'offrir des opportunités aux employés nouvellement formés pour que ceux-ci mettent en pratique leurs nouveaux acquis.

La clé du succès est de trouver l'équilibre entre les modes passagères et les concepts importants, et d'avoir une vision du futur de la compagnie et de la technologie. Une stratégie gagnante est de tester de nouvelles idées et de nouvelles technologies sur une base périodique chez des groupes choisis avant de se risquer à dépenser pour de la formation à grande échelle.

Métriques de production

Nombre de personnes/ jours pour développer et tester une classe. Cette métrique permet de mieux évaluer le temps de développement de produits logiciels et de mesurer le progrès l'assimilation du modèle OO chez les employés.

Devrait être plus long initialement que les équivalents structurés, puis raccourcir avec la croissance de nombre d'unités de code réutilisables dans la compagnie. Si le temps de développement ne baisse pas vers une valeur asymptotique, il y a lieu de s'interroger sur la qualité du design OO choisi (a-t-on tendance à refaire la roue?).

Comment : évaluer au préalable le temps estimé, comparer avec le temps réellement utilisé. Comparer le temps requis pour arriver à un résultat seul ou en équipe.

Contre : ne tient pas nécessairement compte des dépendances entre objets. N'a de sens que sur une longue période (les temps initiaux de développement auront tendance à être plus élevé que ceux notés quelques semaines ou quelques mois plus tard). Ne tient pas compte du fait que les objets peuvent évoluer sur une longue période, dans la mesure où leur barrière d'interface reste stable, sans véritablement affecter de manière adverse leur utilisation. Ne tient pas compte de l'intégration de la classe dans le contexte où elle sera utilisée.

[144] Pour en savoir plus, lire l'essai de maîtrise de **Pierre Boyer** à l'université de Sherbrooke, 2005, titré *Migration d'un environnement TI de programmation procédurale à un environnement orienté objet*.

Temps de rédaction de code. Dans une approche OO, on devrait noter une diminution du nombre de nouvelles lignes de code en comparaison avec l'équivalent procédural, mais un plus grand investissement de temps dans la recherche de classes déjà faites et offrant la fonctionnalité recherchée (un analogue à la recherche de fonctions dans des bibliothèques).

Contre : difficile à mesurer. Métrique indicative que le développement OO rapporte à la compagnie, mais qui ne peut être prise isolément. Devrait entraîner ou du moins accompagner une diminution des coûts en correctifs après livraison et en service après vente. Conséquemment, pourrait être déduite de phénomènes voisins plutôt que mesurée par et pour elle-même.

En général, on estime que cette répartition différente du temps de travail devrait mener à un risque plus faible d'erreurs du fait que les objets existants devraient avoir été testés et être pleinement encapsulés. Le temps économisé peut alors être investi dans une meilleure documentation et dans un meilleur design, donc dans de meilleurs produits.

Comment : difficile à mesurer de manière quantitative, mais peut se mesurer par voie de sondage auprès des programmeurs.

Nombre de jours requis pour livrer un système. Si le design est bon, le temps requis pour bâtir et livrer un système devrait réduire asymptotiquement avec le temps à cause de la réutilisation de code existant.

Contre : métrique très macro. Difficile d'isoler le facteur OO dans la réduction du temps de production. Dépend beaucoup du design et de la quantité de code réutilisable de manière propice au projet se trouvant à la disposition des programmeurs.

Comment : d'un point de vue macro, une bonne stratégie OO devrait entraîne une réduction de l'intervalle de temps entre le moment où un projet est entrepris et celui où le projet est livré. La phase de tests devrait être raccourcie par le fait même.

Nombre d'erreurs repérées. En temps normal, ce nombre devrait diminuer avec le temps à cause de la réutilisation de code existant et testé.

Contre : métrique pertinente mais sensible à des facteurs externes. En général, les tests unitaires sont une bonne chose, mais il ne faut pas négliger cette simple réalité de la vie qui nous rappelle que le code complexe et riche contient souvent plus d'erreurs que le code simple.

Comment : consulter les rapports générés à partir des processus de tests et de contrôle de la qualité de l'entreprise. Procéder systématiquement à des tests unitaires.

Métriques propres au développement

Complexité calculée en termes de nombre de méthodes par classe. Certains estiment qu'un trop grand nombre de méthodes par classe est une preuve de mauvais découpage, et nuit à la bonne catégorisation et au potentiel d'extensibilité d'une hiérarchie de classes. Une classe munie d'un éventail large de méthodes est difficile à tester de façon rigoureuse.

Contre : reflète un biais philosophique: certains prétendent qu'un grand nombre de méthodes par classe est une bonne stratégie, puisque les classes dérivées sont *de facto* capables de faire beaucoup de choses. Les stratégies stratifiées des langages à héritage simple ont une approche différente de celles des langages à héritage multiple. Ce qui est bon pour l'un peut l'être beaucoup moins pour l'autre. L'interprétation à donner à cette métrique doit être faite de manière prudente, variant selon les choix d'implantation et d'outils.

Comment : outils automatisés d'analyse du code. Compter manuellement si le code est vraiment petit. Se mesure par classe, par regroupement ou par hiérarchie. Dans un système, on peut vouloir mesurer le nombre moyen de méthodes par classe[145], le nombre minimal et le nombre maximal de méthodes par classe dans une hiérarchie donnée

Complexité calculée en termes de profondeur de l'arborescence d'héritage. L'héritage simple met l'accent sur des hiérarchies verticales, où les classes plus profondes dans une branche de l'arbre décrit par les relations d'héritage tendent à être plus riches en potentiel opérationnel. L'héritage multiple penche plus du côté de graphes plus plats, mettant en relief des hiérarchies plus horizontales, et constitue un enfant comme un assemblage ontique d'un ou de plusieurs parents.

Contre : l'adaptation de cette métrique à l'héritage multiple est sujette à débats. D'un point de vue philosophique, les langages à héritage simple mènent fréquemment à des hiérarchies plus verticales que les langages à héritage multiple, même si la complexité des classes terminales se ressemble dans les deux cas—la grosse différence entre les deux approches, en fait, tient surtout aux classes intermédiaires, plus massives dans un contexte d'héritage simple que dans un contexte d'héritage multiple.

Certains suggèrent conséquemment d'utiliser la longueur de la plus longue branche dans l'arbre décrit par les relations d'héritage (en situation d'héritage simple) comme métrique.

On pourrait utiliser, dans une situation plus générale, une métrique décrite par le rapport entre le nombre de méthodes héritées par une classe dérivée et la profondeur maximale où elle si situe si on examine toutes les branches menant de son plus ancien ancêtre jusqu'à elle. Une métrique connexe pour évaluer la qualité du design dans une hiérarchie de classes serait de mesurer le nombre de méthodes inutilisées dans les classes terminales.

Comment : se calcule pour une classe donnée, pour la moyenne des classes, pour la moyenne des classes terminales, *etc*.

Complexité et généricité calculée en termes de nombre de classes dérivées d'une classe donnée. Plus grand sera le nombre de classes dérivées d'une classe parent donnée et plus grand sera l'impact de cette classe parent sur le système dans son ensemble.

Contre : très pertinent pour jauger le facteur de risque associé à une classe dans une situation de développement interne, mais convient beaucoup moins lorsqu'une classe donnée est vouée à un déploiement commercial (qui sait alors ce que les gens en feront?).

Comment : outils automatisés, schémas `UML`.

[145] Nombre total de méthodes divisé par nombre total de classes dans le système; la manière de percevoir le polymorphisme et la surcharge simple dans une telle optique est sujette à débats.

Complexité en termes de couplage. Un objet est plus réutilisable s'il dépend de moins d'autres objets pour réaliser son travail. À l'inverse, plus il existe des liens entre les objets et plus les programmes les utilisant deviennent complexes à mettre à jour.

Un couplage fort est moins coûteux s'il s'applique à des objets d'un même regroupement logique (module, paquetage, espace nommé) que s'il concerne des objets de regroupements disjoints.

Plusieurs font la promotion d'objets les plus mutuellement indépendants possibles (*Low Coupling Objects*), qui sont plus faciles à réutiliser pour et par eux-mêmes. Ces objets tendent aussi à être plus faciles à tester, et à être plus faciles à déployer dans une grande variété de situations.

Comment : comptabiliser le nombre de relations une classe et d'autres classes ou nombre de classes avec lesquelles une classe entretient au moins une relation. On suggère de mesurer la dépendance mutuelle moyenne des objets dans un système par un rapport entre le nombre total de dépendances et le nombre total d'objets qui en font partie.

Contre : prise seule et de manière brute, cette métrique ne peut tenir compte des couplages choisis délibérément, et ne peut tenir compte non plus de certains couplages spécifiquement porteurs d'erreurs de conception. Donne par contre un indice global utile de la qualité du design système. Ce genre de métrique peut être difficile à déduire hors d'un contexte d'utilisation réel, et est plus facile à calculer à l'exécution si on a accès à des attributs de classe pour la comptabilité des données utilisées

Complexité de réponse. On peut mesurer la complexité des actions d'un objet en évaluant combien de méthodes risquent d'être invoquées lors de la réception d'un message (donc lors de l'invocation d'une méthode publique).

Comment : manuellement ou à l'aide d'outils automatisés, mesurer le nombre d'appels par méthode publique.

Contre : métrique dont la valeur est sujette à des fluctuations, étant associée directement à des détails soumis au principe d'encapsulation. Il est donc nécessaire de noter la date de chaque mesure, et de ne pas considérer cette métrique comme garante de valeurs définitives.

Cohésion. On peut vouloir comptabiliser le nombre de méthodes d'instance d'une classe donnée faisant directement référence à un même attribut. Un petit nombre est un bon indicateur de la stabilité d'un objet lorsque celui-ci doit évoluer.

Comment : examen à l'interne d'une classe, manuellement ou de manière automatique.

Contre : le respect strict du principe d'encapsulation rend caduque cette métrique (tout attribut ne devrait être référencé directement qu'une ou deux fois). Elle peut par contre être utile dans une situation où du code généré par un tiers doit être intégré à un système contrôlé de manière stricte.

Complexité structurelle immédiate. Quel est le nombre d'attributs à tenir à jour dans une classe donnée ou dans les classes d'un regroupement? Un petit nombre tend à mener à du code plus facile à tenir à jour.

Comment : comptabilité manuelle ou automatique. On pourrait trouver pertinent de connaître le minimum, le maximum, la moyenne et la médiane.

Contre : comme dans tout mécanisme du genre, il est difficile d'utiliser une telle métrique tout en tenant compte du fait que certaines classes sont foncièrement complexes, au sens où elles doivent tenir compte d'une multitude d'attributs. Cette métrique, si elle est appliquée à l'aveuglette, risque de faire ressortir comme étant très complexe des structures dont les objets sont variés comme les collections et les uplets.

Complexité *cyclomatique*. Comptabiliser le nombre de tests (alternatives, répétitives) requis pour qu'une méthode réalise sa tâche. Cousine de la complexité algorithmique. Ce type de métrique s'applique aussi à des bibliothèques développées dans un contexte procédural. Une complexité *cyclomatique* élevée est souvent un bon indicateur que certaines abstractions du modèle OO (en particulier le polymorphisme) ont été négligées.

Contre : métrique dont la valeur est sujette à des fluctuations, étant associée directement à des détails soumis au principe d'encapsulation. Il est donc nécessaire de noter la date de chaque mesure, et de ne pas considérer cette métrique comme garante de valeurs définitives

Comment : comptabilité manuelle ou automatique. On pourrait trouver pertinent de connaître le minimum, le maximum, la moyenne et la médiane. Peut être fait sur une base système, par regroupement ou sur la base des méthodes d'une certaine classe.

Complexité algorithmique. Ce type de métrique s'applique aussi à des bibliothèques développées dans un contexte procédural.

Contre : métrique dont la valeur est sujette à des fluctuations, étant associée directement à des détails soumis au principe d'encapsulation. Il est donc nécessaire de noter la date de chaque mesure, et de ne pas considérer cette métrique comme garante de valeurs définitives. Il faut clarifier le rôle des méthodes héritées (polymorphiques ou non) dans une telle métrique

Comme pour les sous-programmes traditionnels, on tend à favoriser les méthodes les plus simples possibles; celles dont la vocation est claire et sans équivoque.

Comment : comptabilité manuelle ou automatique. On pourrait trouver pertinent de connaître le minimum, le maximum, la moyenne et la médiane. Peut être fait sur une base système, par regroupement ou sur la base des méthodes d'une certaine classe.

Réutilisation par groupement logique. On peut tirer une moyenne de réutilisation de chaque groupement (p. ex. : `std::`, qui sert partout, semble très réutilisable).

Contre : bon indicateur macro de la qualité OO d'un design donné : les entités réutilisées massivement sont, par définition réutilisables. Ne met par contre pas en relief ce qui sert vraiment dans une classe ou dans un regroupement donné, et donne par conséquent une métrique incomplète. Pour que cette métrique soit vraiment pertinente, il faut aussi évaluer dans quelle mesure chaque objet est réutilisé, et examiner comment on évalue la réutilisation des classes parents servant aux objets fréquemment utilisés

Comment : comptabiliser, manuellement ou automatiquement, le nombre de fois qu'une classe ou qu'un regroupement est réutilisé dans un ou plusieurs projets. Dans une bibliothèque, calculer le rapport entre le nombre total d'objets réutilisés d'une bibliothèque et le nombre total d'objets disponibles dans cette bibliothèque.

Taux de réutilisation de méthodes héritées. Si une classe parent offre des méthodes à ses enfants, cela ne garantit pas que ces méthodes soient réinvesties.

Contre : en dit plus sur le design d'un parent que sur celui de ses enfants (du moins, ne permet pas de départager directement les deux). Comment devrait-on calculer le taux de réutilisation pour une classe dérivée d'une autre à travers une séquence de plusieurs ancêtres, ou à travers un treillis d'héritage virtuel. Les hiérarchies horizontales auront tendance à avoir un taux de réutilisation près de 100% alors que les hiérarchies verticales tendront à moins bien performer ici, par définition.

Comment : calculer le rapport entre le nombre de méthodes utilisées par une classe dérivée et le nombre de méthodes de son parent qui lui sont disponibles. Une métrique similaire peut être développée pour le nombre de méthodes surchargées (pour fins de polymorphisme ou non) par un descendant donné.

Taux de spécialisation de méthodes polymorphiques. À quel point les méthodes polymorphiques d'un parent donné sont-elles spécialisées par ses enfants?

Comment : calculer le rapport entre le nombre de méthodes spécialisées par une classe dérivée et le nombre de méthodes qu'il aurait pu spécialiser. D'un autre point de vue, si une hiérarchie se spécialise lentement, on peut calculer combien, parmi les méthodes polymorphiques d'un parent donné, seront effectivement spécialisées une fois une classe terminale conçue.

Contre : se prête mal à évaluer la situation dans un langage comme Java où toutes les méthodes d'instance privées ou protégées autres que les constructeurs et les méthodes qualifiées `final` sont, par définition, des méthodes polymorphiques. Comme dans le cas du taux de réutilisation des méthodes d'un parent, cette métrique ne permet pas facilement de discerner la différence entre la qualité du design du parent et la qualité du design de ses enfants.

Stabilité. À quelle fréquence une classe devra-t-elle être modifiée? Les changements aux volets public et protégé devraient tirer des sonnettes d'alarme.

Comment : tenir à jour des statistiques en cours de développement et après livraison.

Contre : si cette métrique révèle des irritants, ils sont nécessairement graves. S'applique bien à des classes concrètes. Pour les interfaces strictes, la réponse idéale devrait être jamais. Dans l'ensemble, devrait générer des statistiques différentes pour la stabilité des membres privés, protégés et publics (car les membres protégés devraient être très stables et les membres publics devraient être extrêmement stables).

Rapport entre complexité interne et complexité d'utilisation. Comme dans une approche client/ serveur, une classe peut être complexe à l'interne si elle est par le fait même plus simple à utiliser et à réutiliser.

Comment : cette métrique est une métrique synthétique, qui demande de déterminer *a priori* une métrique pour la complexité d'utilisation et une métrique pour la complexité interne (peut-être parmi celles proposées ici). Le rapport entre les deux métriques doit être porteur de sens.

Contre : toute métrique `OO` dépendant au moins en partie de l'utilisation faite d'un objet plutôt que sur sa structure interne est vouée à des fluctuations plus ou moins importantes en fonction des domaines d'application, surtout si l'objet est un tant soit peu générique (p. ex. : `std::vector`), ce qui complique les comparatifs de données saisies à l'aide de cette métrique si les domaines varient.

Nombre d'appels à une méthode donnée par intervalle de temps. Cette métrique devrait être faible pour les constructeurs, et servir d'indicateur de lieux à optimiser localement.

Comment : métrique dynamique, qui exige la mise en place d'un mécanisme automatique de surveillance, au minimum un compteur d'utilisation et une saisie du moment présente à la construction et à la destruction d'un objet (bien que ce ne soit pas suffisant pour une métrique précise).

Contre : si l'encapsulation est stricte et complète, cette métrique risque de rapporter des fréquences très élevées d'appels aux accesseurs de premier ordre. Il faut donc analyser les résultats des mesures générées à l'aide de cette métrique avec une certaine intelligence contextuelle.

Nombre de dérogations à l'approche OO. En général, offrir moins de lignes de code par méthode signifie aussi risquer moins de dérogations.

Comment : peut parfois être fait de manière automatique, mais reposera le plus souvent sur une analyse rigoureuse, elle-même réalisée par des experts humains.

Contre : il est difficile de préciser ce que signifie *déroger à l'approche `OO`* quand la définition de ce que signifie être `OO` est matière à débats. Des versions spécialisées de cette métrique peuvent par contre être envisagées (p. ex. : dérogations au principe d'encapsulation), bien qu'il puisse être difficile à déterminer comment repérer les dérogations (p. ex. : mesurer les points où le polymorphisme aurait dû être appliqué). Ne permet pas non plus de qualifier les dérogations (certaines sont peut-être faites pour des raisons techniques).

Cohésion interne. À quel point les méthodes sont-elles semblables et *regroupables* d'une classe à l'autre?

Plusieurs prétendent qu'une hiérarchie d'objets bien découpée mène à une faible redondance des méthodes dans le système.

Plus de cohésion peut être un indicateur d'un meilleur design.

Comment : calculer le rapport entre le nombre de méthodes uniques, pour lesquelles on ne trouve pas d'équivalent en terme de définition ailleurs dans la hiérarchie, et le nombre total de méthodes

Contre : attention de ne pas briser le design par une cohésion forcée. Ce rapport devrait tendre vers un pour les systèmes conçus par héritage multiple; eux seuls peuvent presque garantir une redondance zéro dans la définition des méthodes. Les systèmes développés par héritage simple sont destinés à avoir un rapport inférieur à un dans ce cas-ci[146], du moins pour la majorité des hiérarchies non banales.

Principes OO et tests unitaires

L'approche OO implique plusieurs considérations philosophiques et techniques qui influenceront l'application de stratégies de test, même plus traditionnelles.

Pour plus d'informations à ce sujet, voir [hdSQA].

Par l'**encapsulation**, un objet est présumé responsable de la qualité de ses états du début à la fin de son existence. C'est à la fois une prétention qu'a l'objet et qu'il faut valider lors de sa conception et, au sens d'un programme utilisant des objets, un axiome sur lequel il doit être possible de faire reposer d'autres assertions.

Il importe aussi que la documentation d'un objet clarifie ce que signifie *être en bon état* pour cet objet, donc quels en sont les invariants. L'implémentation d'un objet est présumée capable de mouvance dans la mesure où son opérationnalité (son visage public), elle, demeure stable. Il faut donc exprimer la documentation des états d'un objet en termes d'invariants, de comportements sur lesquels il est possible de compter, plutôt qu'en termes d'implémentation.

L'encapsulation stricte se teste, du point de vue du code client, par boîte noire. Les invariants par lesquels un objet décrit les garanties qu'il accepte d'offrir quant à ses états doivent être quantifiés, vérifiables et mesurables, de manière non intrusive.

L'implémentation de l'objet, elle, peut être testée avant livraison par boîte blanche, de manière intrusive et (si possible) exhaustive. Tout changement d'implémentation doit pouvoir être vérifié par boîte blanche avec une série de tests spécifiques à la nouvelle implémentation avant livraison, puis avec une série de tests par boîte noire stables peu importe l'implémentation.

Certains experts décriront l'**héritage** comme ajoutant de la complexité à la tâche de tester un objet, mais en réalité les conséquences de l'héritage sur la nature des tests ne devraient affecter que l'équipe de développement, pas le code client.

[146] Rapportez-vous à l'exemple sur les animaux et les insectes volants ou non, dans la section sur l'héritage multiple [POOv01] pour une ébauche d'explication pour appuyer cette affirmation.

L'héritage public (et, dans une perspective à peine moindre, l'héritage protégé) représente un cas d'héritage parmi plusieurs, et ajoute des éléments (ses membres publics) à l'interface publique des enfants. Le visage public d'un objet devrait être stable; conséquemment, modifier l'interface publique d'une classe après livraison, que cette classe soit un parent public d'une autre classe ou non, est une faute de design et implique une modification massive des tests à appliquer au parent comme à ses enfants. Le coût des fautes de design est au moins aussi lourd dans le monde OO que dans un monde procédural plus traditionnel.

Le caractère acceptable ou non des bris de contrat dans les interfaces est matière à débats. Certains vont jusqu'à prétendre qu'il faut concevoir les langages de manière à réduire les impacts d'un changement philosophique tardif dans les interfaces. C#, de par son exigence que les paramètres d'une méthode qualifiés de sortants purs (`out`) ou d'entrants et sortants (`ref`) le soient à la fois par l'appelant et par l'appelé est un bon exemple de cette nouvelle tendance.

Le secret du bonheur, ici comme ailleurs, est de penser les objets comme étant petits, avec une vocation claire et un petit nombre d'états bien définis, et de viser les hiérarchies les plus horizontales possibles. Le risque de se retrouver avec des hiérarchies complètes à repenser est alors beaucoup plus faible.

Le **polymorphisme** et l'**abstraction** ont tous deux beaucoup en commun avec l'encapsulation pour ce qui est de la conception de tests. Pris dans un sens général, ces deux idées ont en commun que, du point de vue du code client, elles tendent à être utilisées indirectement et définies dans une classe distincte de celle utilisée pour leur accéder.

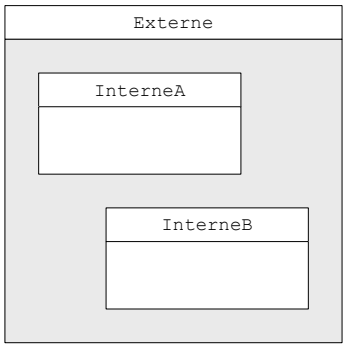
Ici aussi, donc, les abstractions doivent définir leurs prétentions comportementales sur la base d'invariants, les classes offrant les services polymorphiques doivent réaliser des tests en boîte blanche, adaptés aux implémentations individuelles, et le code client doit pouvoir compter sur des tests en boîte noire très stables.

Les tests en boîte blanche sont plus coûteux à définir et à réaliser que ne le sont les tests en boîte noire. Si le budget (ou, plus probablement, le temps) vient à manquer, il faut retenir que les tests en boîte noire, qui reflètent le point de vue du code client et reposent sur les invariants présumés des objets, sont les plus importants à définir et à réaliser.

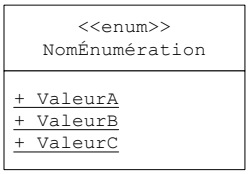
Annexe 00 – Résumé de la notation UML abordée dans ce document

Classes internes. Dans le schéma à droite, les classes `InterneA` et `InterneB` sont deux classes internes de la classe `Externe`.

On utilisera cette schématique dans le but de montrer la relation interne/ externe; cela dit, la plupart du temps, on présentera les classes internes comme des classes à part entière en laissant le nom de chaque classe indiquer la relation d'imbrication entre les classes (ex.: `Externe::Interne`).



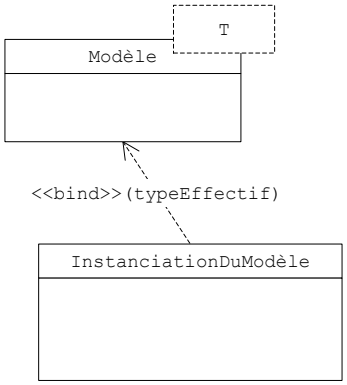
Énumération. Dans le schéma à droite, `NomÉnumération` est le nom d'un type énuméré pouvant prendre les valeurs `ValeurA`, `ValeurB` ou `ValeurC` inclusivement.



La mention `<<enum>>` est une clarification du contenu de la case. La notation `UML` permet d'appliquer une mention sous cette forme au besoin dans le but d'éliminer l'ambiguïté, effective ou apparente (ex.: `<<class>>` s'il pourrait ne pas être clair dans le contexte qu'il s'agit bien d'une classe).

Classes modèles (*Templates*, *Generics*). Dans le schéma à droite, `Modèle` est une classe générique (un *Template* au sens de C++) pour un type `T` donné, alors que `InstanciationDuModèle` est une application de `Modèle` pour laquelle `T` (dans la boîte pointillée) est `typeEffectif`.

Notez la relation entre les deux classes (une flèche pointillée) sur laquelle l'expression `<<bind>>(type)` indique à quel type correspond `T` pour cette application du modèle.



Annexe 01 – Discussions sur quelques réflexions choisies

Cette annexe présente quelques discussions portant sur des questions de réflexion parsemées ici et là dans le document. Ces questions ont été choisies (et posées) délibérément parce que les réponses ne sont pas aussi évidentes et banales qu'il n'y paraît à première vue.

Réflexion 03.0 : construction sécuritaire?

Ce code est-il sécuritaire face aux exceptions? Si oui, pourquoi? Sinon, quels ajustements devraient y être apportés?

La question de réflexion soulevée à droite a été soulevée aux côté du code de constructeurs de liste chaînée, où une répétitive (appel à `std::copy()`) ajoutait un à un des nœuds dans la liste.

La réponse à la question est simple : non, le code n'est pas sécuritaire face à une levée d'exceptions. En effet, le type `value_type` de la liste est inconnu *a priori* et son opérateur d'affectation est susceptible de lever une exception, peu importe la raison.

De toute manière, `push_back()` nécessite l'allocation dynamique d'un nœud, entraînant un risque de levée d'exception (`std::bad_alloc`).

Puisque le constructeur de la liste est en cours d'exécution, une levée d'exception signifierait que l'objet n'a jamais été construit. Conséquemment, son destructeur ne serait pas invoqué, et les ressources associées à chaque nœud seraient perdues.

La clé de la sécurisation de ce code est donc un *catch-any*, suivi de code de nettoyage et d'un *re-throw* (pas souci de neutralité). L'extrait à droite montre comment y arriver avec l'un constructeurs (celui de copie); vous pourrez extrapoler les autres à partir de cet exemple.

```
// ...
liste_simple(const liste_simple &lst)
   : tete_{}, nbelems_{} {
   try {
      copy(lst.begin(), lst.end(),
           back_inserter(*this));
   } catch (...) {
      clear(); // nettoyage
      throw;
   }
}
// ...
```

Annexe 02 – Le problème de l'objet omnipotent

L'un des plus mauvais réflexes des développeurs OO, défaut qui ne se limite pas nécessairement aux moins expérimentés d'entre eux, est celui de vouloir mettre au point des objets omnipotents, offrant tous les services possibles et impossibles.

Ce défaut se décline en plusieurs versions :

- créer des méthodes *sans se demander si elles sont vraiment utiles*. Cette remarque vise les accesseurs et les mutateurs sans se limiter à eux. Pourquoi tout attribut devrait-il nécessairement être exposé par un mutateur ou un accesseur? L'objet est une entité opératoire, présumée active et responsable, et son interface devrait être pensée comme telle;
- chercher à implémenter *à même l'objet* tous les services qu'un client pourrait vouloir éventuellement utiliser. Ceci tend à mener à des objets très lourds, très complexes à entretenir, et pas nécessairement plus utilisables.

Il n'est pas nécessaire de jouer à l'écureuil et d'enfouir un objet sous les méthodes d'instances. L'expérience nous a appris à penser nos objets plus simplement.

Ainsi, plutôt que de construire des objets très massifs et très complexes, préférez des objets petits et faciles à assembler entre eux. Pensez vos objets de manière à ce qu'il soit simple de les combiner. **Pensez petit**. Un objet qui fait peu de choses mais les fait bien est un objet *nettement* plus réutilisable que ne le serait un objet massif qui cherche à tout faire par lui-même.

Les flux, par exemple les classes `std::istream` et `std::ostream`, sont de bons exemples de cette approche. Ils offrent un ensemble simple de fonctionnalités fondamentales (on pourrait parler d'opérations élémentaires) à même l'interface directe de l'objet (ses méthodes d'instance). En général, les opérations élémentaires sont celles qui doivent accéder aux attributs pour accomplir leur mandat, et tendent à être très simples et très efficaces.

Par la suite, les opérations susceptibles d'être implémentées en combinant les opérations plus élémentaires plutôt qu'en accédant aux attributs de l'objet peuvent être implémentées sous forme de fonctions globales opérant sur l'objet ou en tant que services d'autres classes périphériques. Ceci permet un meilleur découplage des fonctionnalités.

Gardez au sein d'un objet les opérations qui y sont élémentaires, intrinsèques ou font véritablement partie de son identité. Découplez le reste : ceci vous permettra souvent d'exprimer des métaphores plus générales et plus faciles à réutiliser.

Pensez simple. Pensez petit. Pensez à des objets qui se combinent entre eux et qui savent collaborer. La meilleure classe est une classe vide[147].

[147] Cette prise de position, si radicale soit-elle, est une excellente maxime de programmation : les meilleurs objets sont les plus petits, et un algorithme reposant sur des classes vides est souvent un algorithme dont on peut être fier.

Annexe 03 – Itérer sur des valeurs (écriture alternative)

En réponse à une question d'un lecteur, **Herb Sutter** a posé[148] la question suivante (que je paraphrase) : « comment pourrait-on utiliser une syntaxe `for(auto …)` pour itérer à l'aide d'indices, sans que le code généré ne génère des avertissements comme ce serait le cas pour une écriture comme `for(auto i = 0; i < v.size(); ++i)` si le type de `v` est quelque chose comme `vector<int>` »?

Ma réponse fut ceci :

```
#include <iostream>
#include <vector>
#include <algorithm>
#include <iterator>
using namespace std;

template <class C>
   constexpr auto size(C &&cont) noexcept {
      return cont.size();
   }
template <class T, std::size_t N>
   constexpr auto size(T(&arr)[N]) noexcept {
      return N;
   }
template <class T, std::size_t N>
   constexpr auto size(const T(&arr)[N]) noexcept {
      return N;
   }

template <class T>
   class value_range : public std::iterator<std::bidirectional_iterator_tag, T> {
      T first, last;
   public:
      template <class U>
         class value_iterator {
            U cur;
         public:
            constexpr value_iterator(U init) : cur{ init } {
            }
            constexpr value_iterator& operator++() {
               ++cur;
               return *this;
            }
            constexpr value_iterator operator++(int) {
               auto temp{ *this };
```

[148] Voir http://herbsutter.com/2015/01/14/reader-qa-auto-and-for-loop-index-variables/

```
            operator++();
            return temp;
         }
         constexpr value_iterator& operator--() {
            --cur;
            return *this;
         }
         constexpr value_iterator operator--(int) {
            auto temp{ *this };
            operator--();
            return temp;
         }
         constexpr bool operator==(const value_iterator &other) const {
            return cur == other.cur;
         }
         constexpr bool operator!=(const value_iterator &other) const {
            return !(*this == other);
         }
         constexpr U operator*() {
            return cur;
         }
         constexpr U operator*() const {
            return cur;
         }
      };
   using iterator = value_iterator<T>;
   using const_iterator = value_iterator<T>;
   constexpr value_range(T first, T last) : first{ first }, last{ last } {
   }
   constexpr iterator begin() { return first; }
   constexpr iterator end() { return last; }
   constexpr const_iterator begin() const { return first; }
   constexpr const_iterator end() const { return last; }
   constexpr const_iterator cbegin() const { return first; }
   constexpr const_iterator cend() const { return last; }
};

template <class C>
   auto value_range_from(C &&cont) -> value_range<decltype(size(cont))> {
      return { 0, size(cont) };
   }

int main() {
   int arr[] { 2, 3, 5, 7, 11 };
   vector<int> v{ begin(arr), end(arr) };
   for (const auto &val : arr)
      cout << val << ' ';
   cout << '\n';
```

```
   for (const auto &val : v)
      cout << val << ' ';
   cout << '\n';
   for (auto i : value_range_from(arr))
         cout << "arr[" << i << "] == " << arr[i] << "; ";
   cout << '\n';
   for (auto i : value_range_from(v))
      cout << "v[" << i << "] == " << v[i] << "; ";
   cout << '\n';
}
```

Quelle serait la vôtre?

Les guides de déduction de C++ 17

Avec C++ 17, les guides de déduction (*Deduction Guides*) permettent d'alléger l'écriture de telles classes, en supprimant les fonctions génératrices comme `value_range_from()`. Ainsi, si nous déclarons plutôt ce qui suit :

```
template <lclass T>
   value_range(int first, T last) -> value_range<T>;
```

... signifiant « si l'on rencontre le nom `value_range` utilisé comme une fonction prenant un `int` et un `T`, alors appellez plutôt le constructeur de `value_range<T>` », le code client pourra maintenant s'écrire comme ceci :

```
int main() {
   int arr[] { 2, 3, 5, 7, 11 };
   vector<int> v{ begin(arr), end(arr) };
   for (const auto &val : arr)
      cout << val << ' ';
   cout << '\n';
   for (const auto &val : v)
      cout << val << ' ';
   cout << '\n';
   for (auto i : value_range(0, size(arr)) // <-- ICI
         cout << "arr[" << i << "] == " << arr[i] << "; ";
   cout << '\n';
   for (auto i : value_range(0, size(v)) // <-- ICI
      cout << "v[" << i << "] == " << v[i] << "; ";
   cout << '\n';
}
```

…ce qui est plutôt chouette à mon avis.

Individus

Les individus suivants sont mentionnés dans le présent document. Vous trouverez, en suivant les liens proposés à droite du nom de chacun, des compléments d'information à leur sujet et des suggestions de lectures complémentaires. Avis aux curieuses et aux curieux!

David Abrahams	http://h-deb.clg.qc.ca/Sujets/Orthogonal/Individus-importants.html#david_abrahams
Andrei Alexandrescu	http://h-deb.clg.qc.ca/Sujets/Orthogonal/Individus-importants.html#andrei_alexandrescu
Edward V. Berard	Je n'ai malheureusement pas d'informations à son sujet pour le moment.
Hand Boehm	http://h-deb.clg.qc.ca/Sujets/Orthogonal/Individus-importants.html#hans_boehm
Pierre Boyer	Maître en génie logiciel (TI) de l'Université de Sherbrooke, auteur de *Migration d'un environnement TI de programmation procédurale à un environnement orienté objet*
Chandler Carruth	http://h-deb.clg.qc.ca/Sujets/Orthogonal/Individus-importants.html#chandler_carruth
Marshall Cline	Je n'ai malheureusement pas d'information à son sujet au moment d'écrire ceci.
Edsger W. Dijkstra	http://h-deb.clg.qc.ca/Sujets/Orthogonal/Individus-importants.html#edsger_dijkstra
Gabriel Dos Reis	http://h-deb.clg.qc.ca/Sujets/Orthogonal/Individus-importants.html#gabriel_dos_reis
Vincent Echelard	http://h-deb.clg.qc.ca/Sujets/Orthogonal/Individus-importants.html#vincent_echelard
Richard Gilbert	Étudiant au DGL de l'université de Sherbrooke dans la 1re décennie du XXIe siècle.
Anders Hejlsberg	http://h-deb.clg.qc.ca/Sujets/Orthogonal/Individus-importants.html#anders_hejlsberg
Howard Hinnant	http://h-deb.clg.qc.ca/Sujets/Orthogonal/Individus-importants.html#howard_hinnant
Nicolai M. Josuttis	http://h-deb.clg.qc.ca/Sujets/Orthogonal/Individus-importants.html#nicolai_josuttis
Andrew Koenig	http://h-deb.clg.qc.ca/Sujets/Orthogonal/Individus-importants.html#andrew_koenig
Barbara Liskov	http://h-deb.clg.qc.ca/Sujets/Orthogonal/Individus-importants.html#barbara_liskov
Greg Lomow	Je n'ai malheureusement pas d'information à son sujet au moment d'écrire ceci.
Alisdair Meredith	http://h-deb.clg.qc.ca/Sujets/Orthogonal/Individus-importants.html#alisdair_meredith
Bertrand Meyer	http://h-deb.clg.qc.ca/Sujets/Orthogonal/Individus-importants.html#bertrand_meyer
Scott Meyers	http://h-deb.clg.qc.ca/Sujets/Orthogonal/Individus-importants.html#scott_meyers
Nathan Myers	http://h-deb.clg.qc.ca/Sujets/Orthogonal/Individus-importants.html#nathan_myers
Tim Peters	http://h-deb.clg.qc.ca/Sujets/Orthogonal/Individus-importants.html#tim_peters
Pierre Prud'homme	http://h-deb.clg.qc.ca/Sujets/Orthogonal/Individus-importants.html#pierre_prudhomme
Jeremy Siek	http://h-deb.clg.qc.ca/Sujets/Orthogonal/Individus-importants.html#jeremy_siek
Alexander Stepanov	http://h-deb.clg.qc.ca/Sujets/Orthogonal/Individus-importants.html#alexander_stepanov
Bjarne Stroustrup	http://h-deb.clg.qc.ca/Sujets/Orthogonal/Individus-importants.html#bjarne_stroustrup

Herb Sutter	http://h-deb.clg.qc.ca/Sujets/Orthogonal/Individus-importants.html#herb_sutter
Lamine Sy	Maître en génie logiciel (TI) de l'Université de Sherbrooke, auteur de *Mise en place de services Web avec la programmation orientée aspect*
Alan Turing	http://h-deb.clg.qc.ca/Sujets/Orthogonal/Individus-importants.html#alan_turing

Références

Les références qui suivent respectent un format quelque peu informel. Elles vous mèneront soit à des notes de cours de votre humble serviteur, soit à des documents pour lesquels mes remarques sont proposées de manière électronique et à partir desquels vous pourrez accéder aux textes d'origine ou à des compléments d'information.

[BDORSQL] http://www.oreilly.com/catalog/orsqlter/examples/object_syntax.pdf

[BoostFct] http://www.boost.org/doc/html/function.html

[CppAllo] http://www.roguewave.com/support/docs/sourcepro/stdlibref/allocator.html

[CppPerf] http://www.research.att.com/~bs/performanceTR.pdf

[EffCpp] http://h-deb.clg.qc.ca/Liens/Suggestions-lecture.html#effective-cpp

[EffStl] http://h-deb.clg.qc.ca/Liens/Suggestions-lecture.html#effective-stl

[ExcCpp] http://h-deb.clg.qc.ca/Liens/Suggestions-lecture.html#exceptional-cpp

[hdAllocCpp11] http://h-deb.clg.qc.ca/Sujets/Divers--cplusplus/allocateurs_cpp11.html

[hdAttPub] http://h-deb.clg.qc.ca/Sujets/Developpement/Encapsulation-attributs-publics.html

[hdConc] http://h-deb.clg.qc.ca/Liens/Langages-programmation--Liens.html#cplusplus_concept

[hdDeco] http://h-deb.clg.qc.ca/Sujets/Developpement/Schemas-conception.html#decorateur

[hdFComp] http://h-deb.clg.qc.ca/Sujets/Divers--cplusplus/Composition-fonctions-00.html

[hdGC] http://h-deb.clg.qc.ca/Liens/Gestion-memoire--Liens.html#collecte_ordures

[hdMicroFab] http://h-deb.clg.qc.ca/Sujets/Developpement/Technique-Microfabriques.html

[hdMOO] http://h-deb.clg.qc.ca/Sujets/Divers--cplusplus/index.html#metriques-OO

[hdNVI] http://h-deb.clg.qc.ca/Sujets/Developpement/Schemas-conception.html#nvi

[hdPatt] http://h-deb.clg.qc.ca/Sujets/Developpement/Schemas-conception.html

[hdProgLang] http://h-deb.clg.qc.ca/Liens/Langages-programmation--Liens.html

[hdProp] http://h-deb.clg.qc.ca/Sujets/Divers--cplusplus/Proprietes.html

[hdRatio] http://h-deb.clg.qc.ca/Sujets/TrucsScouts/rationnel.html

[hdSQA] http://h-deb.clg.qc.cq/Sujets/Developpement/Pratique-programmation.html#assurance_qualite_tests

[hdVarTmpl] http://h-deb.clg.qc.ca/Sujets/Divers--cplusplus/templates_variadiques.html

[hdVecTab] http://h-deb.clg.qc.ca/Sources/comparatif_vecteur_tableau.html

[HejlMS] http://msdn.microsoft.com/vcsharp/homepageheadlines/hejlsberg/default.aspx

[JavaInM] https://jsr-275.dev.java.net/files/documents/4333/34956/jsr-275.pdf

[MExcCpp] http://h-deb.clg.qc.ca/Liens/Suggestions-lecture.html#more-exceptional-cpp

[MetOOSE] http://www.toa.com/pub/moose.htm

[POOv00] POO – Volume 00, par Patrice Roy et Pierre Prud'homme.

[POOv01] POO – Volume 01, par Patrice Roy et Pierre Prud'homme.

[POOv02] POO – Volume 02, par Patrice Roy et Pierre Prud'homme.

[RoleOOM] http://archive.eiffel.com/doc/manuals/technology/bmarticles/computer/metrics/page.html

[StepDobb] http://www.sgi.com/tech/stl/drdobbs-interview.html
[Stl] http://www.sgi.com/tech/stl/index.html
[StlAllo] http://www.sgi.com/tech/stl/alloc.html
[StlBiComp] http://www.sgi.com/tech/stl/binary_compose.html
[StlChTr0] http://www.sgi.com/tech/stl/char_traits.html
[StlChTr1] http://www.sgi.com/tech/stl/character_traits.html
[StlCplx] http://www.sgi.com/tech/stl/complexity.html
[StlCtnr] http://www.sgi.com/tech/stl/Container.html
[StlFwIt] http://www.sgi.com/tech/stl/ForwardIterator.html
[StlItTag] http://www.sgi.com/tech/stl/iterator_tags.html
[StlSqnc] http://www.sgi.com/tech/stl/Sequence.html
[StlSort] http://www.sgi.com/tech/stl/sort.html
[StlUnF] http://www.sgi.com/tech/stl/unary_function.html
[StrouSell] http://lcsd05.cs.tamu.edu/papers/stroustrup.pdf
[StrouSellR] http://www.research.att.com/~bs/SELLrationale.pdf
[StrouSellH] http://www.research.att.com/~bs/SELL-HPC.pdf