PROGRAMMER EFFICACEMENT EN C++

42 conseils pour mieux maîtriser le C++ 11 et le C++ 14

Scott Meyers

Copyright © 2016 Dunod.

DUNOD

Authorized French translation *Effective Modern C++*,
ISBN 9781491903995 © 2015 Scott Meyers.
This translation is published and sold by permission of O'Reilly Media, Inc.,
which owns or controls all rights to publish and sell the same.

Traduit de l'américain par Hervé Soulard.

Conception de la couverture : Ellie Volkhausen
Illustratrice : Rebecca Demarest

Le pictogramme qui figure ci-contre mérite une explication. Son objet est d'alerter le lecteur sur la menace que représente pour l'avenir de l'écrit, particulièrement dans le domaine de l'édition technique et universitaire, le développement massif du photocopillage.
Le Code de la propriété intellectuelle du 1er juillet 1992 interdit en effet expressément la photocopie à usage collectif sans autorisation des ayants droit. Or, cette pratique s'est généralisée dans les établissements d'enseignement supérieur, provoquant une baisse brutale des achats de livres et de revues, au point que la possibilité même pour les auteurs de créer des œuvres nouvelles et de les faire éditer correctement est aujourd'hui menacée.
Nous rappelons donc que toute reproduction, partielle ou totale, de la présente publication est interdite sans autorisation de l'auteur, de son éditeur ou du Centre français d'exploitation du droit de copie (CFC, 20, rue des Grands-Augustins, 75006 Paris).



© Dunod, 2016
5, rue Laromiguière, 75005 Paris
www.dunod.com
ISBN 978-2-10-074846-4

Le Code de la propriété intellectuelle n'autorisant, aux termes de l'article L. 122-5, 2° et 3° a), d'une part, que les « copies ou reproductions strictement réservées à l'usage privé du copiste et non destinées à une utilisation collective » et, d'autre part, que les analyses et les courtes citations dans un but d'exemple et d'illustration, « toute représentation ou reproduction intégrale ou partielle faite sans le consentement de l'auteur ou de ses ayants droit ou ayants cause est illicite » (art. L. 122-4).
Cette représentation ou reproduction, par quelque procédé que ce soit, constituerait donc une contrefaçon sanctionnée par les articles L. 335-2 et suivants du Code de la propriété intellectuelle.

Copyright © 2016 Dunod.

Table des matières

Avant-propos VII

Introduction 1

Chapitre 1 – Déduction de type 9

Conseil n° 1. Comprendre la déduction de type de template 10

Conseil n° 2. Comprendre la déduction de type `auto` 18

Conseil n° 3. Comprendre `decltype` 23

Conseil n° 4. Afficher les types déduits 30

Chapitre 2 – `auto` 37

Conseil n° 5. Préférer `auto` aux déclarations de types explicites 37

Conseil n° 6. Opter pour un initialiseur au type explicite lorsque `auto` déduit des types non souhaités 43

Chapitre 3 – Vers un C++ moderne 49

Conseil n° 7. Différencier `()` et `{}` lors de la création des objets 49

Conseil n° 8. Préférer `nullptr` à `0` et à `NULL` 58

Conseil n° 9. Préférer les déclarations d'alias aux `typedef` 62

Conseil n° 10. Préférer les `enum` délimités aux `enum` non délimités 67

Conseil n° 11. Préférer les fonctions supprimées aux fonctions indéfinies privées ... 73

Copyright © 2016 Dunod.

© Dunod – Toute reproduction non autorisée est un délit.

Conseil n° 12. Déclarer les fonctions de substitution avec `override` 78

Conseil n° 13. Préférer les `const_iterator` aux `iterator` 84

Conseil n° 14. Déclarer `noexcept` les fonctions qui ne lancent pas d'exceptions 88

Conseil n° 15. Utiliser `constexpr` dès que possible 95

Conseil n° 16. Rendre les fonctions membres `const` sûres vis-à-vis des threads 101

Conseil n° 17. Comprendre la génération d'une fonction membre spéciale 106

Chapitre 4 – Pointeurs intelligents 115

Conseil n° 18. Utiliser `std::unique_ptr` pour la gestion d'une ressource à propriété exclusive 116

Conseil n° 19. Utiliser `std::shared_ptr` pour la gestion d'une ressource à propriété partagée 122

Conseil n° 20. Utiliser `std::weak_ptr` pour des pointeurs de type `std::shared_ptr` qui peuvent pendouiller 131

Conseil n° 21. Préférer `std::make_unique` et `std::make_shared` à une utilisation directe de `new` 136

Conseil n° 22. Avec l'idiome Pimpl, définir des fonctions membres spéciales dans le fichier d'implémentation 144

Chapitre 5 – Références rvalue, sémantique du déplacement et transmission parfaite 153

Conseil n° 23. Comprendre `std::move` et `std::forward` 154

conseil n° 24. Distinguer les références universelles et les références rvalue 160

Conseil n° 25. Utiliser `std::move` sur des références rvalue, `std::forward` sur des références universelles 164

Conseil n° 26. Éviter la surcharge sur les références universelles 173

Conseil n° 27. Se familiariser avec les alternatives à la surcharge sur les références universelles 179

Conseil n° 28. Comprendre la réduction de référence 191

Conseil n° 29. Supposer que les opérations de déplacement sont absentes, onéreuses et inutilisées 198

Conseil n° 30. Se familiariser avec les cas d'échec de la transmission parfaite 201

Copyright © 2016 Dunod.

Chapitre 6 – Expressions lambda 211

Conseil n° 31. Éviter les modes de capture par défaut 212

Conseil n° 32. Utiliser des captures généralisées pour déplacer des objets dans des fermetures 219

Conseil n° 33. Utiliser `decltype` sur des paramètres `auto&&` pour les passer à `std::forward` 225

Conseil n° 34. Préférer les expressions lambda à `std::bind` 228

Chapitre 7 – L'API de concurrence 237

Conseil n° 35. Préférer la programmation multitâche plutôt que multithread 237

Conseil n° 36. Spécifier `std::launch::async` si l'asynchronisme est primordial 242

Conseil n° 37. Rendre les `std::thread` non joignables par tous les chemins 246

Conseil n° 38. Être conscient du comportement variable du destructeur du descripteur de thread 253

Conseil n° 39. Envisager les futurs `void` pour communiquer ponctuellement un événement 257

Conseil n° 40. Utiliser `std::atomic` pour la concurrence, `volatile` pour la mémoire spéciale 265

Chapitre 8 – Finitions 275

Conseil n° 41. Envisager un passage par valeur pour les paramètres copiables dont le déplacement est bon marché et qui sont toujours copiés 275

Conseil n° 42. Envisager le placement plutôt que l'insertion 285

Index 295

Copyright © 2016 Dunod.

Avant-propos

Utiliser les exemples de code

Ce livre a comme objectif de vous aider. En règle générale, vous pourrez utiliser sans restriction les exemples de code de cet ouvrage dans vos programmes et vos documentations. Vous n'avez pas besoin de nous contacter pour une autorisation, à moins que vous ne vouliez reproduire des portions significatives de code. La conception d'un programme reprenant plusieurs extraits de code de cet ouvrage ne requiert aucune autorisation. Par contre, la vente et la distribution d'un CD-ROM d'exemples provenant des ouvrages O'Reilly en nécessitent une. Répondre à une question en citant le livre et les exemples de code ne requiert pas de permission. Par contre intégrer une quantité significative d'exemples de code extraits de ce livre dans la documentation de vos produits en nécessite une.

Nous apprécions, sans l'imposer, la citation de la source de ce code. Une citation comprend généralement le titre, l'auteur, l'éditeur et le numéro ISBN. Par exemple, « *Programmer efficacement en* C++, de Scott Meyers (Dunod). Copyright 2016 Dunod pour la version française 978-2-10-074391-9, et 2015 Scott Meyers pour la version d'origine 978-1-491-90399-5 ».

Si vous pensez que l'utilisation que vous avez faite de ce code sort des limites d'une utilisation raisonnable ou du cadre de l'autorisation ci-dessus, n'hésitez pas à nous contacter à l'adresse *permissions@oreilly.com*.

Commentaires et questions

Adressez vos commentaires et questions concernant ce livre à :

infos@dunod.com

Remerciements

C'est en 2009 que nous avons commencé à enquêter sur ce qui s'appelait alors C++0x (le C++11 naissant). Nous avons posté de nombreuses questions sur le

Copyright © 2016 Dunod.
© Dunod – Toute reproduction non autorisée est un délit.

groupe Usenet `comp.std.c++`, et nous sommes reconnaissants envers les membres de cette communauté (notamment Daniel Krügler) pour leurs réponses très utiles. Plus récemment, nous avons confié à Stack Overflow nos interrogations sur C++11 et C++14, et nous sommes tout aussi redevables à cette communauté pour son aide sur notre compréhension des plus petits détails du C++ moderne.

En 2010, nous avons préparé du contenu pour un cours sur C++0x (publié ensuite sous le titre *Overview of the New* C++, Artima Publishing, 2010). L'ensemble de ce contenu et mes connaissances ont largement bénéficié du travail d'investigation effectué par Stephan T. Lavavej, Bernhard Merkle, Stanley Friesen, Leor Zolman, Hendrik Schober et Anthony Williams. Sans leur aide, nous n'aurions probablement jamais été en mesure d'écrire *Effective Modern* C++. Ce titre a été suggéré ou approuvé par plusieurs lecteurs en réponse à notre billet du 18 février 2014, « Help me name my book » (*http://scottmeyers.blogspot.com/2014/02/help-me-name-my-book.html*). Endrei Alexandrescu (auteur de l'ouvrage *Modern* C++ *Design*, Addison-Wesley, 2001) a été très aimable de cautionner ce titre, qui reprend en partie ses termes.

Nous ne sommes pas en mesure de donner l'origine de toutes les informations présentées dans cet ouvrage, mais certaines sources ont eu une influence directe. Au **conseil 4**, l'utilisation d'un template indéfini pour forcer le compilateur à fournir une information de type a été suggérée par Stephan T. Lavavej, et Matt P. Dziubinski nous a indiqué Boost.TypeIndex. Au **conseil 5**, l'exemple `unsigned std::vector<int>::size_type` est extrait de l'article publié le 28 février 2010 par Andrey Karpov, « In what way can C++0x standard help you eliminate 64-bit errors » (*http://www.viva64.com/en/b/0060/*). L'exemple `std::pair<std::string, int>/std::pair<const std::string, int>` de ce même conseil est tiré de la présentation « STL11: Magic && Secrets » de Stephan T. Lavavej sur *Going Native 2012* (*http://channel9.msdn.com/Events/GoingNative/GoingNative-2012/STL11-Magic-Secrets*). Le **conseil 6** se fonde sur l'article publié le 12 août 2013 par Herb Sutter, « GotW #94 Solution: AAA Style (Almost Always Auto) » (*http://herbsutter.com/2013/08/12/gotw-94-solution-aaa-style-almost-always-auto/*). Le **conseil 9** prend ses racines dans le billet posté le 27 mai 2012 par Martinho Fernandes, « Handling dependent names » (*http://flamingdangerzone.com/cxx11/2012/05/27/dependent-names-bliss.html*). L'exemple du **conseil 12** illustrant la surcharge sur les qualificatifs de référence repose sur la réponse de Casey à la question « What's a use case for overloading member functions on reference qualifiers? » (*http://stackoverflow.com/questions/21052377/whats-a-use-case-for-overloading-member-functions-on-reference-qualifiers*) posée sur Stack Overflow le 14 janvier 2014. Au **conseil 15**, notre description de la prise en charge des fonctions `constexpr` comprend des informations fournies par Rein Halbersma. Le **conseil 16** emprunte énormément à la présentation « You don't know `const` and `mutable` » de Herb Sutter sur C++ *and Beyond 2012*. Le **conseil 18**, faire en sorte que les fonctions fabriques retournent des `std::unique_ptr`, provient de l'article publié le 30 mai 2013 par Herb Sutter, « GotW# 90 Solution: Factories » (*http://herbsutter.com/2013/05/30/gotw-90-solution-factories/*). Au **conseil 20**, la fonction `fastLoadWidget` a été suggérée par la présentation « My Favorite C++ 10-Liner » de Herb Sutter sur *Going Native 2013* (*http://channel9.msdn.com/Events/GoingNative/2013/My-Favorite-Cpp-10-Liner*). Nos explications, au **conseil 22**, sur `std::unique_ptr` et les types incomplets se fondent sur l'article publié le 27 novembre 2011 par Herb Sutter, « GotW #100: Compilation

Copyright © 2016 Dunod.

Firewalls » (*http://herbsutter.com/gotw/_100/*), ainsi que sur la réponse du 22 mai 2011 de Howard Hinnant à la question « Is std::unique_ptr<T> required to know the full definition of T? » posée sur Stack Overflow (*http://stackoverflow.com/questions/6012157/is-stdunique-ptrt-required-to-know-the-full-definition-of-t*). L'exemple d'addition de `Matrix` donné au **conseil 25** provient des publications de David Abrahams. Le commentaire rédigé le 8 décembre 2012 par JoeArgonne à propos du billet du 30 novembre 2012, « Another alternative to lambda move capture » (*http://jrb-programming.blogspot.com/2012/11/another-alternative-to-lambda-move.html*), est à l'origine de l'approche fondée sur `std::bind` pour la simulation de la capture généralisée de C++11 décrite au **conseil 32**. Les explications du **conseil 37** sur le problème du `detach` implicite dans le destructeur de `std::thread` sont tirées de l'article du 4 décembre 2008 rédigé par Hans-J. Boehm, « N2802: A plea to reconsider detach-on-destruction for thread objects » (*http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2008/n2802.html*). Le **conseil 41** a été motivé à l'origine par les interrogations de David Abrahams dans son billet du 15 août 2009, « Want speed? Pass by value. » (*http://web.archive.org/web/20140113221447/http:/cpp-next.com/archive/2009/08/want-speed-pass-by-value/*). L'idée que les types réservés au déplacement méritent un traitement particulier revient à Matthew Fioravante, tandis que l'analyse de la copie par affectation découle des commentaires de Howard Hinnant. Au **conseil 42**, Stephan T. Lavavej et Howard Hinnant nous ont aidés à comprendre les différences de performances entre les fonctions de placement et d'insertion, et Michael Winterberg a attiré notre attention sur les fuites de ressources potentielles liées au placement. Michael met ses informations au crédit de la présentation de Sean Parent, « C++ Seasoning », sur *Going Native 2013*, *http://channel9.msdn.com/Events/GoingNative/2013/Cpp-Seasoning*). Michael a également souligné l'utilisation de l'initialisation directe par les fonctions de placement, et celle de l'initialisation par copie par les fonctions d'insertion.

Le travail de relecture d'un ouvrage technique demande beaucoup d'implication, de temps et de critique. Nous sommes reconnaissants envers toutes les personnes qui ont accepté d'y participer. Les brouillons complets ou partiels de *Effective Modern C++* ont officiellement été relus par Cassio Neri, Nate Kohl, Gerhard Kreuzer, Leor Zolman, Bart Vandewoestyne, Stephan T. Lavavej, Nevin « :-) » Liber, Rachel Cheng, Rob Stewart, Bob Steagall, Damien Watkins, Bradley E. Needham, Rainer Grimm, Fredrik Winkler, Jonathan Wakely, Herb Sutter, Andrei Alexandrescu, Eric Niebler, Thomas Becker, Roger Orr, Anthony Williams, Michael Winterberg, Benjamin Huchley, Tom Kirby-Green, Alexey A Nikitin, William Dealtry, Hubert Matthews et Tomasz Kamiński. Nous avons également eu le retour de plusieurs lecteurs au travers de O'Reilly's Early Release EBooks et Safari Books Online's Rough Cuts, de commentaires sur notre blog *The View from Aristeia* (*http://scottmeyers.blogspot.com/*), et de courriers électroniques. Nous remercions tous ces contributeurs pour leur aide, dont cet ouvrage a largement profité. Nous sommes particulièrement redevables à Stephan T. Lavavej et Rob Stewart, dont les remarques extraordinairement détaillées et complètes laissent à penser qu'ils ont passé plus de temps sur cet ouvrage que nous-mêmes. Merci également à Leor Zolman, qui, outre sa relecture du manuscrit, a revérifié tous les exemples de code.

Gerhard Kreuzer, Emyr Williams et Bradley E. Needham se sont chargés de la révision des versions électroniques de ce livre.

Copyright © 2016 Dunod.

© Dunod – Toute reproduction non autorisée est un délit.

Notre choix de limiter la longueur des lignes de code se fonde sur les informations données par Michael Maher.

Grâce à Ashley Morgan Williams, nos dîners au Lake Oswego Pizzicato ont été particulièrement divertissants.

Plus de 20 ans après ma première expérience d'auteur, ma femme Nancy L. Urbano a encore une fois toléré les nombreux mois de conversations distraites, qu'elle a accompagnés d'un cocktail de résignation, d'exaspération et de débordements opportuns de compréhension et de soutien.

Copyright © 2016 Dunod.

Introduction

Si vous êtes un programmeur C++ expérimenté et si vous nous ressemblez, vous avez probablement abordé C++11 en pensant : « Oui, oui, j'ai compris. C'est du C++, juste amélioré. » Mais, en progressant dans votre apprentissage, vous avez dû être surpris par l'étendue des changements. Les déclarations `auto`, les boucles `for` basées sur une plage, les expressions lambda et les références rvalue ont changé la face de C++, sans parler des nouvelles fonctionnalités de concurrence. Ajoutons à cela les changements idiomatiques. `0` et `typedef` sont partis, bienvenue à `nullptr` et aux déclarations d'alias. Les énumérations peuvent à présent être délimitées. Les pointeurs intelligents doivent désormais être préférés aux pointeurs intégrés. Le déplacement des objets est normalement plus efficace que leur copie.

Nous avons beaucoup à découvrir sur C++11, et plus encore sur C++14.

Mais le plus important est que nous ayons beaucoup à apprendre sur l'utilisation *efficace* de ces nouvelles possibilités. Si vous recherchez des informations de base sur les fonctionnalités du C++ « moderne », les ressources abondent. En revanche, si vous cherchez à comprendre comment les employer pour créer un logiciel approprié, performant, facile à maintenir et portable, les difficultés commencent. C'est là où cet ouvrage peut vous être utile. Il est consacré non pas à la description des fonctionnalités de C++11 et de C++14, mais à leur mise en application efficace.

Les informations données dans cet ouvrage prennent la forme de recommandations réparties en **conseils**. Voulez-vous comprendre les différentes formes de déduction de type ? Souhaitez-vous savoir quand (ne pas) utiliser les déclarations `auto` ? Aimeriez-vous découvrir pourquoi les fonctions membres `const` doivent être sûres vis-à-vis des threads, comment implémenter l'idiome Pimpl avec `std::unique_ptr`, pourquoi éviter le mode de capture par défaut dans les expressions lambda, ou les différences entre `std::atomic` et `volatile` ? Toutes les réponses se trouvent ici. Elles sont indépendantes de la plate-forme et conformes à la norme. Cet ouvrage présente un C++ *portable*.

Les conseils font des recommandations, sans définir des règles, car il existe toujours des exceptions. Le point le plus important de chaque conseil est non pas la recommandation qu'il donne, mais les raisons qui l'étayent. Après les avoir étudiées, vous serez en mesure de déterminer si le cas particulier d'un projet justifie qu'une recommandation ne soit pas suivie. Le véritable objectif de ce livre n'est pas de préciser

Copyright © 2016 Dunod.
© Dunod – Toute reproduction non autorisée est un délit.

ce que vous devez faire ou ne pas faire, mais de vous apporter une compréhension plus profonde du fonctionnement de C++11 et de C++14.

Terminologie et conventions

Afin d'être certains que nous nous comprenions, il est important que nous soyons d'accord sur la terminologie, ne serait-ce que sur « C++ ». Il existe quatre versions officielles de C++, dont le nom fait référence à l'année d'adoption de la norme ISO correspondante : C++98, C++*03*, C++*11* et C++*14*. Puisque C++98 et C++03 diffèrent uniquement sur des détails techniques, nous les regroupons dans cet ouvrage sous le nom C++98. Lorsque nous mentionnons C++11, il s'agit à la fois de C++11 et de C++14, car C++14 et un sur-ensemble de C++11. Nous précisons C++14 lorsque les explications concernent uniquement cette version. Quant à C++, cela signifie que le contenu est suffisamment général pour correspondre à toutes les versions du langage (tableau 1).

Tableau 1 — Terminologie des versions de C++.

Terme employé	Versions du langage concernées
C++	Toutes
C++98	C++98 et C++03
C++11	C++11 et C++14
C++14	C++14

Par exemple, nous pouvons écrire que C++ met l'accent sur l'efficacité (vrai pour toutes les versions), que C++98 ne prend pas en charge la concurrence (vrai uniquement pour C++98 et C++03), que C++11 prend en charge les expressions lambda (vrai pour C++11 et C++14) et que C++14 offre la déduction généralisée du type de retour d'une fonction (vrai uniquement pour C++14).

La fonctionnalité C++11 la plus endémique est probablement la sémantique de déplacement, qui se fonde sur la distinction des expressions qui sont des *rvalues* et celles qui sont des *lvalues*. En effet, les rvalues signalent des objets éligibles aux opérations de déplacement, contrairement aux lvalues qui, en général, ne le sont pas. Conceptuellement (mais pas toujours en pratique), les rvalues correspondent à des objets temporaires retournés par des fonctions, tandis que les lvalues correspondent à des objets auxquels nous pouvons faire référence, que ce soit par leur nom ou en suivant un pointeur ou une référence lvalue.

Pour savoir si une expression est une lvalue, une méthode généraliste consiste à se demander s'il est possible d'en prendre l'adresse. Dans l'affirmative, il s'agit généralement d'une lvalue. Sinon, il s'agit habituellement d'une rvalue. Cette approche nous aide également à nous rappeler que le type d'une expression n'est pas lié au fait qu'elle soit une lvalue ou une rvalue. Autrement dit, étant donné le type `T`, nous pouvons avoir aussi bien des lvalues que des rvalues de type `T`. Il est important de ne pas oublier ce point lorsque nous manipulons un paramètre de type référence rvalue car le paramètre lui-même est une lvalue :

Copyright © 2016 Dunod.

```
class Widget {
public:
  Widget(Widget&& rhs);     // rhs est une lvalue, même si son type
  …                         // est une référence rvalue.
};
```

Dans cet exemple, nous pouvons parfaitement prendre l'adresse de `rhs` dans le constructeur de déplacement de `Widget`. Par conséquent, `rhs` est une lvalue même si son type est une référence rvalue. (Avec un raisonnement similaire, tous les paramètres sont des lvalues.)

Cet extrait de code illustre plusieurs conventions que nous allons suivre :

- La classe se nomme `Widget`. Nous utilisons `Widget` dès que nous voulons faire référence à un type quelconque défini par l'utilisateur. À moins que nous ne voulions montrer des détails spécifiques de la classe, nous employons `Widget` sans la déclarer.
- Le paramètre se nomme *`rhs`* (*right-hand side*, partie du côté droit). Ce nom a notre préférence pour les *opérations de déplacement* (constructeur de déplacement et opérateur d'affectation par déplacement) et pour les *opérations de copie* (constructeur de copie et opérateur d'affectation par copie). Nous l'employons également pour les paramètres placés à droite des opérateurs binaires :

```
Matrix operator+(const Matrix& lhs, const Matrix& rhs);
```

  Vous ne serez pas surpris d'apprendre que *`lhs`* (*left-hand side*) correspond à la partie du côté gauche.

- Nous utilisons une mise en forme spéciale pour les parties du code ou des commentaires qui exigent votre attention. Dans le constructeur de déplacement de `Widget`, nous avons mis en exergue la déclaration de `rhs` et la partie du commentaire qui révèle que `rhs` est une lvalue. Le code surligné n'est ni bon ni mauvais, il mérite simplement une attention particulière.
- Nous utilisons « … » pour indiquer que d'autres lignes de code se trouvent à cet emplacement. Il ne faut pas confondre ces points de suspension étroits avec les points de suspension larges (« . . . ») utilisés dans le code source pour les templates variadiques de C++11. Malgré les apparences, il n'y a pas de confusion possible. Par exemple :

```
template<typename… Ts>                // Points de suspension
void processVals(const Ts&… params)   // dans du code source
{                                     // C++.

  …                                   // Représente d'autres lignes
                                      // de code.
}
```

  La déclaration de `processVals` montre que nous utilisons `typename` pour déclarer des paramètres de type dans les templates, mais il s'agit d'une préférence

© Dunod – Toute reproduction non autorisée est un délit.
Copyright © 2016 Dunod.

personnelle. Le mot clé `class` convient également. Lorsque nous montrons du code qui provient de la norme C++, nous déclarons les paramètres de type avec `class` car c'est le mot clé qu'elle utilise.

Lorsque l'initialisation d'un objet se fait à partir d'un autre objet du même type, le nouvel objet est une *copie* de l'objet d'initialisation, même si la copie a été créée par le constructeur de déplacement. Malheureusement, la terminologie de C++ ne permet pas de distinguer un objet qui correspond à une copie construite par copie et un objet qui est une copie construite par déplacement :

```
void someFunc(Widget w);        // Le paramètre w de someFunc
                                // est passé par valeur.

Widget wid;                     // wid est un Widget.

someFunc(wid);                  // Dans cet appel à someFunc,
                                // w est une copie de wid créée via
                                // une construction par copie.

someFunc(std::move(wid));       // Dans cet appel à someFunc,
                                // w est une copie de wid créée via
                                // une construction par déplacement.
```

Les copies de rvalues sont généralement construites par déplacement, tandis que les copies de lvalues sont habituellement construites par copie. En conséquence, si nous savons uniquement qu'un objet est une copie d'un autre objet, il nous est impossible de connaître le coût de construction de cette copie. Par exemple, dans le code précédent, il est impossible de déterminer le coût de la création du paramètre `w` sans savoir si une rvalue ou une lvalue a été passée à `someFunc`. (Nous devons également connaître le coût du déplacement et de la copie des `Widget`.)

Dans un appel de fonction, les expressions passées au point d'appel constituent les *arguments* de la fonction. Ils servent à initialiser les *paramètres* de la fonction. Dans le premier appel à la fonction `someFunc` précédente, l'argument est `wid`. Dans le second appel, il s'agit de `std::move(wid)`. Dans ces deux appels, le paramètre est `w`. Il est important de faire la différence entre les arguments et les paramètres, car les paramètres sont des lvalues, alors que les arguments qui servent à leur initialisation peuvent être des rvalues ou des lvalues. Cela concerne en particulier le processus de *transmission parfaite*, au cours duquel un argument passé à une fonction est transmis à une seconde fonction en conservant le statut de rvalue ou lvalue de l'argument d'origine. (La transmission parfaite fait l'objet du conseil 30.)

Les fonctions bien conçues sont sûres vis-à-vis des exceptions. Autrement dit, elles offrent au moins une garantie de sécurité basique vis-à-vis des exceptions (la *garantie minimale*). Elles garantissent au code appelant que, même en cas d'exception, les invariants du programme sont conservés (aucune structure de données n'est corrompue) et aucune ressource n'est perdue. Les fonctions qui offrent une garantie de sécurité élevée vis-à-vis des exceptions (la *garantie forte*) garantissent au code appelant que, en cas d'exception, le programme reste dans l'état qu'il avait avant l'appel.

Copyright © 2016 Dunod.

Lorsque nous faisons référence à un *objet fonction*, nous parlons en général d'un objet dont le type prend en charge une fonction membre `operator()`. Autrement dit, il s'agit d'un objet qui se comporte comme une fonction. Nous employons parfois ce terme de façon plus générale pour désigner tout ce qui peut être invoqué à l'aide de la syntaxe d'un appel de fonction non-membre (c'est-à-dire « *`nomDeFonction(arguments)`* »). Cette définition plus large couvre non seulement les objets qui prennent en charge `operator()`, mais également les fonctions et les pointeurs de fonctions que l'on trouve en C. (La définition restrictive vient de C++98, la plus souple, de C++11.) En ajoutant les pointeurs de fonctions membres, nous arrivons à une généralisation encore plus importante : les *objets invocables*. Les distinctions fines peuvent en général être ignorées. Il suffit simplement de voir les objets fonctions et les objets invocables comme des éléments de C++ qui peuvent être invoqués au travers d'une certaine syntaxe d'appel de fonction.

Les objets fonctions créés par des expressions lambda sont appelés *fermetures*. Il est rarement nécessaire de distinguer les expressions lambda et les fermetures qu'elles génèrent. Nous conservons donc simplement le terme *expressions lambda*. De manière comparable, nous faisons rarement la différence entre les *templates de fonctions* (c'est-à-dire les templates qui génèrent des fonctions) et les *fonctions templates* (c'est-à-dire les fonctions générées à partir de templates de fonctions). Il en va de même pour les *templates de classes* et les *classes templates*.

En C++, de nombreux éléments peuvent être déclarés et définis. Une *déclaration* donne le nom et le type sans apporter d'autres détails, comme l'emplacement de la mémoire ou la manière d'implémenter les choses :

```
extern int x;                          // Déclaration d'un objet.

class Widget;                          // Déclaration d'une classe.

bool func(const Widget& w);            // Déclaration d'une fonction.

enum class Color;                      // Déclaration d'une énumération
                                       // délimitée (voir le conseil 10).
```

Une *définition* précise l'emplacement de la mémoire ou les détails d'implémentation :

```
int x;                                 // Définition d'un objet.

class Widget {                         // Définition d'une classe.
  ...
};

bool func(const Widget& w)
{ return w.size() < 10; }              // Définition d'une fonction.

enum class Color
{ Yellow, Red, Blue };                 // Définition d'une énumération
                                       // délimitée.
```

© Dunod – Toute reproduction non autorisée est un délit.
Copyright © 2016 Dunod.

Une définition est également une déclaration. Par conséquent, à moins qu'il ne soit réellement important d'avoir une définition, nous préférons les déclarations.

La *signature* d'une fonction correspond à la partie de sa déclaration qui précise les types des paramètres et le type de retour. Les noms de la fonction et des paramètres ne sont pas compris dans la signature. Dans l'exemple précédent, la signature de `func` est `bool(const Widget&)`. Les éléments de la déclaration d'une fonction autres que les types de ses paramètres et de sa valeur de retour (par exemple les mots clés `noexcept` ou `constexpr`, le cas échéant) sont exclus. (`noexcept` et `constexpr` sont décrits aux conseils 14 et 15.) La définition officielle d'une signature est légèrement différente de la nôtre (elle omet parfois le type de retour), mais, dans le cadre de cet ouvrage, notre définition est plus utile.

Les nouvelles normes de C++ préservent en général la validité du code écrit selon des normes plus anciennes, mais le comité de normalisation déclare parfois certaines fonctionnalités *obsolètes*. Ces fonctionnalités sont placées sur une voie de garage et risquent de disparaître des normes futures. Le compilateur informe parfois de l'utilisation des fonctionnalités obsolètes, mais il est préférable de les éviter. Elles peuvent non seulement conduire à des problèmes de portage ultérieur, mais elles sont également souvent inférieures aux fonctionnalités qui les remplacent. Par exemple, l'utilisation de `std::auto_ptr` est désapprouvée en C++11, car `std::unique_ptr` assure la même fonction, en mieux.

La norme stipule parfois qu'une opération a un *comportement indéfini*. Cela signifie que son comportement à l'exécution est imprévisible et il va sans dire qu'il vaut mieux rester loin d'une telle incertitude. Parmi les exemples de comportement indéfini mentionnons l'utilisation des crochets (« `[]` ») avec un indice qui dépasse les limites d'un `std::vector`, le déréférencement d'un itérateur non initialisé ou l'entrée dans une condition de concurrence (c'est-à-dire deux threads ou plus, l'un d'eux étant un écrivain, qui accèdent simultanément au même emplacement mémoire).

Les pointeurs intégrés, comme ceux renvoyés par `new`, sont appelés *pointeurs bruts*. À l'opposé d'un pointeur brut, nous trouvons le *pointeur intelligent*. Les pointeurs intelligents surchargent normalement les opérateurs de déréférencement d'un pointeur (`operator->` et `operator*`), mais le conseil 20 explique que `std::weak_ptr` fait exception.

Copyright © 2016 Dunod.

Signaler des bogues ou suggérer des améliorations

Nous avons fait de notre mieux pour que les informations données dans cet ouvrage soient claires, précises et utiles. Néanmoins, il reste toujours de la place pour des améliorations. Si vous trouvez des erreurs de quelque sorte que ce soit (techniques, explicatives, grammaticales, typographiques, etc.) ou si vous avez des suggestions pour améliorer cet ouvrage, n'hésitez pas à nous contacter[1] par courrier électronique à l'adresse *emc++@aristeia.com*. Les nouvelles impressions nous donnent l'opportunité de réviser *Programmer efficacement en* C++, mais nous ne pouvons pas traiter les problèmes dont nous n'avons pas connaissance !

Pour consulter la liste des problèmes connus rendez-vous sur la page dédiée[2] (*http://www.aristeia.com/BookErrata/emc++-errata.html*).

© Dunod – Toute reproduction non autorisée est un délit.
Copyright © 2016 Dunod.

1. En anglais de préférence.
2. Page de la version originale américaine.

1

Déduction de type

En C++98, un seul jeu de règles servait à déduire les types, celui employé pour les templates de fonctions. C++11 a ajouté deux règles, l'une pour `auto`, l'autre pour `decltype`. C++14 a ensuite étendu les contextes d'utilisation de ces deux mots clés. Grâce à une généralisation toujours plus importante de l'inférence de type, le programmeur n'est plus obligé de préciser les types qui sont évidents ou redondants. Le logiciel écrit en C++ devient plus flexible car la modification d'un type en un point du code source se propage automatiquement aux autres emplacements. En revanche, le code est peut-être plus difficile à analyser car les types déduits par les compilateurs risquent de ne pas apparaître aussi clairement que souhaité.

Sans une parfaite compréhension du fonctionnement de la déduction de type, il est pratiquement impossible de programmer efficacement dans un C++ moderne. Les contextes d'utilisation de l'inférence de type sont tout simplement trop nombreux : dans les appels aux templates de fonctions, dans la plupart des cas où `auto` apparaît, dans les expressions `decltype` et, depuis C++14, dans les énigmatiques constructions `decltype(auto)`.

Dans ce chapitre, le développeur C++ trouvera toutes les informations dont il a besoin sur la déduction des types. Nous y expliquons le fonctionnement de la déduction de type de template, comment elle est exploitée par `auto` et comment procède `decltype`. Nous précisons également comment obliger les compilateurs à dévoiler les résultats de leurs déductions afin que nous puissions vérifier qu'elles correspondent à nos attentes.

Copyright © 2016 Dunod.
© Dunod – Toute reproduction non autorisée est un délit.

CONSEIL N° 1. COMPRENDRE LA DÉDUCTION DE TYPE DE TEMPLATE

Lorsque l'on est capable d'utiliser un système complexe sans en comprendre le fonctionnement, tout en étant satisfait du résultat, on peut imaginer que ce système est bien conçu. Sur ce point, la déduction de type de template de C++ est une réussite incontestable. Des millions de programmeurs passent des arguments à des fonctions templates, avec des résultats totalement satisfaisants, et, pourtant, la plupart d'entre eux auraient bien du mal à donner une description autre que vague de la manière dont les types employés par ces fonctions ont été déterminés.

Si vous faites partie de ces personnes dans le flou, nous avons de bonnes et de mauvaises nouvelles. Tout d'abord, sachez que l'inférence de type pour les templates constitue le socle de l'une des fonctionnalités les plus intéressantes du C++ moderne : `auto`. Si vous étiez satisfait de la façon dont C++98 déduisait les types, vous ne serez pas déçu par la déduction de type `auto` en C++11. Cependant, l'application des règles dans le contexte de `auto` semblera parfois moins intuitive que dans le contexte des templates. Il est donc indispensable de maîtriser tous les aspects de la déduction de type de template sur lesquels se fonde `auto`. Tel est l'objectif de ce conseil.

Si vous êtes prêt à accepter un petit bout de pseudocode, voici comment se présente un template de fonction :

```
template<typename T>
void f(ParamType param);
```

Et voici comment se présente un appel à cette fonction :

```
f(expr);                    // Appeler f avec une expression.
```

Au cours de la compilation, le compilateur se sert de *expr* pour déduire le type de `T` et celui de *ParamType*. Ces types sont souvent différents car *ParamType* est généralement accompagné d'autres mots clés, comme `const` ou un qualificatif de référence. Supposons, par exemple, que le template soit déclaré de la manière suivante :

```
template<typename T>
void f(const T& param);     // ParamType correspond à const T&.
```

et que nous ayons l'appel suivant :

```
int x = 0;

f(x);                       // Appeler f avec un int.
```

`T` est déterminé comme étant de type `int`, mais *ParamType* est de type `const int&`.

Copyright © 2016 Dunod.

Il est normal d'imaginer que le type déduit pour `T` soit celui de l'argument passé à la fonction, autrement dit que `T` soit le type de *expr*. C'est le cas dans l'exemple précédent : `x` est un `int` et `T` est déterminé comme étant de type `int`. Mais cela ne fonctionne pas toujours ainsi. Le type déduit pour `T` dépend non seulement du type de *expr*, mais également de la forme de *ParamType*. Il existe trois cas :

- *ParamType* est un pointeur ou une référence, mais sans être une référence universelle. (Les références universelles font l'objet du conseil 24. À ce stade, sachez simplement qu'elles existent et sont différentes selon qu'elles sont des lvalues ou des rvalues.)
- *ParamType* est une référence universelle.
- *ParamType* n'est ni un pointeur ni une référence.

Nous devons donc étudier trois scénarios pour la déduction de type. Chacun reprend la forme générale d'un template et de son appel :

```
template<typename T>
void f(ParamType param);

f(expr);                // Déduire T et ParamType à partir de expr.
```

Cas 1 : `ParamType` est un pointeur ou une référence non universelle

Examinons la situation la plus simple, lorsque *ParamType* est un pointeur ou une référence, sans être une référence universelle. Dans ce cas, voici comment fonctionne la déduction de type :

1. Si le type de *expr* est une référence, ignorer la partie référence.
2. Effectuer ensuite une correspondance de motif entre le type de *expr* et *ParamType* de façon à déterminer `T`.

Prenons comme exemple le template suivant :

```
template<typename T>
void f(T& param);       // param est une référence.
```

et ces déclarations de variables :

```
int x = 27;             // x est un int.
const int cx = x;       // cx est un const int.
const int& rx = x;      // rx est une référence à x de type const int.
```

Voici les types déduits pour `param` et `T` dans les différents appels :

```
f(x);                   // T est de type int, param de type int&.

f(cx);                  // T est de type const int,
                        // param de type const int&.
```

Copyright © 2016 Dunod.
© Dunod – Toute reproduction non autorisée est un délit.

```
f(rx);                  // T est de type const int,
                        // param de type const int&.
```

Vous noterez que, dans les deuxième et troisième appels, puisque `cx` et `rx` désignent des valeurs `const`, `T` est déterminé comme étant `const int` et le type du paramètre est donc `const int&`. Ce point est important pour les appels. En effet, lorsqu'un objet `const` est passé à un paramètre de type référence, on suppose que cet objet reste non modifiable, c'est-à-dire que le paramètre est une référence à un `const`. C'est pour cette raison que passer un objet `const` à un template qui prend un paramètre `T&` est sûr : le caractère `const` de l'objet fait partie du type déduit pour `T`.

Dans le troisième exemple, vous remarquerez que, même si `rx` est de type référence, `T` est déterminé comme n'étant pas une référence. En effet, le fait que `rx` soit une référence est ignoré au cours de la déduction de type.

Les paramètres de tous ces exemples sont des références lvalue, mais la déduction de type opère de la même manière pour les références rvalue. Bien entendu, seules des rvalues peuvent être passées en arguments si les paramètres sont des références rvalue, mais cette contrainte n'a aucun rapport avec la déduction de type.

Si nous modifions le type du paramètre de `f`, en remplaçant `T&` par `const T&`, les résultats sont légèrement différents, mais sans réelle surprise. Le caractère `const` de `cx` et de `rx` est toujours respecté, mais, puisque nous supposons à présent que `param` est une référence à un `const`, il est inutile que `const` soit compris dans la déduction du type de `T` :

```
template<typename T>
void f(const T& param);  // param est une référence à un const.

int x = 27;              // Comme précédemment.
const int cx = x;        // Comme précédemment.
const int& rx = x;       // Comme précédemment.

f(x);                    // T est de type int, param de type const int&.

f(cx);                   // T est de type int, param de type const int&.

f(rx);                   // T est de type int, param de type const int&.
```

Comme précédemment, le fait que `rx` soit une référence est ignoré au cours de la déduction de type.

Si `param` était non plus une référence mais un pointeur, ou un pointeur sur un `const`, le fonctionnement serait quasi identique :

```
template<typename T>
void f(T* param);        // param est à présent un pointeur.

int x = 27;              // Comme précédemment.
const int *px = &x;      // px est un pointeur sur x de type const int.
```

Copyright © 2016 Dunod.

```
f(&x);                  // T est de type int, param de type int*.

f(px);                  // T est de type const int,
                        // param de type const int*.
```

Vous êtes probablement en train de bailler car les règles de déduction de type de C++ pour les paramètres de type référence et pointeur sont si naturelles que les lire se révèle particulièrement ennuyeux. Tout est si évident ! Mais c'est précisément ce que nous attendons d'un système de déduction de type.

Cas 2 : `ParamType` est une référence universelle

Le fonctionnement est moins évident lorsque les templates prennent en paramètres des références universelles. Ces paramètres sont déclarés comme des références rvalue (autrement dit, dans un template de fonction qui prend un paramètre de type `T`, la déclaration de type d'une référence universelle est `T&&`), mais le comportement est différent lorsque des arguments lvalue sont transmis. Tous les détails seront donnés au conseil 24, mais en voici une version résumée :

- Si *expr* est une lvalue, `T` et *ParamType* sont tous deux déterminés comme des références lvalue. C'est plutôt inhabituel, à deux points de vue. Premièrement, il s'agit du seul cas de déduction de type de template où `T` est déterminé comme une référence. Deuxièmement, même si *ParamType* est déclaré avec la syntaxe associée à une référence rvalue, son type déduit correspond à une référence lvalue.
- Si *expr* est une rvalue, les règles « normales » (c'est-à-dire le cas 1) s'appliquent.

Par exemple :

```
template<typename T>
void f(T&& param);      // param est à présent une référence universelle.

int x = 27;             // Comme précédemment.
const int cx = x;       // Comme précédemment.
const int& rx = x;      // Comme précédemment.

f(x);                   // x est une lvalue, T est donc de type int&
                        // et param également de type int&.

f(cx);                  // cx est une lvalue, T est donc de type const int&
                        // et param également de type const int&.

f(rx);                  // rx est une lvalue, T est donc de type const int&
                        // et param également de type const int&.

f(27);                  // 27 est une rvalue, T est donc de type int
                        // et param donc de type int&&.
```

Le conseil 24 explique précisément le fonctionnement de ces exemples. Il faut retenir ici que les règles de déduction de type pour les paramètres qui sont des

Copyright © 2016 Dunod.
© Dunod – Toute reproduction non autorisée est un délit.

références universelles diffèrent de celles des paramètres qui sont des références lvalue ou rvalue. En particulier, avec des références universelles, la déduction de type distingue les arguments lvalue et les arguments rvalue. Cela ne se produit jamais pour les références non universelles.

Cas 3 : `ParamType` n'est ni un pointeur ni une référence

Lorsque *`ParamType`* n'est ni un pointeur ni une référence, nous sommes dans le cas d'un passage par valeur :

```
template<typename T>
void f(T param);          // param est passé par valeur.
```

Cela signifie que `param` sera une copie de l'argument transmis, c'est-à-dire un objet totalement nouveau. Cet état de fait dicte les règles de déduction du type de `T` à partir de *`expr`* :

1. Comme précédemment, si le type de *`expr`* est une référence, ignorer la partie référence.
2. Si, après avoir ignoré la partie référence de *`expr`*, *`expr`* est `const`, ignorer également cette caractéristique. Faire de même s'il est `volatile`. (Les objets `volatile` sont rares et servent généralement à l'implémentation de pilotes de périphériques. Pour de plus amples informations, consultez le conseil 40.)

Poursuivons notre exemple :

```
int x = 27;           // Comme précédemment.
const int cx = x;     // Comme précédemment.
const int& rx = x;    // Comme précédemment.

f(x);                 // T et param sont tous deux des int.

f(cx);                // T et param sont à nouveau des int.

f(rx);                // T et param sont toujours des int.
```

Vous remarquerez que même si `cx` et `rx` représentent des valeurs `const`, `param` n'est pas `const`. C'est parfaitement normal car `param` est un objet totalement indépendant de `cx` et de `rx` – une *copie* de `cx` ou de `rx`. Le fait que `cx` et `rx` ne puissent pas être modifiés ne donne aucune indication sur les possibilités de modification de `param`. C'est pourquoi le caractère `const` (ou `volatile`) de *`expr`* est ignoré au moment de la déduction du type pour `param` : ce n'est pas parce que *`expr`* ne peut pas être modifié que sa copie ne peut pas l'être.

Il est important de comprendre que `const` (et `volatile`) est ignoré uniquement pour les paramètres passés par valeur. Nous l'avons vu, pour les paramètres qui sont des références ou des pointeurs vers des `const`, cette caractéristique de *`expr`* est préservée au cours de la déduction de type. Toutefois, examinons le cas où *`expr`* est un pointeur `const` sur un objet `const` et où *`expr`* est passé par valeur à `param` :

Copyright © 2016 Dunod.

```
template<typename T>
void f(T param);          // param est encore passé par valeur.

const char* const ptr =   // ptr est un pointeur const sur un objet
  "Fun with pointers ";   // const.

f(ptr);                   // Passer un argument de type
                          // const char * const.
```

Dans cet exemple, le mot clé `const` placé à droite de l'astérisque déclare que `ptr` est constant : il est impossible de faire pointer `ptr` ailleurs et il ne peut pas être fixé à `null`. (Le mot clé `const` à gauche de l'astérisque indique que l'élément pointé par `ptr` – la chaîne de caractères – est constant et qu'il ne peut donc pas être modifié.) Lorsque `ptr` est passé à `f`, les bits qui composent le pointeur sont copiés dans `param`. Le *pointeur lui-même (`ptr`) est donc passé par valeur*. Conformément à la règle de déduction de type pour les paramètres par valeur, le caractère `const` de `ptr` est ignoré et le type déduit pour `param` sera `const char*`, c'est-à-dire un pointeur modifiable sur une chaîne de caractères constante. Le caractère `const` de l'élément sur lequel `ptr` pointe est conservé pendant la déduction de type, mais celui de `ptr` lui-même est ignoré lors de sa copie pour créer le nouveau pointeur, `param`.

Tableaux en arguments

Voilà qui couvre essentiellement tous les cas généraux de la déduction de type de template, mais vous devez avoir connaissance d'un cas particulier. Il s'agit des types tableaux, qui sont différents des types pointeurs même s'ils semblent parfois interchangeables. En effet, dans de nombreux contextes, un tableau *se dégrade* (*decay*) en un pointeur sur son premier élément. C'est grâce à cette dégradation que le code suivant peut être compilé :

```
const char name[] = "J. P. Briggs";  // name est de type const char[13].

const char * ptrToName = name;       // Dégradation du tableau en
                                     // pointeur.
```

Le pointeur `ptrToName` de type `const char*` est initialisé avec `name`, qui est de type `const char[13]`. Ces deux types, `const char*` et `const char[13]`, ne sont pas identiques, mais, en raison de la règle de dégradation d'un tableau en pointeur, le code est compilable.

Mais que se passe-t-il lorsqu'un tableau est transmis à un template qui attend un paramètre passé par valeur ?

```
template<typename T>
void f(T param);      // Template avec un paramètre passé par valeur.

f(name);              // Quels types sont déduits pour T et param ?
```

Copyright © 2016 Dunod.
© Dunod – Toute reproduction non autorisée est un délit.

Observons tout d'abord qu'il n'y a aucun paramètre de fonction qui soit un tableau. La syntaxe est parfaitement valide :

```
void myFunc(int param[]);
```

mais la déclaration de tableau est traitée comme une déclaration de pointeur. Autrement dit, `myFunc` pourrait également être déclarée de la manière suivante :

```
void myFunc(int* param);          // Même fonction que précédemment.
```

Cette équivalence entre les paramètres de type tableau et de type pointeur prend ses racines dans le langage C ayant servi de base à C++ et nourrit l'illusion qu'il n'y a aucune différence entre les types tableau et pointeur.

Puisque la déclaration d'un paramètre de type tableau est traitée comme s'il s'agissait d'un paramètre de type pointeur, le type d'un tableau passé par valeur à une fonction template est déterminé comme étant de type pointeur. Cela signifie que, dans l'appel au template `f`, le type déduit pour `T` est `const char*` :

```
f(name);          // name est un tableau, mais le type déduit
                  // pour T est const char*.
```

Attention toutefois, car si les fonctions ne peuvent pas déclarer des paramètres qui soient réellement des tableaux, elles *peuvent* en déclarer qui sont des *références* à des tableaux ! Modifions le template `f` de sorte qu'il prenne son argument par référence :

```
template<typename T>
void f(T& param);       // Template avec un paramètre passé par
                        // référence.
```

et passons-lui un tableau :

```
f(name);                // Passer un tableau à f.
```

Dans ce cas, le type déduit pour `T` est le véritable type du tableau ! Il comprend la taille du tableau (dans cet exemple, le type déduit pour `T` est donc `const char [13]`) et le type du paramètre de `f` (une référence à ce tableau) est `const char (&)[13]`. La syntaxe a effectivement un côté malsain, mais la connaître vous permettra d'obtenir l'estime des quelques personnes qui y tiennent.

Grâce à cette possibilité de déclarer des références à des tableaux, nous pouvons créer un template qui déduit le nombre d'éléments contenus dans un tableau :

```
// Renvoyer la taille d'un tableau sous forme d'une constante
// définie à la compilation. (Le paramètre de type tableau n'a pas de
// nom, car seul le nombre d'éléments qu'il contient nous intéresse.)
template<typename T, std::size_t N>                  // Voir ci-après
constexpr std::size_t arraySize(T (&)[N]) noexcept   // pour constexpr
```

Copyright © 2016 Dunod.

```
{                                                    // et noexcept.
  return N;
}
```

Le conseil 15 l'expliquera, en déclarant cette fonction `constexpr` nous pouvons disposer de sa valeur de retour au moment de la compilation. Il est ainsi possible de déclarer, par exemple, un tableau avec le même nombre d'éléments qu'un deuxième tableau dont la taille est calculée à partir d'un initialiseur à accolades :

```
int keyVals[] = { 1, 3, 7, 9, 11, 22, 35 };      // keyVals contient
                                                 // 7 éléments.

int mappedVals[arraySize(keyVals)];              // Et donc mappedVals
                                                 // également.
```

Bien entendu, en tant que développeur C++ moderne, vous préférerez utiliser `std::array` pour construire un tableau :

```
std::array<int, arraySize(keyVals)> mappedVals;  // La taille de
                                                 // mappedVals est 7.
```

Quant à la déclaration `noexcept` de `arraySize`, il s'agit d'aider le compilateur à produire un meilleur code. Vous trouverez les informations détaillées au conseil 14.

Fonctions en arguments

En C++, la dégradation en pointeurs ne concerne pas uniquement les tableaux. Les types fonctions peuvent se dégrader en pointeurs de fonctions et toutes nos explications sur la déduction de type pour les tableaux s'appliquent également à celle des fonctions et à leur dégradation en pointeurs de fonctions. Par conséquent :

```
void someFunc(int, double);  // someFunc est une fonction,
                             // de type void(int, double).

template<typename T>
void f1(T param);            // Dans f1, param est passé par valeur.

template<typename T>
void f2(T& param);           // Dans f2, param est passé par référence.

f1(someFunc);                // Type déduit pour param : pointeur de
                             // fonction, de type void (*)(int, double).

f2(someFunc);                // Type déduit pour param : référence de
                             // fonction, de type void (&)(int, double).
```

Dans la pratique, cela fait rarement une différence. Mais si vous devez connaître la dégradation des tableaux en pointeurs, il n'est pas inutile de connaître également celle des fonctions en pointeurs.

Copyright © 2016 Dunod.
© Dunod – Toute reproduction non autorisée est un délit.

Et voilà pour les règles de déduction de type de template associées à `auto`. Nous avons mentionné leur relative simplicité au début du conseil et c'est le cas pour la plupart. Le traitement particulier accordé aux lvalues lors de la déduction des types pour des références universelles rend cependant les choses un peu plus confuses, tout comme les règles de dégradation des tableaux et des fonctions en pointeurs. Parfois, vous voudrez simplement demander à votre compilateur : « Dis-moi quel type tu as déterminé ! » Lorsque cela se produira, consultez le conseil 4 car il traite justement de cette question.

À retenir

- Lors de la déduction de type de template, les arguments qui sont des références sont traités comme s'ils n'étaient pas des références. Autrement dit, leur statut de référence est ignoré.
- Lors de la déduction de type pour des paramètres qui sont des références universelles, les arguments lvalue font l'objet d'un traitement spécifique.
- Lors de la déduction de type pour des paramètres passés par valeur, les arguments `const` et/ou `volatile` sont traités comme s'ils n'étaient pas `const` ou `volatile`.
- Lors de la déduction de type de template, les arguments qui sont des noms de tableaux ou de fonctions se dégradent en pointeurs, à moins qu'ils ne servent à initialiser des références.

CONSEIL N° 2. COMPRENDRE LA DÉDUCTION DE TYPE AUTO

Si vous avez lu le conseil 1 sur la déduction de type de template, vous savez déjà pratiquement tout ce que vous devez savoir sur la déduction de type `auto`. En effet, à une seule étrange exception près, elle est identique à celle de template. Comment peut-il en être ainsi ? La déduction de type de template implique des templates, des fonctions et des paramètres, alors que tous ces éléments sont absents avec `auto`.

C'est vrai, mais cela n'a pas d'importance. Il existe une correspondance directe entre la déduction de type de template et celle pour `auto`. Elles sont liées par une transformation algorithmique.

Au conseil 1, nous avons expliqué la déduction de type de template en utilisant le template de fonction général suivant :

```
template<typename T>
void f(ParamType param);
```

et cette forme d'appel :

```
f(expr);                    // Appeler f avec une expression.
```

Copyright © 2016 Dunod.

Dans un appel à `f`, le compilateur se sert de *`expr`* pour déduire le type de `T` et de *`ParamType`*.

Lorsqu'une variable est déclarée avec `auto`, ce mot clé tient le rôle de `T` dans le template, et le spécificateur de type de la variable, celui de *`ParamType`*. Ce fonctionnement sera plus simple à comprendre à partir d'exemples :

```
auto x = 27;
```

Dans ce cas, le spécificateur de type pour `x` est simplement `auto`. Avec la déclaration suivante :

```
const auto cx = x;
```

le spécificateur de type est `const auto`. Et là :

```
const auto& rx = x;
```

le spécificateur de type est `const auto&`. Pour déduire les types de `x`, `cx` et `rx` dans ces exemples, le compilateur fait comme s'il existait un template pour chaque déclaration ainsi qu'un appel à ce template avec l'expression d'initialisation appropriée :

```
template<typename T>                // Template conceptuel pour déduire
void func_for_x(T param);           // le type de x.

func_for_x(27);                     // Appel conceptuel : le type déduit
                                    // pour param est le type de x.

template<typename T>                // Template conceptuel pour déduire
void func_for_cx(const T param);    // le type de cx.

func_for_cx(x);                     // Appel conceptuel : le type déduit
                                    // pour param est le type de cx.

template<typename T>                // Template conceptuel pour déduire
void func_for_rx(const T& param);   // le type de rx.

func_for_rx(x);                     // Appel conceptuel : le type déduit
                                    // pour param est le type de rx.
```

Répétons-le, à une seule exception près, que nous verrons plus loin, la déduction des types pour `auto` est identique à celle pour les templates.

Au conseil 1, nous avons vu que la déduction de type de template considère trois cas, selon les caractéristiques de *`ParamType`*, le spécificateur de type pour `param` dans le template de fonction général. Dans la déclaration d'une variable avec `auto`, le spécificateur de type prend la place de *`ParamType`*, et nous avons donc encore trois cas :

- Cas 1 : le spécificateur de type est un pointeur ou une référence non universelle.
- Cas 2 : le spécificateur de type est une référence universelle.

Copyright © 2016 Dunod.
© Dunod – Toute reproduction non autorisée est un délit.

- Cas 3 : le spécificateur de type n'est ni un pointeur ni une référence.

Nous avons déjà présenté des exemples pour les cas 1 et 3 :

```
auto x = 27;        // Cas 3 (x n'est ni un pointeur ni une référence).

const auto cx = x;  // Cas 3 (idem pour cx).

const auto& rx = x; // Cas 1 (rx est une référence non universelle).
```

Le cas 2 ne vous surprendra pas :

```
auto&& uref1 = x;   // x est un int et une lvalue,
                    // uref1 est donc de type int&.

auto&& uref2 = cx;  // cx est un const int et une lvalue,
                    // uref2 est donc de type const int&.

auto&& uref3 = 27;  // 27 est un int et une rvalue,
                    // uref3 est donc de type int&&.
```

À la fin du conseil 1, nous avons expliqué la dégradation des tableaux et des fonctions en pointeurs lorsque les spécificateurs de types ne sont pas des références. Cela se produit également lors de la déduction de type `auto` :

```
const char name[] =           // name est de type is const char[13].
  "R. N. Briggs";

auto arr1 = name;             // arr1 est de type const char*.

auto& arr2 = name;            // arr2 est de type const char (&)[13].


void someFunc(int, double);   // someFunc est une fonction,
                              // de type void(int, double).

auto func1 = someFunc;        // func1 est de type
                              // void (*)(int, double).

auto& func2 = someFunc;       // func2 est de type
                              // void (&)(int, double).
```

Vous le constatez, la déduction de type `auto` se passe de la même manière que la déduction de type de template. Elles sont au fond les deux faces d'une même pièce.

Mais voici leur point de divergence. Commençons par observer que, pour déclarer un `int` de valeur initiale 27, C++98 offre deux syntaxes :

```
int x1 = 27;
int x2(27);
```

Copyright © 2016 Dunod.

C++11, malgré sa prise en charge de l'initialisation uniforme, ajoute les possibilités suivantes :

```
int x3 = { 27 };
int x4{ 27 };
```

Autrement dit, quatre syntaxes pour un seul et même résultat : un `int` de valeur 27.

Cependant, comme nous l'expliquerons au conseil 5, la déclaration de variables avec `auto` à la place d'un type figé présente quelques avantages. Il serait donc bon de remplacer `int` par `auto` dans les déclarations de variables précédentes :

```
auto x1 = 27;
auto x2(27);
auto x3 = { 27 };
auto x4{ 27 };
```

Si la compilation de ces déclarations ne pose pas de difficultés, elles n'ont pas la même signification que celles d'origine. Les deux premières instructions déclarent bien une variable de type `int` qui a la valeur 27. En revanche, les deux dernières déclarent une variable de type `std::initializer_list<int>` qui contient un seul élément ayant la valeur 27 !

```
auto x1 = 27;             // De type int, de valeur 27.

auto x2(27);              // Idem.

auto x3 = { 27 };         // De type std::initializer_list<int>,
                          // de valeur { 27 }.

auto x4{ 27 };            // Idem.
```

Ce résultat vient d'une règle particulière de la déduction de type pour `auto`. Lorsque l'initialiseur d'une variable déclarée avec `auto` est placé entre accolades, le type déduit est `std::initializer_list`[1]. S'il n'est pas possible de déduire ce type, par exemple en raison de valeurs entre accolades de types différents, le code est rejeté :

```
auto x5 = { 1, 2, 3.0 };  // Erreur ! Impossible de déduire T pour
                          // std::initializer_list<T>.
```

Comme l'explique le commentaire, la déduction de type échoue dans ce cas, mais il est important de comprendre qu'en réalité deux sortes de déduction de type ont

1. En novembre 2014, le Comité de normalisation de C++ a adopté la proposition N3922, qui supprime la règle spécifique de déduction de type pour `auto` et les initialiseurs entre accolades avec une syntaxe d'initialisation directe, c'est-à-dire sans signe « = » avant les accolades (voir le conseil 42). Avec la proposition N3922 (qui ne fait pas partie de C++11 et de C++14, mais qui a été mise en œuvre par certains compilateurs), `x4` dans les exemples précédents n'est plus de type `std::initializer_list<int>` mais `int`.

Copyright © 2016 Dunod.
© Dunod – Toute reproduction non autorisée est un délit.

lieu. La première découle de l'utilisation de `auto` : le type de `x5` doit être déduit. Puisque l'initialiseur de `x5` est placé entre des accolades, le type déduit pour `x5` doit être `std::initializer_list`. Mais `std::initializer_list` est un template. Pour un type `T`, les instanciations sont `std::initializer_list<T>`, et cela signifie que le type de `T` doit aussi être déduit. Cette déduction est du domaine de la seconde sorte : la déduction de type de template. Dans cet exemple, cette déduction échoue car les valeurs d'initialisation placées entre les accolades ne sont pas d'un seul et même type.

C'est uniquement dans le traitement des initialiseurs entre accolades que la déduction de type `auto` et la déduction de type de template diffèrent. Lorsqu'une variable déclarée avec `auto` est initialisée à l'aide de valeurs entre accolades, le type déduit est une instanciation de `std::initializer_list`. Cependant, si le même initialiseur est passé au template correspondant, la déduction de type échoue et le code est rejeté :

```
auto x = { 11, 23, 9 };    // x est de type
                           // std::initializer_list<int>.

template<typename T>       // Template avec une déclaration de paramètre
void f(T param);           // équivalente à la déclaration de x.

f({ 11, 23, 9 });          // Erreur ! Impossible de déduire le type
                           // pour T.
```

Toutefois, si nous indiquons dans le template que `param` est un `std::initializer_list<T>` pour un `T` inconnu, la déduction de type de template pourra déterminer le type de `T` :

```
template<typename T>
void f(std::initializer_list<T> initList);

f({ 11, 23, 9 });          // Le type déduit pour T est int, et le type
                           // de initList est std::initializer_list<int>.
```

Par conséquent, la seule véritable différence entre les déductions de type `auto` et celle de template vient du fait que, dans le cas de `auto` et contrairement aux templates, un initialiseur entre accolades est *supposé* représenter un `std::initializer_list`.

Vous vous demandez peut-être pourquoi la déduction de type `auto` emploie une règle particulière pour les initialiseurs entre accolades, ce que ne fait pas celle de template. Nous nous posons la même question. Hélas, nous n'avons pas pu trouver d'explication convaincante. Mais les règles sont les règles, et vous ne devez donc pas oublier que si vous déclarez une variable avec `auto` et l'initialisez avec des valeurs entre accolades, le type déduit sera toujours `std::initializer_list`. Il est particulièrement important de garder ce point à l'esprit si vous adoptez l'initialisation uniforme, c'est-à-dire placez les valeurs d'initialisation entre accolades. Avec C++11, l'une des erreurs classiques est de déclarer par mégarde une variable `std::initializer_list` alors que l'objectif était une autre déclaration. C'est en raison de ce danger que certains

Copyright © 2016 Dunod.

développeurs placent des accolades autour des initialiseurs uniquement lorsqu'elles sont indispensables (le conseil 7 explique quand cela est nécessaire).

L'histoire se termine là pour C++11, mais elle continue pour C++14. En C++14, il est possible d'utiliser `auto` pour indiquer que le type de retour d'une fonction doit être déduit (voir le conseil 3) et les déclarations de paramètre des expressions lambda peuvent employer `auto`. Cependant, dans ces utilisations de `auto`, la déduction de type concernée est non pas celle de `auto` mais celle de template. Par conséquent, le compilateur refusera une fonction qui a un type de retour `auto` et qui renvoie un initialiseur entre accolades :

```
auto createInitList()
{
  return { 1, 2, 3 };          // Erreur ! Impossible de déduire le type
}                              // pour { 1, 2, 3 }.
```

La situation est identique lorsque la spécification du type d'un paramètre d'une expression lambda en C++14 utilise `auto` :

```
std::vector<int> v;
…

auto resetV =
  [&v](const auto& newValue) { v = newValue; };     // C++14.

…

resetV({ 1, 2, 3 });           // Erreur ! Impossible de déduire le type
                               // pour { 1, 2, 3 }.
```

À retenir

- La déduction de type `auto` est en général identique à la déduction de type de template, mais elle suppose qu'un initialiseur entre accolades représente un `std::initializer_list`, ce qui n'est pas le cas de la déduction de type de template.
- L'utilisation de `auto` dans le type de la valeur de retour d'une fonction ou d'un paramètre d'une expression lambda conduit à une déduction de type de template, non à la déduction de type `auto`.

CONSEIL N° 3. COMPRENDRE DECLTYPE

`decltype` est une créature plutôt étrange. Si nous lui donnons un nom ou une expression, elle vous indique le type de ce nom ou de cette expression. En général, `decltype` indique exactement ce que nous avions prévu, mais il arrive parfois qu'elle donne des résultats qui risquent de nous laisser pantois et qui nous inciteront à consulter des sites de référence ou des FAQ.

Copyright © 2016 Dunod.
© Dunod – Toute reproduction non autorisée est un délit.

Nous allons commencer avec des cas classiques, ceux sans surprises. Au contraire de ce qui se passe lors de la déduction de type pour les templates et `auto` (voir les conseils 1 et 2), `decltype` se contente de répéter le type exact du nom ou de l'expression qui lui est donné :

```
const int i = 0;            // decltype(i) donne const int.

bool f(const Widget& w);    // decltype(w) donne const Widget&.
                            // decltype(f) donne bool(const Widget&).

struct Point {
  int x, y;                 // decltype(Point::x) donne int.
};                          // decltype(Point::y) donne int.

Widget w;                   // decltype(w) donne Widget.

if (f(w)) …                 // decltype(f(w)) donne bool.

template<typename T>        // Version simplifiée de std::vector.
class vector {
public:
  …
  T& operator[](std::size_t index);
  …
};

vector<int> v;              // decltype(v) donne vector<int>.
…
if (v[0] == 0) …            // decltype(v[0]) donne int&.
```

Vous le constatez, aucune surprise.

En C++11, `decltype` est probablement utilisé essentiellement pour la déclaration de templates d'une fonction dont le type de la valeur de retour dépend des types de ses paramètres. Par exemple, supposons que nous souhaitions écrire une fonction prenant en arguments un conteneur qui accepte l'indexation sous forme de crochets (c'est-à-dire « `[]` ») et un indice. Elle commence par authentifier l'utilisateur avant de retourner le résultat de l'indexation. Le type de retour de la fonction doit être identique à celui retourné par l'opération d'indexation.

En invoquant `operator[]` sur un conteneur d'objets de type `T`, nous obtenons en général un élément de type `T&`. C'est par exemple le cas pour `std::deque`, et presque toujours le cas pour `std::vector`. En revanche, pour `std::vector<bool>`, `operator[]` ne renvoie pas un `bool&` mais un tout nouvel objet. Les détails de ce fonctionnement seront révélés au conseil 6. L'important ici est que le type retourné par la fonction `operator[]` d'un conteneur dépend de ce conteneur.

Avec `decltype`, il est facile d'obtenir le bon fonctionnement. Voici une version mal dégrossie du template que nous aimerions écrire. Elle montre l'utilisation de `decltype` pour déterminer le type de la valeur de retour. Le template aura besoin de quelques ajustements, mais nous y reviendrons ultérieurement :

Copyright © 2016 Dunod.

```
template<typename Container, typename Index>    // Opérationnel mais
auto authAndAccess(Container& c, Index i)       // a besoin d'être
  -> decltype(c[i])                             // affiné.
{
  authenticateUser();
  return c[i];
}
```

La présence de `auto` avant le nom de la fonction n'a aucun rapport avec la déduction de type. Dans ce cas, ce mot clé indique que la syntaxe du « type de retour arrière » (*trailing return type*) de C++11 est utilisée, autrement dit que le type de retour de la fonction sera déclaré après la liste de paramètres (après « `->` ») Grâce au type de retour arrière, il est possible d'utiliser les paramètres de la fonction dans la spécification du type de retour. Par exemple, dans `authAndAccess`, nous spécifions le type de retour en fonction de `c` et de `i`. Si le type de la valeur de retour était placé avant le nom de la fonction, comme cela se fait de façon conventionnelle, `c` et `i` ne pourraient être utilisés car ils ne seraient pas encore déclarés.

Avec cette déclaration, `authAndAccess` retourne le type renvoyé par `operator[]` lorsque cet opérateur est invoqué sur le conteneur passé en argument. Exactement ce que nous souhaitons.

En C++11, les types de retour des expressions lambda à instruction unique peuvent être déduits. C++14 va plus loin en autorisant cela pour toutes les expressions lambda et toutes les fonctions, y compris celles qui sont constituées de plusieurs instructions. Dans le cas de `authAndAccess`, cela signifie que nous pouvons, en C++14, omettre le type de retour arrière, pour ne laisser que le spécificateur `auto` du début. Dans une telle déclaration, l'utilisation de `auto` signifie que la déduction de type aura lieu. Plus précisément, elle signifie que le compilateur déduira le type de retour de la fonction à partir de son implémentation :

```
template<typename Container, typename Index>    // C++14 ;
auto authAndAccess(Container& c, Index i)       // non totalement
{                                               // correct.
  authenticateUser();
  return c[i];              // Type de retour déduit à partir de c[i].
}
```

Le conseil 2 explique que pour les fonctions qui spécifient leur type de retour avec `auto`, le compilateur met en place la déduction de type de template. Dans ce cas, cela pose un problème. Comme nous l'avons indiqué, pour la plupart des conteneurs de `T`, `operator[]` renvoie un `T&`, mais le conseil 1 explique que la déduction de type de template ignore le fait qu'une expression d'initialisation soit une référence. Voyons ce que cela signifie pour le code suivant :

```
std::deque<int> d;
...
authAndAccess(d, 5) = 10;   // Authentifier un utilisateur,
```

Copyright © 2016 Dunod.
© Dunod – Toute reproduction non autorisée est un délit.

```
                                    // retourner d[5],
                                    // puis lui affecter 10.
                                    // Ce code ne compile pas !
```

Dans cet exemple, `d[5]` renvoie un `t&`, mais la déduction de type `auto` sur la valeur de retour de `authAndAccess` omet la référence et lui détermine donc le type `int`. Puisque ce `int` est la valeur de retour d'une fonction, il s'agit d'une rvalue. Le code précédent tente donc d'affecter 10 à une rvalue de type `int`. Cette opération est interdite en C++ et la compilation du code échoue.

Pour que `authAndAccess` fonctionne comme nous le souhaitons, nous devons employer la déduction de type `decltype` pour sa valeur de retour, c'est-à-dire préciser que `authAndAccess` doit renvoyer exactement le même type que celui renvoyé par l'expression `c[i]`. Les gardiens du C++, anticipant le besoin d'utiliser les règles de déduction de type `decltype` dans certains cas d'inférence des types, ont apporté cette possibilité en C++14 au travers du spécificateur `decltype(auto)`. Ce qui pourrait sembler à première vue contradictoire (`decltype` *et* `auto` ?) a en réalité tout son sens : `auto` précise que le type doit être déduit, tandis que `decltype` indique que les règles `decltype` doivent être employées pour la déduction. Voici la nouvelle version de `authAndAccess` :

```
template<typename Container, typename Index>   // C++14 ;
decltype(auto)                                 // opérationnel mais
authAndAccess(Container& c, Index i)           // a encore besoin
{                                              // d'être affiné.
  authenticateUser();
  return c[i];
}
```

`authAndAccess` retourne désormais exactement ce que renvoie `c[i]`. En particulier, dans le cas classique où `c[i]` renvoie un `T&`, `authAndAccess` retourne également un `T&`. Et, dans le cas moins fréquent où `c[i]` retourne un objet, `authAndAccess` retourne aussi un objet.

L'utilisation de `decltype(auto)` ne se limite pas aux types de retour des fonctions. Il est également possible de l'employer dans la déclaration de variables de façon à appliquer les règles de déduction de type `decltype` à l'expression d'initialisation :

```
Widget w;

const Widget& cw = w;

auto myWidget1 = cw;              // Déduction de type auto :
                                  // myWidget1 est de type Widget.

decltype(auto) myWidget2 = cw;    // Déduction de type decltype :
                                  // myWidget2 est de type const Widget&.
```

Nous l'avons mentionné, mais pas encore décrit. Examinons à présent la question du peaufinage de `authAndAccess`. Reprenons la déclaration de la version C++14 de `authAndAccess` :

Copyright © 2016 Dunod.

```
template<typename Container, typename Index>
decltype(auto) authAndAccess(Container& c, Index i);
```

Le conteneur est passé sous forme d'une référence lvalue à un non-`const`, car renvoyer une référence à un élément du conteneur permet à l'appelant de modifier celui-ci. Mais cela signifie qu'il est impossible de passer à cette fonction des conteneurs en rvalue. Les rvalues ne peuvent pas être liées à des références lvalue (excepté pour les références lvalue à un `const`, ce qui n'est pas le cas ici).

Nous devons l'admettre, passer un conteneur en rvalue à `authAndAccess` est un cas limite. Puisqu'un conteneur rvalue est un objet temporaire, il sera détruit à la fin de l'instruction qui contient l'appel à `authAndAccess`. Toute référence à un élément de ce conteneur (ce que renverrait normalement `authAndAccess`) sera donc une référence dans le vide à la fin de l'instruction qui l'a créé. Cela dit, passer un objet temporaire à `authAndAccess` peut avoir un sens. L'appelant pourrait simplement souhaiter faire une copie d'un élément du conteneur temporaire, par exemple :

```
std::deque<std::string> makeStringDeque();   // Fonction fabrique.

// Faire une copie du 5e élément du deque retourné par
// makeStringDeque.
auto s = authAndAccess(makeStringDeque(), 5);
```

Pour qu'une telle utilisation soit possible, nous devons revoir la déclaration de `authAndAccess` afin qu'elle accepte les lvalues et les rvalues. La surcharge est une solution (une version déclarerait un paramètre de référence lvalue, l'autre, un paramètre de référence rvalue), mais nous aurions deux fonctions à maintenir. Pour éviter cela, une approche consiste à déclarer `authAndAccess` avec un paramètre de référence qui peut se lier aux lvalues *et* aux rvalues ; le conseil 24 explique que c'est justement le rôle des références universelles. Voici donc comment déclarer `authAndAccess` :

```
template<typename Container, typename Index>    // c est à présent
decltype(auto) authAndAccess(Container&& c,     // une référence
                             Index i);          // universelle.
```

Dans ce template, nous ne connaissons pas le type de conteneur que nous manipulons et nous désignerons donc également le type des objets qui lui servent d'indices. Lorsque des objets de type inconnu sont passés par valeur, les performances en sont généralement impactées de façon négative, en raison de la copie inutile, du problème de découpage d'objet (*object slicing*, décrit au conseil 41) et des moqueries de nos collègues. Toutefois, dans le cas des indices de conteneurs, suivre le modèle de la bibliothèque standard (par exemple dans `operator[]` pour `std::string`, `std::vector` et `std::deque`) semble plutôt raisonnable. Nous allons donc conserver le passage par valeur.

Copyright © 2016 Dunod.
© Dunod – Toute reproduction non autorisée est un délit.

Nous devons cependant revoir l'implémentation du template afin qu'il tienne compte de l'avertissement concernant l'application de `std::forward` aux références universelles (voir le conseil 25) :

```
template<typename Container, typename Index>        // Version
decltype(auto)                                      // finale
authAndAccess(Container&& c, Index i)               // en C++14.
{
  authenticateUser();
  return std::forward<Container>(c)[i];
}
```

Cette version répond parfaitement à nos besoins, mais elle exige un compilateur C++14. Si vous n'en disposez pas, vous devrez vous tourner vers la version C++11 du template. Elles sont équivalentes, mais il vous faudra préciser le type de retour :

```
template<typename Container, typename Index>        // Version
auto                                                // finale
authAndAccess(Container&& c, Index i)               // en C++11.
-> decltype(std::forward<Container>(c)[i])
{
  authenticateUser();
  return std::forward<Container>(c)[i];
}
```

Vous vous souvenez certainement de notre remarque au début de ce conseil : `decltype` renvoie *presque* toujours le type attendu, ce qui réserve *rarement* des surprises. Il est peu probable que vous rencontriez ces exceptions à la règle, sauf si vous développez des bibliothèques élaborées.

Pour comprendre *parfaitement* le comportement de `decltype`, vous devez vous familiariser avec quelques cas particuliers. La plupart d'entre eux sont trop complexes pour être abordés dans cet ouvrage, mais nous allons en présenter un car cela nous fournira de précieuses informations sur `decltype` et son utilisation.

En appliquant `decltype` à un nom, nous obtenons le type déclaré pour ce nom. Les noms sont des expressions lvalue, mais cela n'affecte pas le comportement de `decltype`. En revanche, lorsque les expressions lvalue sont plus compliquées que de simples noms, `decltype` s'assure que le type renvoyé est toujours une référence lvalue. Autrement dit, si une expression lvalue autre qu'un nom est de type `T`, `decltype` indique qu'elle est de type `T&`. Ce comportement n'a généralement pas d'impact, car le type de la plupart des expressions lvalue comprend un qualificatif de référence lvalue. Par exemple, les fonctions qui retournent des lvalues, renvoient toujours des références lvalue.

Ce fonctionnement a cependant une implication qu'il est bon de connaître. Prenons l'instruction suivante :

```
int x = 0;
```

Puisque `x` est le nom d'une variable, `decltype(x)` renvoie `int`. En plaçant le nom `x` entre parenthèses, « `(x)` », nous obtenons une expression plus complexe. En tant

Copyright © 2016 Dunod.

que nom, `x` est une lvalue et C++ définit l'expression `(x)` comme étant également une lvalue. `decltype((x))` renvoie donc `int&`. Le simple fait de placer des parenthèses autour d'un nom peut modifier le type renvoyé par `decltype` pour ce nom !

En C++11, ce fonctionnement n'est guère qu'une curiosité mais, placé dans le contexte de `decltype(auto)` en C++14, il signifie qu'un changement d'apparence triviale dans la façon d'écrire une instruction `return` peut affecter le type déduit pour une fonction :

```
decltype(auto) f1()
{
  int x = 0;
  …
  return x;      // decltype(x) donne int, donc f1 renvoie un int.
}

decltype(auto) f2()
{
  int x = 0;
  …
  return (x);    // decltype((x)) donne int&, donc f2 renvoie un int&.
}
```

Notez que non seulement `f2` a un type de retour différent de `f1`, mais qu'elle renvoie également une référence à une variable locale ! C'est le genre de code qui conduit à coup sûr vers un comportement indéfini.

En conclusion, vous aurez deviné qu'il faut faire très attention à l'utilisation de `decltype(auto)`. Des détails semble-t-il insignifiants dans l'expression dont nous souhaitons déduire le type peuvent avoir un effet sur le type renvoyé par `decltype(auto)`. Pour être certain que le type déduit soit celui que nous attendons, les techniques décrites au conseil 4 doivent être adoptées.

Cela dit, il faut garder une vision d'ensemble. Si `decltype` (seul ou avec `auto`) peut parfois occasionner des surprises sur la déduction du type, ce ne sera pas le cas dans une situation normale. `decltype` renvoie généralement le type attendu. C'est notamment le cas avec un nom car nous obtenons alors le type déclaré de ce nom.

À retenir

- `decltype` renvoie presque toujours le type d'une variable ou d'une expression sans le modifier.
- Pour les expressions lvalue de type `T` autres que les noms, `decltype` indique toujours un type `T&`.
- C++14 prend en charge `decltype(auto)`, qui, à l'instar de `auto`, déduit le type à partir de son initialiseur, mais la déduction se fait conformément aux règles `decltype`.

Copyright © 2016 Dunod.
© Dunod – Toute reproduction non autorisée est un délit.

CONSEIL N° 4. AFFICHER LES TYPES DÉDUITS

Les outils à employer pour consulter les résultats de la déduction de type dépendent de la phase du processus de développement logiciel dans laquelle nous souhaitons cette information. Nous allons présenter trois contextes d'affichage des informations de déduction de type : pendant la saisie du code, pendant la compilation et pendant l'exécution.

Éditeurs de code

Dans les IDE, les éditeurs de code affichent souvent les types des entités d'un programme (comme des variables, des paramètres, des fonctions, etc.), par exemple en plaçant le curseur de la souris au-dessus d'une entité. Prenons le code suivant :

```
const int theAnswer = 42;

auto x = theAnswer;
auto y = &theAnswer;
```

Un éditeur de code indiquera probablement que le type déduit pour `x` est `int` et `const int*` pour `y`.

Pour que cela fonctionne, le code doit être dans un état plus ou moins compilable, car si l'éditeur peut offrir cette fonctionnalité c'est en raison du compilateur C++ (tout au moins sa partie frontale) qui s'exécute au sein de l'IDE. Si le compilateur n'est pas en mesure de donner un sens à notre code et d'effectuer la déduction de type, il ne pourra pas afficher le type d'une entité.

Dans le cas des types simples, comme `int`, l'information donnée par l'éditeur se révèle en général exacte. Cependant, et nous allons le voir, cette information sera sans doute moins utile lorsqu'elle concerne des types plus complexes.

Diagnostics du compilateur

Pour forcer un compilateur à indiquer le type qu'il a déduit, une solution efficace consiste à employer ce type de façon à déclencher des problèmes de compilation. Le message d'erreur qui décrit le problème affichera à coup sûr le type qui en est à l'origine.

Par exemple, supposons que nous souhaitions connaître les types qui ont été déduits pour les variables `x` et `y` du code précédent. Nous commençons par déclarer un template de classe qui ne les définit pas :

```
template<typename T>        // Déclaration uniquement pour TD ;
class TD;                   // TD == "Type Displayer".
```

Copyright © 2016 Dunod.

Toute tentative d'instanciation de ce template produira un message d'erreur, car il n'y a aucune définition de template à instancier. Pour afficher les types de `x` et de `y`, il suffit d'essayer de créer une instance de `TD` avec leur type :

```
TD<decltype(x)> xType;      // Obtenir des erreurs qui indiquent
TD<decltype(y)> yType;      // le type de x et de y.
```

Nous utilisons des noms de variables de la forme *nomDeLaVariable*`Type`, car ils permettent d'obtenir des messages d'erreur dans lesquels il est plus facile de trouver les informations que nous recherchons. Pour le code précédent, l'un de nos compilateurs a produit les diagnostics suivants (l'information de type qui nous intéresse est repérée) :

```
error: aggregate 'TD<int> xType' has type incomplet and
    cannot be defined
error: aggregate 'TD<const int *> yType' has type incomplet
    and cannot be defined
```

Un autre compilateur apporte la même information, mais sous une forme différente :

```
error: 'xType' uses undefined class 'TD<int>'
error: 'yType' uses undefined class 'TD<const int *>'
```

Si l'on met de côté la question de la présentation, tous les compilateurs que nous avons testés génèrent des messages d'erreur qui contiennent des informations de type utiles.

Affichage à l'exécution

Afficher les informations de type à l'aide de `printf` (non que nous recommandions cette solution) n'est possible qu'au moment de l'exécution, mais elle apporte une plus grande liberté dans le format de la sortie. La difficulté sera de créer une représentation textuelle affichable du type qui nous intéresse. Simple pensez-vous peut-être, il suffit d'appeler `typeid` et `std::type_info::name` à la rescousse. Pour afficher les types déduits pour `x` et `y`, vous pourriez être tenté d'écrire le code suivant :

```
std::cout << typeid(x).name() << '\n';    // Afficher les types
std::cout << typeid(y).name() << '\n';    // de x et de y.
```

Cette approche se fonde sur le fait que l'invocation de `typeid` sur un objet comme `x` ou `y` produit un objet `std::type_info` et que `std::type_info` offre la fonction `name` qui génère une représentation sous forme d'une chaîne de caractères C (c'est-à-dire un `const char*`) du nom du type.

Rien ne garantit que les appels à `std::type_info::name` retourneront un contenu sensé, mais les différentes implémentations font de leur mieux. Leur niveau de réussite varie. Les compilateurs GNU et Clang, par exemple, indiquent que le type de `x`

Copyright © 2016 Dunod.
© Dunod – Toute reproduction non autorisée est un délit.

est « i », et celui de y, « PKi ». Ces résultats trouvent leur sens lorsque l'on sait que, dans la sortie de ces compilateurs, « i » signifie « int » et que « PK » veut dire « pointeur sur ~~konst~~ const ». (Ces deux compilateurs disposent d'un outil, c++filt, pour « décrypter » ces types.) Le compilateur de Microsoft produit une sortie plus claire : « int » pour x et « int const * » pour y.

Puisque les résultats obtenus sont corrects pour les types de x et de y, vous pourriez croire que le problème d'affichage du type est résolu. Ce serait un peu précipité. Prenons un exemple plus complexe :

```
template<typename T>                    // Fonction template
void f(const T& param);                 // à appeler.

std::vector<Widget> createVec();        // Fonction fabrique.


const auto vw = createVec();            // Initialiser vw avec la valeur
                                        // de retour de la fabrique.

if (!vw.empty()) {
  f(&vw[0]);                            // Appeler f.
  …
}
```

Ce code implique un type défini par l'utilisateur (Widget), un conteneur STL (std::vector) et une variable auto (vw). Il est plus représentatif des cas où une visibilité sur les types déduits par les compilateurs nous intéresse. Par exemple, il serait intéressant de connaître les types déterminés pour le paramètre de type T du template et pour le paramètre de fonction param dans f.

Employer typeid pour résoudre le problème n'a rien de compliqué. Il suffit d'ajouter du code à f de façon à afficher les types concernés :

```
template<typename T>
void f(const T& param)
{
  using std::cout;

  cout << "T =     " << typeid(T).name() << '\n';     // Afficher T.

  cout << "param = " << typeid(param).name() << '\n'; // Afficher le
  …                                                  // type de
}                                                    // param.
```

Voici l'affichage produit par l'exécution du code généré par les compilateurs GNU et Clang :

```
T =     PK6Widget
param = PK6Widget
```

Copyright © 2016 Dunod.

Pour ces compilateurs, nous savons déjà que `PK` signifie « pointeur sur `const` ». Il nous reste à comprendre la signification du chiffre `6`. Il s'agit simplement du nombre de caractères présents dans le nom de la classe qui vient ensuite (`Widget`). En résumé, ces compilateurs nous indiquent que `T` et `param` sont de type `const Widget*`.

Le compilateur de Microsoft est d'accord :

```
T =     class Widget const *
param = class Widget const *
```

Trois compilateurs indépendants donnent la même information. Nous pouvons donc penser qu'elle est exacte. Mais voyons cela de plus près. Dans le template `f`, le type déclaré de `param` est `const T&`. Dans ce cas, n'est-il pas étrange que `T` et `param` aient le même type ? Si `T` était un `int`, par exemple, le type de `param` serait `const int&` – un type totalement différent.

Malheureusement, les résultats de `std::type_info::name` ne sont pas fiables. Dans notre exemple, le type affiché par les trois compilateurs pour `param` n'est pas correct. Par ailleurs, il *doit* être incorrect car la spécification de `std::type_info::name` exige que le type soit traité comme s'il était passé par valeur à la fonction template. Nous l'avons expliqué au conseil 1, cela signifie que si le type est une référence, ce fait est ignoré, et si le type obtenu après l'omission de la référence est `const` (ou `volatile`), cette caractéristique est également ignorée. C'est pourquoi le type de `param` (en réalité un `const Widget * const &`) est donné comme étant `const Widget*`. La notion de référence dans le type est tout d'abord retirée, puis le caractère `const` du pointeur résultant est oublié.

Il est également malheureux que l'information de type affichée par les IDE ne soit pas plus fiable, ou tout au moins pas plus utilement fiable. Avec ce même exemple, voici ce qu'un éditeur a affiché comme type pour `T` :

```
const
std::_Simple_types<std::_Wrap_alloc<std::_Vec_base_types<Widget,
std::allocator<Widget> >::_Alloc>::value_type>::value_type *
```

Et voici ce qu'il a donné pour `param` :

```
const std::_Simple_types<...>::value_type *const &
```

Le résultat est moins intimidant que pour le type de `T`, mais les « ... » au milieu le rendent confus, jusqu'à ce que nous comprenions que l'IDE indique par-là « j'ai omis toute la partie qui concerne le type de `T` ». Avec un peu de chance, votre environnement de développement donnera de meilleurs résultats.

Si vous préférez faire confiance aux bibliothèques plutôt qu'à la chance, vous serez content d'apprendre que là où `std::type_info::name` et les IDE peuvent échouer, la bibliothèque Boost TypeIndex (souvent nommée *Boost.TypeIndex*) a été conçue pour réussir. Elle ne fait pas partie du C++ standard, mais pas plus que les IDE ou les templates comme `TD`. Par ailleurs, le fait que les bibliothèques de Boost (disponibles

Copyright © 2016 Dunod.
© Dunod – Toute reproduction non autorisée est un délit.

sur le site *boost.com*) soient multiplates-formes, open-source et proposées avec une licence qui satisferait même le service juridique le plus pointilleux, signifie que le code qui les utilise est quasiment aussi portable que celui qui se fonde sur la bibliothèque standard.

Voici comment notre fonction `f` peut afficher une information de type précise en utilisant la bibliothèque Boost.TypeIndex :

```
#include <boost/type_index.hpp>

template<typename T>
void f(const T& param)
{
  using std::cout;
  using boost::typeindex::type_id_with_cvr;

  // Afficher T.
  cout << "T =     "
       << type_id_with_cvr<T>().pretty_name()
       << '\n';

  // Afficher le type de param.
  cout << "param = "
       << type_id_with_cvr<decltype(param)>().pretty_name()
       << '\n';
  …
}
```

Étudions le fonctionnement de cet exemple. Le template de fonction `boost::typeindex::type_id_with_cvr` prend un argument de type (celui à propos duquel nous souhaitons des informations) et ne retire *pas* les qualificatifs `const`, `volatile` ou de référence (d'où la partie « `with_cvr` » dans le nom du template). Le résultat est un objet `boost::typeindex::type_index`, dont la fonction membre `pretty_name` génère un `std::string` qui contient une représentation lisible du type.

Avec cette implémentation de `f`, examinons de nouveau l'appel qui a donné une information de type incorrect pour `param` car fondée sur `typeid` :

```
std::vector<Widget> createVec();      // Fonction fabrique.

const auto vw = createVec();          // Initialiser vw avec la valeur
                                      // de retour de la fabrique.

if (!vw.empty()) {
  f(&vw[0]);                          // Appeler f.
  …
}
```

Copyright © 2016 Dunod.

Avec les compilateurs GNU et Clang, Boost.TypeIndex affiche le résultat (juste) suivant :

```
T =     Widget const*
param = Widget const* const&
```

Celui obtenu avec les compilateurs de Microsoft est quasi identique :

```
T =     class Widget const *
param = class Widget const * const &
```

Une telle uniformité est appréciable, mais il est important de ne pas oublier que les IDE, les messages d'erreur du compilateur et les bibliothèques comme Boost.TypeIndex ne sont que des outils qui aideront à déterminer les types déduits par les compilateurs. Ils seront tous utiles, mais rien ne saura remplacer une parfaite compréhension des informations de déduction de type données dans les conseils 1 à 3.

À retenir

- Il est souvent possible d'afficher les types déduits en utilisant les IDE, les messages d'erreur du compilateur et la bibliothèque Boost.TypeIndex.
- Les résultats obtenus avec certains outils risquent d'être ni utiles ni précis. Il est donc indispensable de comprendre les règles de déduction de type de C++.

Copyright © 2016 Dunod.

2

auto

En soi, `auto` est aussi simple qu'il puisse l'être, mais il est aussi plus subtil qu'il ne le paraît. Certes, il permet de faire des économies de saisie, mais il évite également les problèmes d'exactitude et de performances qui peuvent tourmenter les déclarations de type manuelles. Par ailleurs, certains des résultats de la déduction de type `auto`, bien que scrupuleusement conformes à l'algorithme prévu, sont, du point de vue du programmeur, tout simplement faux. Lorsque c'est le cas, il est important de savoir orienter `auto` vers la bonne réponse, car revenir à des déclarations de type manuelles est une solution qu'il est souvent préférable d'éviter.

Ce court chapitre fait le tour des avantages et des inconvénients de `auto`.

CONSEIL N° 5. PRÉFÉRER `AUTO` AUX DÉCLARATIONS DE TYPES EXPLICITES

Pourquoi faire compliqué quand on peut faire simple ?

```
int x;
```

Mince, nous avons oublié d'initialiser `x`. Sa valeur est donc indéterminée. La variable peut avoir été fixée à zéro, mais cela dépend du contexte. Zut !

Peu importe, passons au plaisir simple de la déclaration d'une variable locale initialisée en déréférençant un itérateur :

```
template<typename It>    // Algorithme pour appliquer un traitement
void dwim(It b, It e)    // (dwim, pour "do what I mean") à tous
{                        // les éléments entre b et e.
```

Copyright © 2016 Dunod.
© Dunod – Toute reproduction non autorisée est un délit.

```
  while (b != e) {
    typename std::iterator_traits<It>::value_type
      currValue = *b;
    …
  }
}
```

Horrible ! « `typename std::iterator_traits<It>::value_type` » pour exprimer le type de la valeur pointée par un itérateur ? Est-ce bien vrai ? Nous avons dû refouler le souvenir du plaisir que procurait ce type d'expression. Nous n'avons quand même pas écrit cela !

Quel bonheur simple de déclarer une variable locale dont le type est celui d'une fermeture… D'accord, le type d'une fermeture n'est connu que du compilateur et ne peut donc pas être écrit. Mince !

Zut alors, la programmation en C++ n'est pas aussi agréable qu'elle le devrait !

L'a-t-elle vraiment été ? Quoi qu'il en soit, depuis C++11 et l'introduction de `auto`, toutes ces questions ne se posent plus. Puisque le type d'une variable `auto` est déduit à partir de son initialiseur, elle doit être initialisée. Autrement dit, dès lors que nous optons pour du C++ moderne, nous pouvons dire adieu à tout un ensemble de problèmes liés aux variables non initialisées :

```
int x1;                    // Potentiellement non initialisée.

auto x2;                   // Erreur ! L'initialiseur est obligatoire.

auto x3 = 0;               // Parfait, la valeur de x est fixée.
```

Nous évitons même les problèmes associés à la déclaration d'une variable locale dont la valeur est celle d'un itérateur déréférencé :

```
template<typename It>      // Comme précédemment.
void dwim(It b, It e)
{
  while (b != e) {
    auto currValue = *b;
    …
  }
}
```

Et, puisque `auto` se fonde sur la déduction de type (voir le conseil 2), il est possible de représenter des types connus uniquement du compilateur :

```
auto derefUPLess =                          // Fonction de comparaison
  [](const std::unique_ptr<Widget>& p1,     // de Widget
     const std::unique_ptr<Widget>& p2)     // désignés par
  { return *p1 < *p2; };                    // des std::unique_ptr.
```

Plutôt intéressant ! C++14 va encore plus loin, car les paramètres des expressions lambda peuvent être définis avec `auto` :

Copyright © 2016 Dunod.

```
auto derefLess =                        // C++14 : fonction de comparaison
  [](const auto& p1,                    // de valeurs quelconques
     const auto& p2)                    // désignées par ce qui
  { return *p1 < *p2; };                // ressemble à des pointeurs.
```

Malgré l'intérêt de cette solution, vous pensez peut-être que nous n'avons pas vraiment besoin de `auto` pour déclarer une variable qui contient une fermeture car nous pouvons employer un objet `std::function`. C'est exact, mais ce n'est probablement pas ce à quoi vous pensiez. Vous vous demandez sans doute à présent ce qu'est un objet `std::function`.

`std::function` est fourni par la bibliothèque standard de C++11 et ce template a pour objectif de généraliser la notion de pointeur de fonction. Cependant, alors que les pointeurs de fonctions ne peuvent désigner que des fonctions, les objets `std::function` peuvent faire référence à n'importe quel objet invocable, c'est-à-dire tout ce qui peut être invoqué comme une fonction. Lors de la création d'un pointeur de fonction, nous devons spécifier le type de la fonction pointée (c'est-à-dire la signature des fonctions sur lesquelles nous voulons pointer). De la même manière, lors de la création d'un objet `std::function`, nous devons préciser le type de la fonction référencée. Cela se fait au travers du paramètre de template de `std::function`. Par exemple, pour déclarer un objet `std::function` nommé `func` qui peut faire référence à tout objet invocable qui aurait la signature suivante :

```
bool(const std::unique_ptr<Widget>&,  // Signature C++11 pour
     const std::unique_ptr<Widget>&)  // une fonction de comparaison
                                      // de std::unique_ptr<Widget>.
```

il suffit d'écrire ce code :

```
std::function<bool(const std::unique_ptr<Widget>&,
                   const std::unique_ptr<Widget>&)> func;
```

Puisque les expressions lambda impliquent des objets invocables, des fermetures peuvent être placées dans des objets `std::function`. Nous pouvons donc déclarer la version C++11 de `derefUPLess` sans utiliser `auto` :

```
std::function<bool(const std::unique_ptr<Widget>&,
                   const std::unique_ptr<Widget>&)>
  derefUPLess = [](const std::unique_ptr<Widget>& p1,
                   const std::unique_ptr<Widget>& p2)
                  { return *p1 < *p2; };
```

Sans tenir compte de la verbosité syntaxique et de la nécessité de répéter les paramètres de type, l'utilisation de `std::function` n'est pas identique à celle de `auto`. Une variable déclarée `auto` qui contient une fermeture a le même type que cette fermeture et n'utilise donc que la quantité de mémoire requise par la fermeture. Le type d'une variable déclarée `std::function` et contenant une fermeture est une instanciation du template `std::function` dont la taille est figée quelle que soit la

Copyright © 2016 Dunod.
© Dunod – Toute reproduction non autorisée est un délit.

signature. Cette taille ne correspondra peut-être pas aux besoins de la fermeture à stocker. Dans ce cas, le constructeur de `std::function` allouera une zone de mémoire sur le tas pour y placer la fermeture. Par conséquent, l'objet `std::function` consomme en général une quantité de mémoire plus importante que l'objet déclaré avec `auto`. Et, en raison des détails d'implémentation qui restreignent les fonctions `inline` et conduisent à des appels de fonctions indirects, l'invocation d'une fermeture au travers d'un objet `std::function` sera à coup sûr plus lente que son invocation au travers d'un objet déclaré avec `auto`. Autrement dit, la solution `std::function` est en général plus coûteuse et plus lente que l'approche `auto`, sans compter qu'elle peut mener à des exceptions de dépassement de mémoire. Par ailleurs, nous l'avons vu dans les exemples précédents, il est plus simple de saisir « `auto` » que d'écrire le type de l'instanciation de `std::function`. Dans le match entre `auto` et `std::function` pour le stockage d'une fermeture, le grand gagnant est sans conteste `auto`. (Un argument comparable peut être développé lors du choix entre `auto` et `std::function` pour stocker le résultat des appels à `std::bind`, mais, au conseil 34, nous ferons tout pour vous convaincre d'utiliser les expressions lambda à la place de `std::bind`.)

Les avantages de `auto` ne se limitent pas à éviter les variables non initialisées, les déclarations verbeuses de variables et la possibilité de contenir directement des fermetures. Il faut y ajouter l'élimination des problèmes que nous disons associés aux « raccourcis de types ». Voici un code que vous avez probablement déjà vu, voire écrit vous-même :

```
std::vector<int> v;
…
unsigned sz = v.size();
```

Officiellement, le type de la valeur de retour de `v.size()` est `std::vector<int>::size_type`, mais peu de développeurs le savent. La spécification de `std::vector<int>::size_type` stipule qu'il s'agit d'un type entier non signé et de nombreux programmeurs pensent que `unsigned` suffit et écrivent donc du code comparable au précédent. Les conséquences peuvent être assez intéressantes. Par exemple, sur un système Windows 32 bits, `unsigned` et `std::vector<int>::size_type` ont la même taille, mais, sur la version 64 bits, un `unsigned` occupe 32 bits tandis qu'un `std::vector<int>::size_type` demande 64 bits. Autrement dit, un code opérationnel sur un système Windows 32 bits peut afficher un comportement incorrect sur un système Windows 64 bits. Par ailleurs, en cas de portage de l'application d'un système 32 bits vers un système 64 bits, qui souhaitera passer du temps sur de tels problèmes ?

Grâce à `auto`, toutes ces interrogations disparaissent :

```
auto sz = v.size();  // sz est de type std::vector<int>::size_type.
```

Si vous n'êtes toujours pas convaincu de l'intérêt d'utiliser `auto`, étudions le code suivant :

Copyright © 2016 Dunod.

```
std::unordered_map<std::string, int> m;
…

for (const std::pair<std::string, int>& p : m)
{
  …                    // Utiliser p.
}
```

Il semble parfaitement raisonnable, mais il cache un problème. L'avez-vous trouvé ?

Pour trouver ce qui ne va pas, il faut se rappeler que l'élément essentiel d'un `std::unordered_map` est `const`. Par conséquent, le type de `std::pair` dans la table de hachage (ce qu'est réellement `std::unordered_map`) est non pas `std::pair<std::string, int>`, mais `std::pair<const std::string, int>`. Pourtant, ce n'est pas le type déclaré pour la variable `p` dans la boucle. Le compilateur va donc se débrouiller pour trouver une façon de convertir des objets `std::pair<const std::string, int>` (ce que contient la table de hachage) en objets `std::pair<std::string, int>` (le type déclaré pour `p`). Pour cela, il va créer un objet temporaire du type auquel `p` veut se lier en copiant chaque objet de `m`, puis il va lier la référence `p` à cet objet temporaire. Celui-ci sera détruit à la fin de chaque itération de la boucle. Il est probable que vous soyez surpris par ce comportement car, en écrivant cette boucle, l'idée était certainement de lier simplement la référence `p` à chaque élément de `m`.

Les erreurs de ce genre peuvent être évitées grâce à `auto` :

```
for (const auto& p : m)
{
  …                                  // Comme précédemment.
}
```

Non seulement ce code est plus efficace, mais il est également plus facile à saisir, sans oublier qu'il présente un autre intérêt. Si nous prenons l'adresse de `p`, nous sommes certains d'obtenir un pointeur sur un élément dans `m`. Avec le code qui n'utilise pas `auto`, nous obtenons un pointeur sur un objet temporaire, qui sera détruit à la fin de chaque itération de la boucle.

Ces deux derniers exemples, écrire `unsigned` alors qu'il aurait fallu écrire `std::vector<int>::size_type` et écrire `std::pair<std::string, int>` à la place de `std::pair<const std::string, int>`, montrent bien qu'une spécification explicite des types peut conduire à des conversions implicites, ni voulues, ni attendues. En utilisant le type `auto` sur la variable cible, nous n'avons plus à nous inquiéter des incohérences entre le type déclaré de la variable et celui de l'expression d'initialisation.

Les raisons de préférer `auto` à des déclarations de type explicites sont donc nombreuses. Pourtant, `auto` n'est pas parfait. Le type d'une variable `auto` est déduit de son expression d'initialisation et certaines de ces expressions ont des types qui sont ni anticipés ni souhaités. Les cas où cela se produit et la façon de les résoudre font l'objet des conseils 2 et 6. Nous ne les aborderons donc pas ici et, à la place, nous

Copyright © 2016 Dunod.
© Dunod – Toute reproduction non autorisée est un délit.

allons nous intéresser à un autre problème de l'utilisation de `auto` en remplacement des déclarations de type classiques : la lisibilité du code source résultant.

Soyez tranquille, `auto` est une option, non une obligation. Si, selon votre jugement professionnel, votre code est plus clair et plus facile à maintenir, ou meilleur de tout autre manière, en déclarant explicitement les types, vous pouvez poursuivre sur cette voie. Mais n'oubliez pas que C++ n'innove pas en adoptant ce qui est généralement connu dans le monde des langages de programmation sous le terme *inférence de type*. D'autres langages procéduraux à typage statique (comme C#, D, Scala et Visual Basic) proposent une fonctionnalité plus ou moins équivalente, sans parler des langages fonctionnels à typage statique (comme ML, Haskell, OCaml, F#, etc.). Cette généralisation est en partie due au succès des langages à typage dynamique, comme Perl, Python et Ruby, dans lesquels les variables sont rarement typées de façon explicite. La communauté du développement logiciel possède une longue expérience de l'inférence de type et il a été démontré que cette technologie n'est en aucun cas contraire à la création et à la maintenance de larges bases de code de qualité industrielle.

Certains développeurs sont perturbés par le fait que l'utilisation de `auto` retire toute possibilité de déterminer le type d'un objet par un simple examen du code source. Cependant, la capacité des IDE à afficher le type d'un objet tend à atténuer ce problème (même en tenant compte des difficultés mentionnées au conseil 4) et, dans de nombreux cas, une vue quelque peu abstraite du type d'un objet est aussi utile qu'une connaissance exacte. Par exemple, il suffit souvent de savoir qu'un objet est un conteneur, un compteur ou un pointeur intelligent, sans qu'il soit indispensable de connaître précisément le type de conteneur, de compteur ou de pointeur intelligent. Si les noms des variables sont bien choisis, cette information de type abstraite devrait presque toujours être disponible.

Le fait est qu'écrire explicitement les types n'apporte souvent rien d'autre qu'une opportunité d'introduire des erreurs subtiles, que ce soit sur le plan de l'exactitude ou de l'efficacité, si ce n'est les deux. Par ailleurs, les types `auto` s'adaptent automatiquement lorsque le type de l'expression d'initialisation change. Autrement dit, grâce à `auto`, le remaniement du code est plus facile. Par exemple, si une fonction déclare retourner un `int` et si nous décidons ultérieurement qu'un `long` conviendrait mieux, le code appelant se mettra automatiquement à jour lors de la compilation suivante, à condition que les résultats de l'appel de la fonction soient mémorisés dans des variables `auto`. En plaçant les résultats dans des variables déclarées explicitement comme des `int`, nous aurons à rechercher tous les appels de façon à les corriger.

À retenir

- Les variables `auto` doivent être initialisées, sont généralement immunisées contre les erreurs de typage qui peuvent mener à des problèmes de portabilité ou d'efficacité, peuvent faciliter le remaniement du code et demandent en général une saisie moindre que les types spécifiés explicitement.
- Les variables `auto` sont sujettes aux risques décrits dans les conseils 2 et 6.

Copyright © 2016 Dunod.

CONSEIL N° 6. OPTER POUR UN INITIALISEUR AU TYPE EXPLICITE LORSQUE AUTO DÉDUIT DES TYPES NON SOUHAITÉS

Le conseil 5 explique que la déclaration de variables avec `auto` apporte plusieurs avantages techniques par rapport à la spécification explicite des types. Malheureusement, la déduction de type `auto` va parfois à gauche alors que nous voudrions qu'elle aille à droite. Supposons, par exemple, que nous ayons une fonction qui prend un `Widget` en paramètre et retourne un `std::vector<bool>`, dans lequel chaque `bool` indique si le `Widget` offre une certaine fonction :

```
std::vector<bool> features(const Widget& w);
```

Supposons également que le bit 5 indique si le `Widget` dispose d'une priorité élevée. Nous pouvons alors écrire un code comparable au suivant :

```
Widget w;
...

bool highPriority = features(w)[5]; // La priorité de w est-elle haute ?
...

processWidget(w, highPriority);     // Manipuler w en fonction de
                                    // sa priorité.
```

Ce code est correct et fonctionnera parfaitement. Mais apportons un changement d'apparence inoffensive en remplaçant le type explicite de `highPriority` par `auto` :

```
auto highPriority = features(w)[5];  // La priorité de w est-elle
                                     // haute ?
```

Dans ce cas, la situation change. Le code reste compilable, mais son comportement devient incertain :

```
processWidget(w, highPriority);      // Comportement indéfini !
```

Nous l'indiquons dans le commentaire, l'appel à `processWidget` affiche à présent un comportement indéfini. Mais pour quelle raison ? La réponse risque de vous surprendre. Avec `auto`, `highPriority` n'est plus de type `bool`. Même si `std::vector<bool>` contient conceptuellement des `bool`, la fonction `operator[]` pour `std::vector<bool>` ne retourne pas une référence à un élément du conteneur (`std::vector::operator[]` retourne bien une référence, mais pour tous les types *autres* que `bool`). À la place, elle retourne un objet de type `std::vector<bool>::reference` (une classe imbriquée dans `std::vector<bool>`).

La classe `std::vector<bool>::reference` existe car `std::vector<bool>` doit représenter ses éléments `bool` sous forme compacte, un bit par `bool`. Cela pose un problème

Copyright © 2016 Dunod.
© Dunod – Toute reproduction non autorisée est un délit.

à la fonction operator[] de std::vector<bool>, car la fonction operator[] de std::vector<T> est supposée retourner un T&, alors que les références à des bits sont interdites en C++. Puisqu'elle n'est pas en mesure de retourner un bool&, la fonction operator[] de std::vector<bool> renvoie un objet qui « sert » de bool&. Pour que cela fonctionne, des objets std::vector<bool>::reference doivent pouvoir être employés dans les mêmes contextes que des bool&. Parmi tous les éléments de std::vector<bool>::reference qui participent à ce bon fonctionnement, nous trouvons une conversion implicite en bool. (Non en bool&, mais en *bool*. Expliquer l'ensemble des techniques employées par std::vector<bool>::reference pour simuler le comportement d'un bool& nous éloignerait trop de notre propos. Signalons simplement que cette conversion implicite n'est qu'un élément d'un ensemble plus vaste.)

En gardant cette information à l'esprit, examinons à nouveau cette partie du code d'origine :

```
bool highPriority = features(w)[5];  // Déclarer explicitement
                                     // le type de highPriority.
```

features renvoie ici un objet std::vector<bool>, sur lequel operator[] est invoqué. operator[] retourne un objet std::vector<bool>::reference, qui est ensuite converti implicitement en un bool de façon à initialiser highPriority. Cette variable prend alors la valeur du bit 5 qui provient du std::vector<bool> renvoyé par features, exactement comme nous le voulons.

Comparons ce fonctionnement à celui obtenu avec une déclaration auto pour highPriority :

```
auto highPriority = features(w)[5];  // Déduire le type de
                                     // highPriority.
```

À nouveau, features renvoie un objet std::vector<bool> sur lequel operator[] est invoqué. operator[] retourne encore un objet std::vector<bool>::reference, mais, cette fois-ci, auto en fait le type de highPriority par déduction. highPriority n'a pas du tout la valeur du bit 5 du std::vector<bool> retourné par features.

La valeur qui lui est attribuée dépend de l'implémentation de std::vector<bool>::reference. Dans une version, un tel objet contient un pointeur sur le mot *machine* qui comprend le bit référencé, ainsi que le décalage du bit dans ce mot. Voyons ce que cette implémentation de std::vector<bool>::reference signifie pour l'initialisation de highPriority.

L'appel à features renvoie un objet std::vector<bool> temporaire. Cet objet n'a pas de nom mais, pour faciliter les explications, appelons-le *temp*. operator[] est invoqué sur *temp* et l'objet std::vector<bool>::reference renvoyé comprend un pointeur sur un mot dans la structure de données qui contient les bits gérés par *temp*, ainsi que le décalage qui correspond au bit 5 dans ce mot. highPriority est une copie de cet objet std::vector<bool>::reference et comprend donc un pointeur sur un mot dans *temp*, ainsi que le décalage qui correspond au bit 5. Puisque *temp* est un objet temporaire, il est détruit à la fin de l'instruction. Par conséquent, highPriority

Copyright © 2016 Dunod.

contient un pointeur dans le vide et c'est là la raison du comportement indéfini dans l'appel à `processWidget` :

```
processWidget(w, highPriority);    // Comportement indéfini !
                                   // highPriority contient un
                                   // pointeur dans le vide !
```

`std::vector<bool>::reference` est un exemple de *classe proxy*, c'est-à-dire une classe conçue pour émuler et étendre le comportement d'un autre type. Les classes proxy sont employées à divers objectifs. Par exemple, `std::vector<bool>::reference` offre l'illusion que la fonction `operator[]` de `std::vector<bool>` retourne une référence sur un bit. De manière comparable, les types de pointeurs intelligents de la bibliothèque standard (voir le chapitre 4) sont des classes proxy qui greffent la gestion de ressources sur des pointeurs bruts. L'intérêt des classes proxy est reconnu. Le design pattern Proxy est même l'un des plus anciens membres du Panthéon des design patterns logiciels.

Certaines classes proxy sont faites pour être vues par les clients. C'est notamment le cas de `std::shared_ptr` et de `std::unique_ptr`. D'autres agissent de manière plus ou moins visible, comme `std::vector<bool>::reference` et son homologue `std::bitset::reference` pour `std::bitset`.

Certaines classes des bibliothèques C++ appartiennent également à ce groupe et emploient une technique appelée *templates d'expression*. Ces bibliothèques ont été développées à l'origine dans le but d'améliorer l'efficacité des calculs numériques. Par exemple, étant donné une classe `Matrix` et les objets `m1`, `m2`, `m3` et `m4` de cette classe, l'expression

```
Matrix sum = m1 + m2 + m3 + m4;
```

peut être calculée plus efficacement si la fonction `operator+` de `Matrix` renvoie un proxy du résultat à la place du résultat lui-même. Autrement dit, `operator+` pour deux objets `Matrix` retournera un objet d'une classe proxy comme `Sum<Matrix, Matrix>` à la place d'un objet `Matrix`. Comme c'était le cas avec `std::vector<bool>::reference` et `bool`, il existe une conversion implicite de la classe proxy vers `Matrix`, ce qui permet d'initialiser `sum` à partir de l'objet proxy généré par l'expression placée du côté droit du signe « = ». (Le type de cet objet encode habituellement l'intégralité de l'expression d'initialisation, c'est-à-dire quelque chose comme `Sum<Sum<Sum<Matrix, Matrix>, Matrix>, Matrix>`. Vous comprenez pourquoi il est préférable de masquer ce type au code client.)

De façon générale, les classes proxy « invisibles » s'entendent mal avec `auto`. Les instances de ces classes sont rarement conçues pour vivre au-delà d'une seule instruction. Par conséquent, la création de variables de ces types va à l'encontre des hypothèses fondamentales de la conception d'une bibliothèque. C'est le cas avec `std::vector<bool>::reference` et nous avons vu que ne pas respecter ces hypothèses peut conduire à un comportement indéfini.

Il vaut donc mieux éviter la forme de code suivante :

Copyright © 2016 Dunod.
© Dunod – Toute reproduction non autorisée est un délit.

```
auto someVar = expression de type d'une classe proxy "invisible";
```

La question est donc de savoir si des objets proxy sont employés. Le logiciel qui les utilise n'annoncera probablement pas leur existence. Ils sont supposés être *invisibles*, tout au moins conceptuellement ! Et si nous les trouvons, devons-nous réellement abandonner `auto` et ses nombreux avantages décrits au conseil 5 ?

Commençons par voir comment les trouver. Bien que les classes proxy « invisibles » soient conçues pour voler sous la couverture radar du programmeur dans une utilisation normale, les bibliothèques qui les utilisent les mentionnent souvent dans leur documentation. Plus nous sommes familiers des choix de conception des bibliothèques que nous utilisons, moins nous risquons d'être pris de court par l'emploi de proxy dans ces bibliothèques.

Si la documentation manque de détails, les fichiers d'en-tête combleront ses lacunes. Le code source aura beaucoup de mal à dissimuler totalement les objets proxy. Ils sont habituellement retournés par des fonctions que les clients sont supposés appeler et les signatures de ces fonctions révèlent donc leur existence. Voici par exemple la définition de `std::vector<bool>::operator[]` :

```
namespace std {              // Tiré de la bibliothèque standard de C++.

template <class Allocator>
  class vector<bool, Allocator> {
  public:
    …
    class reference { … };

    reference operator[](size_type n);
    …
  };
}
```

En supposant que nous sachions que la fonction `operator[]` pour `std::vector<T>` renvoie normalement un `T&`, le type de retour inhabituel pour `operator[]` dans ce cas constitue un indice de l'utilisation d'une classe proxy. En examinant soigneusement les interfaces que nous utilisons, nous pouvons souvent découvrir l'existence de classes proxy.

Dans la pratique, de nombreux développeurs découvrent l'utilisation de classes proxy uniquement lorsqu'ils tentent de comprendre les problèmes de compilation ou de déboguer des résultats erronés obtenus lors des tests unitaires. Quelle que soit la manière de les trouver, dès lors que `auto` déduit le type d'une classe proxy à la place du type qu'elle représente, la solution n'est pas d'abandonner `auto` car il n'est pas en cause. Le problème vient du fait que `auto` ne déduit pas le type que nous attendons. La solution est donc d'obliger une autre déduction de type. Pour cela, nous allons utiliser l'idiome de l'*initialiseur au type explicite*.

Cet idiome implique la déclaration d'une variable avec `auto` et le forçage du type de l'expression d'initialisation à celui qui doit être déduit par `auto`. Par exemple, voici comment le mettre en place pour imposer à `highPriority` le type `bool` :

Copyright © 2016 Dunod.

```
auto highPriority = static_cast<bool>(features(w)[5]);
```

`features(w)[5]` retourne encore un objet `std::vector<bool>::reference`, comme cela a toujours été le cas, mais la conversion de type change celui de l'expression en `bool`. `auto` attribue donc ce type à `highPriority`. Au moment de l'exécution, l'objet `std::vector<bool>::reference` renvoyé par `std::vector<bool>::operator[]` effectue sa conversion vers `bool`, au cours de laquelle le pointeur toujours valide sur le `std::vector<bool>` renvoyé par `features` est déréférencé. Nous évitons ainsi le comportement indéfini précédent. L'indice 5 est ensuite appliqué aux bits ciblés par le pointeur et la valeur `bool` qui en résulte sert à initialiser `highPriority`.

Voici comment employer l'idiome de l'initialiseur au type explicite dans le cas de `Matrix` :

```
auto sum = static_cast<Matrix>(m1 + m2 + m3 + m4);
```

L'usage de cet idiome ne se limite pas aux initialiseurs conduisant à des types qui correspondent à des classes proxy. Il peut également se révéler utile pour indiquer la création délibérée d'une variable dont le type diffère de celui généré par l'expression d'initialisation. Supposons, par exemple, que nous ayons une fonction de calcul d'une valeur de tolérance :

```
double calcEpsilon();           // Retourner une valeur de tolérance.
```

`calcEpsilon` retourne clairement un `double`, mais supposons que nous sachions que, dans notre application, la précision d'un `float` convient et que la différence de taille entre les `float` et les `double` revêt une importance. Nous pouvons déclarer une variable `float` pour contenir le résultat fourni par `calcEpsilon` :

```
float ep = calcEpsilon();       // Conversion implicite
                                // de double vers float.
```

Mais cette solution n'annonce pas clairement que nous diminuons volontairement la précision de la valeur retournée par la fonction. En revanche, une déclaration fondée sur l'idiome de l'initialiseur au type explicite transmet ce message :

```
auto ep = static_cast<float>(calcEpsilon());
```

Un raisonnement comparable peut être tenu lorsqu'une expression en virgule flottante est mémorisée sous forme de valeur entière. Supposons que nous voulions calculer l'indice d'un élément dans un conteneur à l'aide d'itérateurs à accès aléatoire (par exemple `std::vector`, `std::deque` ou `std::array`) et que nous recevions une valeur `double` dans la plage `0.0` à `1.0` pour indiquer la place de l'élément souhaité dans le conteneur (`0.5` représente le milieu du conteneur). Supposons de plus que nous soyons certains que l'indice résultant tiendra dans un `int`. Si le conteneur est `c` et si la valeur est `d`, nous pouvons calculer l'indice de la manière suivante :

Copyright © 2016 Dunod.
© Dunod – Toute reproduction non autorisée est un délit.

```
int index = d * c.size();
```

Cependant, cette solution n'indique pas clairement que la conversion en `int` de la valeur `double` placée à droite est intentionnelle. Grâce à l'idiome de l'initialiseur au type explicite, les choses deviennent transparentes :

```
auto index = static_cast<int>(d * c.size());
```

À retenir

- Les types proxy « invisibles » peuvent conduire `auto` à la déduction du « mauvais » type pour une expression d'initialisation.
- L'idiome de l'initialiseur au type explicite force `auto` à déduire le type qui est attendu.

Copyright © 2016 Dunod.

3

Vers un C++ moderne

Sur le plan des fonctionnalités aux noms pompeux, C++11 et C++14 ne sont pas en reste : `auto`, pointeurs intelligents, sémantique de déplacement, expressions lambda, concurrence, pour ne citer qu'elles. Mais elles sont si importantes qu'un chapitre est consacré à chacune d'elles et il est indispensable de les maîtriser. Toutefois, devenir un véritable programmeur en C++ moderne passe également par une suite de petites étapes. Chacune répond à des questions précises, qui se posent lors du passage du C++98 au C++ moderne. Quand devons-nous remplacer les parenthèses par des accolades lors de la création d'un objet ? Pourquoi les déclarations d'alias doivent être préférées aux `typedef` ? Quelles sont les différences entre `constexpr` et `const` ? Quel est le rapport entre les fonctions membres `const` et la sûreté vis-à-vis des threads ? Nous pourrions continuer cette liste encore longtemps. Une par une, ce chapitre fournira les réponses.

CONSEIL N° 7. DIFFÉRENCIER `()` ET `{}` LORS DE LA CRÉATION DES OBJETS

Selon le point de vue, les différentes syntaxes proposées par C++11 pour l'initialisation d'un objet représentent soit une abondance de biens, soit une source de désordre. De façon générale, les valeurs d'initialisation peuvent être indiquées avec des parenthèses, un signe égal ou des accolades :

```
int x(0);              // Initialiseur entre parenthèses.

int y = 0;             // Initialiseur après le signe "=".

int z{ 0 };            // Initialiseur entre accolades.
```

© Dunod – Toute reproduction non autorisée est un délit.
Copyright © 2016 Dunod.

Dans de nombreux cas, il est également possible d'utiliser un signe égal et des accolades :

```
int z = { 0 };      // Initialiseur avec un signe "=" et des accolades.
```

Dans la suite de ce conseil, nous ignorerons généralement cette dernière possibilité car C++ la traite habituellement de la même manière que celle qui implique uniquement des accolades.

Ceux qui pensent qu'il s'agit d'une « source de désordre » soulignent que l'initialisation avec un signe égal induit souvent en erreur les débutants, car ils pensent, à tort, y voir une affectation. Pour les types intégrés, comme `int`, la différence est théorique mais, pour les types définis par l'utilisateur, il est important de distinguer initialisation et affectation car elles font appel à des fonctions différentes :

```
Widget w1;            // Appeler le constructeur par défaut.

Widget w2 = w1;       // Pas d'affectation ; appel du constructeur de
                      // copie.

w1 = w2;              // Affectation ; appel de l'opérateur = de
                      // copie.
```

Malgré les différentes syntaxes d'initialisation, il existe des cas où C++98 ne permettait pas d'exprimer l'initialisation souhaitée. Par exemple, il n'était pas possible d'indiquer directement qu'un conteneur STL devait être créé pour stocker un ensemble précis de valeurs (par exemple 1, 3 et 5).

Pour résoudre le problème de confusion des multiples syntaxes d'initialisation, ainsi que l'impossibilité de prendre en charge tous les scénarios d'initialisation, C++11 apporte l'*initialisation uniforme*. Il s'agit d'une syntaxe d'initialisation unique qui, tout au moins dans le concept, peut être utilisée partout et peut tout exprimer. Elle se fonde sur les accolades et c'est pourquoi nous préférons l'expression *initialisation à accolades*. L'« initialisation uniforme » est un concept, l'« initialisation à accolades » est une construction syntaxique.

L'initialisation à accolades nous permet d'exprimer ce qui était auparavant inexprimable. Grâce à cette syntaxe, il est facile de préciser le contenu initial d'un conteneur :

```
std::vector<int> v{ 1, 3, 5 }; // Le contenu initial de v est 1, 3, 5.
```

Elle permet également de préciser les valeurs d'initialisation par défaut pour des données membres non statiques. Cette possibilité, apportée par C++11, peut aussi être obtenue par la syntaxe d'initialisation avec le signe « = », mais pas avec les parenthèses :

```
class Widget {
  ...
```

Copyright © 2016 Dunod.

```
private:
  int x{ 0 };              // Parfait, la valeur par défaut de x est 0.
  int y = 0;               // Également parfait.
  int z(0);                // Erreur !
};
```

En revanche, les objets non copiables (par exemple `std::atomic`, voir le conseil 40) peuvent être initialisés avec des accolades ou des parenthèses, mais pas en utilisant le signe « = » :

```
std::atomic<int> ai1{ 0 };      // Parfait.

std::atomic<int> ai2(0);        // Parfait.

std::atomic<int> ai3 = 0;       // Erreur !
```

Vous devez à présent comprendre pourquoi l'initialisation à accolades est dite « uniforme ». Des trois façons de désigner une expression d'initialisation en C++, seules les accolades peuvent être employées partout.

L'initialisation à accolades apporte une nouvelle fonctionnalité, en interdisant les *conversions restrictives* implicites entre les types intégrés. Si la valeur d'une expression placée dans un initialiseur à accolades ne peut pas s'exprimer de façon sûre dans le type de l'objet en cours d'initialisation, la compilation du code échoue :

```
double x, y, z;

...

int sum1{ x + y + z };       // Erreur ! Une somme de double peut ne pas
                             // pouvoir s'exprimer sous forme de int.
```

Dans le cas d'une initialisation avec des parenthèses ou avec le signe « = », la conversion restrictive n'est pas vérifiée car cela remettrait en cause une grande quantité de code ancien :

```
int sum2(x + y + z);          // Valide (la valeur de l'expression est
                              // tronquée pour tenir dans un int).

int sum3 = x + y + z;         // Idem.
```

L'initialisation à accolades présente une autre caractéristique remarquable : elle résiste au problème de *most vexing parse* de C++. La règle de C++ selon laquelle tout ce qui peut être interprété comme une déclaration doit être interprété comme tel a un effet secondaire. Le problème de « most vexing parse » affecte en général les développeurs lorsqu'ils veulent construire par défaut un objet, mais en arrivent à déclarer à la place une fonction. L'origine du problème vient du fait qu'appeler un constructeur avec un argument peut se faire de la manière suivante :

Copyright © 2016 Dunod.
© Dunod – Toute reproduction non autorisée est un délit.

```
Widget w1(10);     // Appeler le constructeur de Widget avec
                   // l'argument 10.
```

Mais, si l'appel du constructeur de `Widget` se fait avec la même syntaxe mais sans aucun argument, nous déclarons en réalité une fonction à la place d'un objet :

```
Widget w2();    // Problème de "most vexing parse" ! Déclaration d'une
                // fonction nommée w2 qui retourne un Widget !
```

Puisque, dans la déclaration d'une fonction, la liste des paramètres ne peut pas être placée entre accolades, la construction par défaut d'un objet à l'aide des accolades n'est pas sujette à ce problème :

```
Widget w3{};       // Appeler le constructeur de Widget sans argument.
```

Nous avons beaucoup de bien à dire à propos de l'initialisation à accolades. Cette syntaxe peut être employée dans le plus grand nombre de contextes, elle évite les conversions restrictives implicites et n'est pas sujette au problème de « most vexing parse » de C++. Alors, pourquoi l'intitulé de ce conseil n'est-il pas comme « Préférer la syntaxe de l'initialisation à accolades » ?

L'inconvénient de l'initialisation à accolades réside dans un comportement parfois surprenant. Ce comportement vient de la relation assez compliquée entre les initialiseurs à accolades, les `std::initializer_list`, et de la résolution de la surcharge de constructeur. Les interactions entre tous ces éléments peuvent conduire à du code qui semble réaliser une certaine action mais qui, en réalité, en effectue une autre. Par exemple, le conseil 2 explique que si l'initialisation d'une variable déclarée avec `auto` se fait avec des accolades, le type déduit est `std::initializer_list`, alors que d'autres façons de déclarer la variable avec les mêmes initialiseurs donneraient un type plus intuitif. Par conséquent, plus on aime `auto`, moins on est enthousiaste vis-à-vis de l'initialisation à accolades.

Dans les appels aux constructeurs, les parenthèses et les accolades ont la même signification tant que des paramètres `std::initializer_list` ne sont pas impliqués :

```
class Widget {
public:
  Widget(int i, bool b);      // Constructeurs qui ne déclarent pas des
  Widget(int i, double d);    // paramètres std::initializer_list.
  …
};

Widget w1(10, true);          // Invoque le premier constructeur.

Widget w2{10, true};          // Invoque le premier constructeur.

Widget w3(10, 5.0);           // Invoque le second constructeur.

Widget w4{10, 5.0};           // Invoque le second constructeur.
```

Copyright © 2016 Dunod.

En revanche, si un ou plusieurs constructeurs déclarent un paramètre de type `std::initializer_list`, les appels fondés sur la syntaxe de l'initialisation à accolades préfèrent souvent les surcharges qui prennent des `std::initializer_list`. Si le compilateur trouve un moyen d'interpréter un appel avec un initialiseur à accolades en tant que constructeur qui prend un `std::initializer_list`, alors il fera cette interprétation. Par exemple, étendons la classe `Widget` précédente avec un constructeur qui prend un `std::initializer_list<long double>` :

```
class Widget {
public:
  Widget(int i, bool b);                              // Comme précédemment.
  Widget(int i, double d);                            // Comme précédemment.

  Widget(std::initializer_list<long double> il);      // Ajout.

  ...
};
```

Les `Widget` `w2` et `w4` sont alors créés en utilisant le nouveau constructeur, même si le type des éléments `std::initializer_list` (`long double`) donne, en comparaison des constructeurs sans `std::initializer_list`, une plus mauvaise correspondance pour les deux arguments ! Voyons cela :

```
Widget w1(10, true);      // Utilise les parenthèses et, comme
                          // précédemment, appelle le premier
                          // constructeur.

Widget w2{10, true};      // Utilise des accolades, mais appelle à
                          // présent le constructeur
                          // std::initializer_list (10 et true sont
                          // convertis en long double).

Widget w3(10, 5.0);       // Utilise les parenthèses et, comme
                          // précédemment, appelle le second
                          // constructeur.

Widget w4{10, 5.0};       // Utilise des accolades, mais appelle à
                          // présent le constructeur
                          // std::initializer_list ctor (10 et 5.0 sont
                          // convertis en long double).
```

Même ce qui serait normalement une construction par copie et déplacement peut être détourné par des constructeurs `std::initializer_list` :

```
class Widget {
public:
  Widget(int i, bool b);                              // Comme précédemment.
  Widget(int i, double d);                            // Comme précédemment.
  Widget(std::initializer_list<long double> il);      // Comme précédemment.

  operator float() const;                             // Conversion en
  ...                                                 // float.
```

Copyright © 2016 Dunod.
© Dunod – Toute reproduction non autorisée est un délit.

```
};

Widget w5(w4);      // Avec des parenthèses, appel du constructeur
                    // de copie.

Widget w6{w4};      // Avec des accolades, appel du constructeur
                    // std::initializer_list (w4 est converti en
                    // float, et un float est converti en long double).

Widget w7(std::move(w4));    // Avec des parenthèses, appel du
                             // constructeur de déplacement.

Widget w8{std::move(w4)};    // Avec des accolades, appel du
                             // constructeur std::initializer_list
                             // (même raison que pour w6).
```

La détermination des compilateurs à faire correspondre les initialiseurs à accolades aux constructeurs qui prennent des `std::initializer_list` est si forte qu'elle prévaut même si le constructeur `std::initializer_list` de meilleure correspondance ne peut pas être appelé. Par exemple :

```
class Widget {
public:
  Widget(int i, bool b);                    // Comme précédemment.
  Widget(int i, double d);                  // Comme précédemment.

  Widget(std::initializer_list<bool> il);  // Le type de l'élément est
                                            // à présent bool.

  …                                        // Aucune fonction de conversion
};                                         // implicite.

Widget w{10, 5.0};      // Erreur ! Conversions restrictives requises.
```

Dans ce cas, le compilateur ignorera les deux premiers constructeurs (le deuxième offre une correspondance exacte pour les deux types d'arguments) et tentera d'invoquer le constructeur qui prend un `std::initializer_list<bool>`. Pour appeler ce constructeur, il faut convertir un `int` (`10`) et un `double` (`5.0`) en `bool`. Ces deux conversions sont restrictives (un `bool` ne peut pas représenter de façon précise l'une ou l'autre de ces valeurs) et sont interdites à l'intérieur des initialiseurs à accolades. L'appel est donc invalide et le code est refusé.

Le compilateur se replie sur la résolution normale d'une surcharge uniquement lorsqu'il n'existe aucun moyen de convertir les types des arguments de l'initialiseur à accolades vers le type indiqué dans un `std::initializer_list`. Par exemple, si nous remplaçons le constructeur `std::initializer_list<bool>` par un constructeur qui prend un `std::initializer_list<std::string>`, les constructeurs sans `std::initializer_list` sont à nouveau éligibles car il n'existe aucun moyen de convertir des `int` et des `bool` en `std::string` :

Copyright © 2016 Dunod.

```
class Widget {
public:
  Widget(int i, bool b);                // Comme précédemment.
  Widget(int i, double d);              // Comme précédemment.

  // Un élément de std::initializer_list est à
  // présent de type std::string.
  Widget(std::initializer_list<std::string> il);
  …                                     // Aucune fonction de conversion
};                                      // implicite.

Widget w1(10, true);      // Avec des parenthèses, appelle toujours
                          // le premier constructeur.

Widget w2{10, true};      // Avec des accolades, appelle à présent
                          // le premier constructeur.

Widget w3(10, 5.0);       // Avec des parenthèses, appelle toujours
                          // le second constructeur.

Widget w4{10, 5.0};       // Avec des accolades, appelle à présent
                          // le second constructeur.
```

Voilà qui nous amène proche du terme de notre étude des initialiseurs à accolades et de la surcharge de constructeur, mais il reste encore un cas intéressant. Supposons que nous utilisions un jeu d'accolades vide pour construire un objet qui prend en charge la construction par défaut et celle avec `std::initializer_list`. Quel est le sens donné aux accolades vides ? Si elles signifient « aucun argument », nous obtenons une construction par défaut. En revanche, si elles signifient « un `std::initializer_list` vide », alors nous obtenons une construction avec un `std::initializer_list` dépourvu d'éléments.

La règle est que les accolades vides représentent non pas un `std::initializer_list` vide mais une absence d'argument. Nous obtenons donc une construction par défaut :

```
class Widget {
public:
  Widget();                                 // Constructeur par défaut.

  Widget(std::initializer_list<int> il);    // Constructeur
                                            // std::initializer_list.

  …                   // Aucune fonction de conversion implicite.
};

Widget w1;            // Appelle le constructeur par défaut.

Widget w2{};          // Appelle aussi le constructeur par défaut.

Widget w3();          // "Most vexing parse", déclare une fonction !
```

Copyright © 2016 Dunod.
© Dunod – Toute reproduction non autorisée est un délit.

Si nous *voulons* appeler un constructeur `std::initializer_list` avec un `std::initializer_list` vide, nous devons faire en sorte que les accolades vides soient un argument du constructeur. Pour cela, nous pouvons placer ces accolades vides à l'intérieur de parenthèses ou d'accolades qui délimitent le paramètre passé :

```
Widget w4({});        // Appelle le constructeur std::initializer_list
                      // avec une liste vide.

Widget w5{{}};        // Idem.
```

À ce stade, en raison de toutes ces règles assez obscures sur les initialiseurs à accolades, les `std::initializer_list` et la surcharge de constructeur qui s'entremêlent dans votre esprit, vous vous demandez peut-être à quoi peuvent bien servir autant d'informations dans les développements classiques. À bien plus que vous l'imaginez sans doute, car `std::vector` fait partie des classes directement affectées. `std::vector` dispose d'un constructeur sans `std::initializer_list` qui permet de préciser la taille de départ du conteneur et la valeur initiale de tous ses éléments. Mais il définit également un constructeur qui prend un `std::initializer_list` permettant de préciser les valeurs initiales du conteneur. Si nous créons un `std::vector` avec un type numérique (par exemple un `std::vector<int>`) et si nous passons deux arguments au constructeur, les placer entre parenthèses ou entre accolades fait une énorme différence :

```
std::vector<int> v1(10, 20);  // Utiliser le constructeur sans
                              // std::initializer_list : créer 10
                              // éléments std::vector, ayant tous
                              // la valeur 20.

std::vector<int> v2{10, 20};  // Utiliser le constructeur
                              // std::initializer_list : créer
                              // 2 éléments std::vector, l'un de valeur
                              // 10 et l'autre de valeur 20.
```

Mais repartons de `std::vector` et des détails des règles de résolution des parenthèses, des accolades et de la surcharge de constructeur. Cette discussion a deux conclusions principales. Premièrement, en tant que développeur de classes, il faut savoir que si l'ensemble des constructeurs surchargés comprend une ou plusieurs fonctions qui prennent un `std::initializer_list`, le code client qui utilise l'initialisation à accolades pourrait ne voir que les surcharges avec `std::initializer_list`. Par conséquent, il est préférable de concevoir des constructeurs de sorte que la version surchargée appelée ne dépende pas de l'utilisation des parenthèses ou des accolades. Autrement dit, il faut apprendre de ce qui est à présent considéré comme une erreur de conception de l'interface de `std::vector` et éviter de la reproduire dans nos classes.

Si l'une de nos classes n'a pas de constructeur `std::initializer_list` et si nous en ajoutons un, le code client qui utilise l'initialisation à accolades pourrait ne plus appeler les constructeurs sans `std::initializer_list` mais invoquer la nouvelle fonction. Cette modification de comportement peut évidemment se rencontrer dès que nous ajoutons une nouvelle fonction à un ensemble de surcharges : les appels qui

Copyright © 2016 Dunod.

conduisaient à l'invocation des anciennes surcharges risquent désormais d'appeler la nouvelle. Mais, dans le cas des surcharges de constructeur `std::initializer_list`, la différence tient au fait qu'une surcharge `std::initializer_list` non seulement entre en concurrence avec les autres surcharges mais les éclipse au point qu'elles risquent de ne plus être prises en compte. Il est donc indispensable de bien réfléchir avant d'ajouter de telles surcharges.

Deuxièmement, en tant que client d'une classe, nous devons faire un choix réfléchi entre les parenthèses et les accolades lors de la création des objets. La plupart des développeurs finissent par adopter par défaut l'une des deux sortes de délimiteurs, en utilisant l'autre uniquement lorsque c'est nécessaire. Ceux qui optent pour les accolades sont attirés par leur grande diversité de contextes d'application, leur refus des conversions restrictives et leur immunité au problème de « most vexing parse ». Ces programmeurs savent que, dans certains cas, comme la création d'un `std::vector` avec une taille donnée et une valeur d'élément initiale, les parenthèses sont requises. À l'opposé, les inconditionnels des parenthèses sont attirés par leur cohérence avec la syntaxe classique de C++98, l'absence du problème de la déduction `auto` qui donne un `std::initializer_list` et le fait que la création des objets ne sera pas malencontreusement détournée par les constructeurs `std::initializer_list`. Ils acceptent que les accolades soient parfois indispensables, par exemple lors de la création d'un conteneur avec des valeurs spécifiques. Rien ne permet de faire pencher la balance d'un côté ou de l'autre. Nous vous conseillons donc de choisir une approche et de l'appliquer avec constance.

Pour le développeur de templates, les hésitations entre une création d'objet avec des parenthèses ou des accolades peuvent être assez frustrantes car, en général, il est impossible de savoir celle qui doit être employée. Par exemple, supposons que nous voulions créer un objet de type quelconque à partir d'un nombre d'arguments quelconque. Conceptuellement, un template variadique apporte une réponse simple :

```
template<typename T,                // Type de l'objet à créer.
         typename... Ts>            // Types des arguments à utiliser.
void doSomeWork(Ts&&... params)
{

  Créer un objet T local à partir de params...
  ...

}
```

Il existe deux manières de remplacer la ligne de pseudo-code par du code réel (voir le conseil 25 pour une présentation de `std::forward`) :

```
T localObject(std::forward<Ts>(params)...);    // Avec des parenthèses.

T localObject{std::forward<Ts>(params)...};    // Avec des accolades.
```

Examinons le code d'appel suivant :

Copyright © 2016 Dunod.
© Dunod – Toute reproduction non autorisée est un délit.

```
std::vector<int> v;
…
doSomeWork<std::vector<int>>(10, 20);
```

Si `doSomeWork` utilise des parenthèses dans la création de `localObject`, nous obtenons un `std::vector` avec 10 éléments. S'il emploie des accolades, le résultat est un `std::vector` avec 2 éléments. Quel est le bon comportement ? Le créateur de `doSomeWork` ne peut pas le savoir. Seul l'appelant sait ce qu'il veut.

C'est précisément le problème auquel sont confrontées les fonctions `std::make_unique` et `std::make_shared` de la bibliothèque standard (voir le conseil 21). Elles l'ont résolu en utilisant les parenthèses de façon interne et en documentant ce choix dans leur interface[1].

À retenir

- L'initialisation à accolades est la syntaxe d'initialisation qui peut être employée dans le plus grand nombre de situations. Elle évite les conversions restrictives et n'est pas sujette au problème de « most vexing parse » du C++.
- Pendant la résolution de la surcharge de constructeur, les initialiseurs à accolades sont assortis à des paramètres `std::initializer_list` si cela est possible, même lorsque d'autres constructeurs donneraient potentiellement de meilleures correspondances.
- La création d'un `std::vector<`*type numérique*`>` avec deux arguments est un exemple dans lequel le choix entre les parenthèses et les accolades fait une différence importante.
- Le choix entre les parenthèses et les accolades dans la création d'objets à l'intérieur de templates peut se révéler compliqué.

CONSEIL N° 8. PRÉFÉRER NULLPTR À 0 ET À NULL

Voici le nœud de l'affaire : le littéral `0` est de type `int`, non un pointeur. Lorsque C++ trouve un `0` là où seul un pointeur peut être employé, il interprète ce `0` comme un pointeur nul, mais il s'agit d'une solution de repli. En C++, la règle de base veut que `0` soit un `int`, non un pointeur.

Sur un plan pratique, il en va de même pour `NULL`. Dans le détail, il existe toutefois quelques incertitudes sur le cas de `NULL`, car les implémentations peuvent lui donner un type entier autre que `int` (par exemple `long`). Ce n'est pas fréquent, mais peu importe car la question ici n'est pas le type précis de `NULL`, mais le fait que ni `0` ni `NULL` soient des pointeurs.

1. Il existe des conceptions plus souples, qui permettent à l'appelant de savoir si des parenthèses ou des accolades doivent être employées dans les fonctions générées par un template. Pour de plus amples informations, consultez, sur le blog C++ tenu par Andrzej, l'article du 5 juin 2013 intitulé « Intuitive interface — Part I » (*http://akrzemi1.wordpress.com/2013/06/05/intuitive-interface-part-i/*).

Copyright © 2016 Dunod.

En C++98, la principale conséquence de cette caractéristique était que la surcharge sur des pointeurs et des types entiers pouvait amener quelques surprises. En passant `0` ou `NULL` à ces surcharges, une surcharge avec pointeur n'était jamais invoquée :

```
void f(int);        // Trois surcharges de f.
void f(bool);
void f(void*);

f(0);               // Appelle f(int), non f(void*).

f(NULL);            // Peut ne pas compiler, mais appelle en
                    // général f(int), jamais f(void*).
```

L'incertitude concernant le comportement de `f(NULL)` vient de la latitude accordée aux implémentations vis-à-vis du type de `NULL`. Par exemple, si `NULL` est défini par `0L` (c'est-à-dire `0` de type `long`), l'appel est ambigu car les conversions d'un `long` en `int`, d'un `long` en `bool` et de `0L` en `void*` sont considérées aussi correctes l'une que l'autre. Le point intéressant dans cet appel vient de la contradiction entre la signification *apparente* du code source (« j'appelle `f` avec `NULL` – le pointeur nul ») et son sens *réel* (« j'appelle `f` avec une sorte d'entier – non le pointeur nul »). C'est en raison de ce comportement contre-intuitif qu'il a été conseillé aux programmeurs C++98 d'éviter la surcharge sur des pointeurs et des entiers. Ces directives restent valides en C++11, car, malgré le conseil donné ici, il est probable que certains développeurs continueront d'employer `0` et `NULL`, alors que `nullptr` constitue un meilleur choix.

L'intérêt de `nullptr` vient du fait qu'il n'est pas de type entier. Pour être honnête, il n'est pas non plus de type pointeur, mais il peut être vu comme un pointeur sur n'importe quel type. Le type réel de `nullptr` est `std::nullptr_t` et, par une jolie définition circulaire, `std::nullptr_t` est défini comme étant le type de `nullptr`. Le type `std::nullptr_t` convertit implicitement tous les types de pointeur bruts et c'est pourquoi `nullptr` agit comme s'il était un pointeur de n'importe quel type.

En appelant la fonction surchargée `f` avec `nullptr`, nous invoquons la surcharge `void*` (c'est-à-dire celle avec pointeur), car `nullptr` ne peut pas être considéré comme un entier :

```
f(nullptr);         // Appelle la surcharge f(void*).
```

En remplaçant `0` ou `NULL` par `nullptr`, nous évitons donc les surprises de la résolution de la surcharge, mais ce n'est pas le seul avantage. Cette approche améliore également la clarté du code, notamment en présence de variables `auto`. Par exemple, supposons que l'extrait suivant provienne d'une base de code :

```
auto result = findRecord( /* arguments */ );

if (result == 0) {
  ...
}
```

Copyright © 2016 Dunod.
© Dunod – Toute reproduction non autorisée est un délit.

Si nous ne savons pas (ou ne pouvons pas trouver facilement) ce que renvoie `findRecord`, il ne sera pas facile de déterminer si `result` est un pointeur ou un entier. En effet, `0` (utilisé dans la comparaison avec `result`) peut représenter l'un ou l'autre. En revanche, le code suivant n'est pas ambigu :

```
auto result = findRecord( /* arguments */ );

if (result == nullptr) {
  …
}
```

`result` doit être un type pointeur.

`nullptr` révèle tout son intérêt avec les templates. Supposons que nous ayons des fonctions qui doivent être appelées uniquement lorsque le mutex approprié a été verrouillé. Chaque fonction prend une sorte de pointeur différente :

```
int    f1(std::shared_ptr<Widget> spw);  // Appeler ces fonctions
double f2(std::unique_ptr<Widget> upw);  // lorsque le mutex
bool   f3(Widget* pw);                   // approprié est verrouillé.
```

Voici comment écrire un code appelant qui veut passer des pointeurs nuls :

```
std::mutex f1m, f2m, f3m;         // Mutex pour f1, f2 et f3.

using MuxGuard =                  // typedef de C++11 ; voir le conseil 9.
  std::lock_guard<std::mutex>;
…

{
  MuxGuard g(f1m);              // Verrouiller le mutex pour f1.
  auto result = f1(0);          // Passer 0 comme pointeur nul à f1.
}                               // Déverrouiller le mutex.

…

{
  MuxGuard g(f2m);              // Verrouiller le mutex pour f2.
  auto result = f2(NULL);       // Passer NULL comme pointeur nul à f2.
}                               // Déverrouiller le mutex.

…

{
  MuxGuard g(f3m);              // Verrouiller le mutex pour f3.
  auto result = f3(nullptr);    // Passer nullptr comme pointeur nul à f3.
}                               // Déverrouiller le mutex.
```

Il est regrettable de ne pas utiliser `nullptr` dans les deux premiers appels, mais le code est opérationnel et ce n'est pas négligeable. En revanche, la répétition dans le code appelant – verrouiller le mutex, appeler une fonction, déverrouiller le mutex – est plus gênante. Elle est même perturbante. C'est notamment pour éviter ce type

Copyright © 2016 Dunod.

de duplication dans le code source que les templates ont été conçus. Voyons donc comment les exploiter dans ce cas :

```
template<typename FuncType,
         typename MuxType,
         typename PtrType>
auto lockAndCall(FuncType func,
                 MuxType& mutex,
                 PtrType ptr) -> decltype(func(ptr))
{
  MuxGuard g(mutex);
  return func(ptr);
}
```

Si le type de retour de cette fonction (`auto ... -> decltype(func(ptr))` vous questionne, consultez le conseil 3 pour comprendre ce qui se passe. Vous y découvrirez que, en C++14, le type de retour peut se réduire à un simple `decltype(auto)` :

```
template<typename FuncType,
         typename MuxType,
         typename PtrType>
decltype(auto) lockAndCall(FuncType func,        // C++14.
                           MuxType& mutex,
                           PtrType ptr)
{
  MuxGuard g(mutex);
  return func(ptr);
}
```

En prenant le template `lockAndCall`, quelle que soit sa version, le code appelant peut s'écrire de la manière suivante :

```
auto result1 = lockAndCall(f1, f1m, 0);          // Erreur !

...

auto result2 = lockAndCall(f2, f2m, NULL);       // Erreur !

...

auto result3 = lockAndCall(f3, f3m, nullptr);    // Parfait.
```

Bien que le code puisse être écrit ainsi, deux des trois cas ne compilent pas. Dans le premier appel, le problème vient du fait que le passage de `0` à `lockAndCall` déclenche la déduction de type de template afin de déterminer le type de `0`. Il s'agit toujours d'un `int`, qui est donc le type du paramètre `ptr` à l'intérieur de cet appel à `lockAndCall`. Malheureusement, cela implique qu'un `int` est passé à `func` lors de l'appel à cette fonction dans `lockAndCall`, alors que ce type n'est pas compatible avec le paramètre `std::shared_ptr<Widget>` attendu par `f1`. Le `0` indiqué dans l'appel à `lockAndCall` était supposé représenter un pointeur nul, alors qu'en réalité un `int` ordinaire a été passé. Tenter de passer ce `int` à `f1` en tant que `std::shared_ptr<Widget>` déclenche

Copyright © 2016 Dunod.
© Dunod – Toute reproduction non autorisée est un délit.

une erreur de type. L'appel à `lockAndCall` avec `0` échoue, car, dans le template, un `int` est passé à une fonction qui exige un `std::shared_ptr<Widget>`.

L'analyse de l'appel avec `NULL` est comparable. Lorsque `NULL` est passé à `lockAndCall`, le type déduit pour le paramètre est un entier et une erreur de type se produit lorsque `ptr` (un `int` ou équivalent) est passé à `f2`, car cette fonction attend un `std::unique_ptr<Widget>`.

À l'opposé, l'appel fondé sur `nullptr` ne pose aucun problème. Lorsque `nullptr` est passé à `lockAndCall`, le type déduit pour `ptr` est `std::nullptr_t`. Lorsque `ptr` est passé à `f3`, une conversion implicite de `std::nullptr_t` vers `Widget*` est effectuée. En effet, un `std::nullptr_t` peut implicitement être converti en n'importe quel type de pointeur.

Le fait que la déduction de type de template détermine le « mauvais » type pour `0` et `NULL` (c'est-à-dire leur véritable type plutôt que leur représentation annexe d'un pointeur nul) constitue une raison irréfutable de remplacer `0` ou `NULL` par `nullptr` pour faire référence à un pointeur nul. Avec `nullptr`, les templates ne posent aucun problème particulier. Si l'on ajoute à cela le fait que `nullptr` ne souffre pas des surprises amenées par la surcharge avec `0` et `NULL`, le dossier est clos. Si vous devez faire référence à un pointeur nul, utilisez non pas `0` ou `NULL` mais `nullptr`.

À retenir

- Préférer `nullptr` à `0` et à `NULL`.
- Éviter la surcharge sur des types entiers et des pointeurs.

CONSEIL N° 9. PRÉFÉRER LES DÉCLARATIONS D'ALIAS AUX TYPEDEF

Nous sommes certains que vous pensez vous aussi que l'utilisation des conteneurs STL est une bonne idée, et nous espérons que le conseil 18 saura vous convaincre qu'utiliser `std::unique_ptr` l'est également. Cependant, nous sommes convaincus que personne n'aime saisir plusieurs fois des types comme `std::unique_ptr<std::unordered_map<std::string, std::string>>`. Rien que d'y penser, le risque de souffrir du syndrome du canal carpien augmente.

Pour éviter de tels soucis médicaux, il suffit de se tourner vers `typedef` :

```
typedef
  std::unique_ptr<std::unordered_map<std::string, std::string>>
  UPtrMapSS;
```

Mais les `typedef` font beaucoup trop C++98. Ils sont effectivement reconnus en C++11, mais cette version du C++ offre les *déclarations d'alias* :

Copyright © 2016 Dunod.

```
using UPtrMapSS =
  std::unique_ptr<std::unordered_map<std::string, std::string>>;
```

Puisque `typedef` et une déclaration d'alias produisent exactement le même résultat, il est légitime de se demander s'il existe une véritable raison technique de préférer l'un à l'autre.

Elle existe, mais, avant de la présenter, mentionnons que de nombreux programmeurs trouvent que la déclaration d'alias est plus commode lorsque les types impliquent des pointeurs de fonctions :

```
// FP est synonyme d'un pointeur sur une fonction qui attend un int
// et un const std::string&, et qui n'a pas de valeur de retour.
typedef void (*FP)(int, const std::string&);     // Avec typedef.

// Même chose que précédemment.
using FP = void (*)(int, const std::string&);    // Avec une déclaration
                                                 // d'alias.
```

Évidemment, aucune des deux formes n'est particulièrement difficile à comprendre et peu de développeurs passent beaucoup de temps à gérer des synonymes de pointeurs de fonctions. Ce n'est donc pas la raison de préférer les déclarations d'alias aux `typedef`.

En revanche, les templates apportent une très bonne raison. Plus précisément, les déclarations d'alias peuvent être transformées en templates (auquel cas, elles sont appelées *alias de template*), ce qui est impossible avec `typedef`. Les programmeurs en C++11 disposent alors d'un mécanisme simple pour écrire des expressions qui, en C++98, nécessitent une combinaison de `typedef` imbriqués dans des `struct` définis comme des templates. Supposons, par exemple, que nous voulions définir un synonyme pour une liste chaînée qui utilise un allocateur personnalisé, `MyAlloc`. Avec un alias de template, c'est un jeu d'enfant :

```
template<typename T>                              // MyAllocList<T>
using MyAllocList = std::list<T, MyAlloc<T>>;     // est synonyme de
                                                  // std::list<T,
                                                  // MyAlloc<T>>.

MyAllocList<Widget> lw;                           // Code client.
```

Avec `typedef`, il faut tout constituer à partir de zéro :

```
template<typename T>                        // MyAllocList<T>::type
struct MyAllocList {                        // est synonyme de
  typedef std::list<T, MyAlloc<T>> type;    // std::list<T, MyAlloc<T>>
};

MyAllocList<Widget>::type lw;               // Code client.
```

Copyright © 2016 Dunod.
© Dunod – Toute reproduction non autorisée est un délit.

Cela s'aggrave si nous souhaitons utiliser `typedef` à l'intérieur d'un template dans le but de créer une liste chaînée qui contient des objets du type spécifié par un paramètre de template. Il faut alors placer `typename` avant le nom `typedef` :

```
template<typename T>
class Widget {                          // Widget<T> contient
private:                                // un MyAllocList<T>
  typename MyAllocList<T>::type list;   // comme donnée member.
  ...
};
```

Dans cet exemple, `MyAllocList<T>::type` fait référence à un type qui dépend du paramètre de type du template (`T`). `MyAllocList<T>::type` est donc un *type dépendant*, et l'une des nombreuses règles sympathiques de C++ stipule que les noms des types dépendants doivent être précédés de `typename`.

Si la définition de `MyAllocList` se fait sous forme d'un alias de template, `typename` n'est plus nécessaire, tout comme le suffixe « `::type` » plutôt encombrant :

```
template<typename T>
using MyAllocList = std::list<T, MyAlloc<T>>;  // Comme précédemment.

template<typename T>
class Widget {
private:
  MyAllocList<T> list;                         // Plus de "typename",
  ...                                          // ni de "::type".
};
```

De notre point de vue, on pourrait penser que `MyAllocList<T>` (l'utilisation de l'alias de template) est aussi dépendant du paramètre de template `T` que `MyAllocList<T>::type` (l'utilisation du `typedef` imbriqué), mais nous ne sommes pas un compilateur. Lorsque le compilateur traite le template `Widget` et rencontre l'utilisation de `MyAllocList<T>` (l'utilisation de l'alias de template), il sait que `MyAllocList<T>` correspond au nom d'un type car `MyAllocList` est un alias de template : il *doit* nommer un type. `MyAllocList<T>` est donc un *type non dépendant* et le spécificateur `typename` n'est ni requis ni autorisé.

En revanche, lorsque le compilateur rencontre `MyAllocList<T>::type` (l'utilisation du `typedef` imbriqué) dans le template de `Widget`, il ne peut pas savoir avec certitude qu'il correspond au nom d'un type. En effet, il s'agit d'une spécialisation de `MyAllocList` qui n'a pas encore été vue et où `MyAllocList<T>::type` fait référence à autre chose qu'un type. Cela peut sembler insensé, mais il ne faut pas reprocher cette possibilité au compilateur. Ce sont des personnes en chair et en os qui sont réputées produire un tel code.

Par exemple, un programmeur mal avisé pourrait avoir concocté le code suivant :

```
class Wine { ... };
```

Copyright © 2016 Dunod.

```
template<>                          // Spécialisation de MyAllocList
class MyAllocList<Wine> {           // lorsque T est un Wine.
private:
  enum class WineType               // Plus d'infos sur "enum class"
  { White, Red, Rose };             // au conseil 10.

  WineType type;                    // Dans cette classe, type est
  …                                 // une donnée membre !
};
```

Vous le constatez, `MyAllocList<Wine>::type` ne fait pas référence à un type. Si une instance de `Widget` est créée avec un `Wine`, le `MyAllocList<T>::type` à l'intérieur du template `Widget` fait référence à une donnée membre, non à un type. Dans le template `Widget`, que `MyAllocList<T>::type` fasse ou non référence à un type dépend de `T` et c'est pourquoi le compilateur insiste pour que nous confirmions qu'il s'agit d'un type en ajoutant `typename`.

Si vous vous êtes déjà essayé à la métaprogrammation avec des templates (TMP, *template metaprogramming*), vous vous êtes certainement heurté à la nécessité de prendre des paramètres de type de template et d'en créer des versions revues. Par exemple, étant donné un type `T`, vous pourriez vouloir retirer les qualificatifs `const` ou de référence présents dans `T`, et ainsi transformer `const std::string&` en `std::string`. Ou bien, vous pourriez souhaiter ajouter `const` à un type ou le transformer en référence lvalue, par exemple convertir `Widget` en `const Widget` ou en `Widget&`. (Si vous n'avez jamais utilisé la TMP, c'est vraiment dommage car, pour être un programmeur C++ réellement efficace, vous devrez vous familiariser au moins avec les bases de cet aspect du C++. Vous trouverez des exemples de TMP, notamment les transformations de type précédentes, aux conseils 23 et 27.)

C++11 apportent les outils qui permettent d'effectuer ces transformations au travers des *traits de type*. Il s'agit d'un assortiment de templates dans l'en-tête `<type_traits>`. Cet en-tête en comprend des dizaines, tous ne réalisant pas des transformations de types, mais ceux qui le font ont une interface connue. Étant donné un type `T` auquel nous souhaitons appliquer une transformation, le type résultant est `std::`*`transformation`*`<T>::type`. Par exemple :

```
std::remove_const<T>::type          // Obtenir T à partir de const T.

std::remove_reference<T>::type      // Obtenir T à partir de T& et de T&&.

std::add_lvalue_reference<T>::type  // Obtenir T& à partir de T.
```

Les commentaires résument simplement ce que font les transformations ; il ne faut donc pas trop les prendre au pied de la lettre. Avant de les employer dans un projet, examinez leurs spécifications détaillées.

Nous n'avons pas pour objectif ici de présenter les traits de type. Notons simplement que l'application de ces transformations conduit à l'ajout de « `::type` » à la fin de chaque utilisation. Si elles doivent être appliquées à un paramètre de type dans un template (ce qui correspond à leur emploi classique dans du code réel), il faut faire

Copyright © 2016 Dunod.
© Dunod – Toute reproduction non autorisée est un délit.

précéder chaque utilisation par `typename`. Ces deux contraintes syntaxiques viennent du fait que les traits de type en C++11 sont implémentés sous forme de `typedef` imbriqués dans des templates de `struct`. Vous avez bien lu, ils sont mis en œuvre avec la technique que nous disons inférieure à celle des alias de templates !

La raison en est historique, mais nous ne la donnerons pas. Le comité de normalisation a reconnu tardivement que les alias de templates constituent une meilleure solution et a inclus dans C++14 des templates pour toutes les transformations de type de C++11. Les alias ont une forme commune : pour chaque transformation `std::`*`transformation`*`<T>::type` de C++11, il existe un alias de template C++14 correspondant nommé `std::`*`transformation`*`_t`. Voici quelques exemples :

```
std::remove_const<T>::type           // C++11 : const T → T.
std::remove_const_t<T>               // Équivalent C++14.

std::remove_reference<T>::type       // C++11 : T&/T&& → T.
std::remove_reference_t<T>           // Équivalent C++14.

std::add_lvalue_reference<T>::type   // C++11 : T → T&.
std::add_lvalue_reference_t<T>       // Équivalent C++14.
```

Les constructions C++11 restent valides en C++14, mais il n'y a aucune raison de les utiliser. Si aucun compilateur C++14 n'est pas disponible, écrire nous-mêmes les alias de templates est un jeu d'enfant. Cela nécessite uniquement les caractéristiques du langage C++11, et reproduire un motif est à la portée de n'importe qui. Il est même possible de récupérer une copie électronique de la norme C++14 et de procéder par copier-coller. Voici un point de départ :

```
template <class T>
using remove_const_t = typename remove_const<T>::type;

template <class T>
using remove_reference_t = typename remove_reference<T>::type;

template <class T>
using add_lvalue_reference_t = typename add_lvalue_reference<T>::type;
```

Difficile de faire plus simple.

À retenir

- Les `typedef` sont incompatibles avec les templates, contrairement aux déclarations d'alias.
- Les alias de templates permettent d'éviter le suffixe « `::type` » et, dans les templates, le préfixe « `typename` » souvent requis pour faire référence aux `typedef`.
- C++14 apporte des alias de templates pour toutes les transformations effectuées par des traits de type en C++11.

Copyright © 2016 Dunod.

CONSEIL N° 10. PRÉFÉRER LES `enum` DÉLIMITÉS AUX `enum` NON DÉLIMITÉS

En règle générale, la déclaration d'un nom à l'intérieur d'accolades limite la visibilité de ce nom à la portée fixée par ces accolades. Ce n'est pas le cas pour les énumérateurs déclarés avec les `enum` dans le style de C++98. Les noms de ces énumérateurs appartiennent à la portée dans laquelle se trouve l'`enum`. Cela signifie qu'aucun autre élément dans cette portée ne peut prendre le même nom. Par exemple :

```
enum Color { black, white, red };   // black, white et red sont dans
                                    // la même portée que Color.

auto white = false;                 // Erreur ! white est déjà déclaré
                                    // dans cette portée.
```

Ces énumérations sont dites *non délimitées*, car les noms des énumérateurs ont la même portée que celle de leur définition `enum`. Avec leurs nouveaux homologues C++11, les *énumérations délimitées* (ou `enum` fortement typé), les noms ne sortent pas des accolades :

```
enum class Color { black, white, red };  // black, white et red sont
                                         // limités à Color.

auto white = false;                 // Valide, aucun autre
                                    // "white" dans la portée.

Color c = white;                    // Erreur ! Aucune énumérateur nommé
                                    // "white" dans cette portée.

Color c = Color::white;             // Valide.

auto c = Color::white;              // Valide (et en accord avec
                                    // le conseil 5).
```

Puisque les `enum` délimités sont déclarés avec « `enum class` », ils sont parfois appelés *classe* `enum`.

La diminution de la pollution de l'espace de noms obtenue grâce aux énumérations délimitées suffit déjà à les préférer aux énumérations non délimitées. Mais elles ont également un autre avantage : leurs énumérateurs sont plus fortement typés. Les énumérateurs des `enum` non délimités sont implicitement convertis en types entiers (et, à partir de là, en types à virgule flottante). Les parodies sémantiques suivantes sont donc parfaitement valides :

```
enum Color { black, white, red };       // enum non délimité.

std::vector<std::size_t>                // Fonction qui retourne les
  primeFactors(std::size_t x);          // facteurs premiers de x.
```

Copyright © 2016 Dunod.
© Dunod – Toute reproduction non autorisée est un délit.

```
Color c = red;
…

if (c < 14.5) {                // Comparer un Color à un double (!)

  auto factors =               // Déterminer les facteurs premiers
    primeFactors(c);           // d'un Color (!)
  …
}
```

En revanche, en ajoutant simplement « `class` » après « `enum` », l'énumération non délimitée devient une énumération délimitée et le fonctionnement change totalement. Dans un `enum` délimité, les énumérateurs ne sont plus convertis implicitement en un autre type :

```
enum class Color { black, white, red }; // enum à présent délimité.

Color c = Color::red;                   // Comme précédemment, mais avec
…                                       // un qualificateur de portée.

if (c < 14.5) {                // Erreur ! Impossible de comparer
                               // un Color et un double.

  auto factors =           // Erreur ! Impossible de passer un Color
    primeFactors(c);       // à une fonction qui attend un std::size_t.
  …
}
```

Pour effectuer une conversion intentionnelle d'un `Color` vers un autre type, il faut employer la méthode habituelle qui permet de soumettre le système de typage à notre bon vouloir :

```
if (static_cast<double>(c) < 14.5) {      // Code bizarre mais valide.

  auto factors =                                // Suspect, mais la
    primeFactors(static_cast<std::size_t>(c)); // compilation réussit.
  …
}
```

On pourrait également trouver un troisième avantage aux énumérations délimitées par rapport aux énumérations non délimitées. Il est en effet possible de les utiliser avec une déclaration anticipée, c'est-à-dire de déclarer leur nom sans préciser leurs énumérateurs :

```
enum Color;              // Erreur !

enum class Color;        // Valide.
```

Cela peut prêter à confusion. En C++11, les `enum` non délimités peuvent également faire l'objet d'une déclaration anticipée, mais sous condition d'un petit travail supplémentaire. Il s'agit d'exploiter le fait que chaque `enum` C++ possède un type

Copyright © 2016 Dunod.

entier sous-jacent déterminé par le compilateur. Prenons l'exemple de l'énumération `Color` :

```
enum Color { black, white, red };
```

Le compilateur peut choisir `char` comme type sous-jacent, car il n'a que trois valeurs à représenter. En revanche, certains `enum` affichent une plage de valeurs beaucoup plus vaste :

```
enum Status { good = 0,
              failed = 1,
              incomplete = 100,
              corrupt = 200,
              indeterminate = 0xFFFFFFFF
            };
```

Dans ce cas, les valeurs à représenter vont de `0` à `0xFFFFFFFF`. Sur une machine classique (certains systèmes spécifiques représentent un `char` avec au moins 32 bits), le compilateur devra choisir un type entier plus grand qu'un `char` pour représenter les valeurs de `Status`.

Afin d'optimiser l'utilisation de la mémoire, le compilateur choisit souvent pour représenter un `enum` le plus petit type sous-jacent capable d'accepter l'étendue des valeurs de l'énumérateur. Le compilateur pourra également décider d'optimiser la vitesse plutôt que la taille et, dans ce cas, ne choisira pas le type sous-jacent le plus petit possible, tout en gardant une taille raisonnable. Pour que cela soit possible, C++98 accepte uniquement les définitions d'`enum` (tous les énumérateurs doivent être indiqués) ; les déclarations d'`enum` ne sont pas autorisées. Le compilateur est ainsi capable de sélectionner le type sous-jacent de chaque `enum` avant que l'énumération ne soit utilisée.

Mais l'incapacité à déclarer de façon anticipée des `enum` présente des inconvénients. Le principal étant probablement l'augmentation des dépendances de compilation. Reprenons l'énumération `Status` :

```
enum Status { good = 0,
              failed = 1,
              incomplete = 100,
              corrupt = 200,
              indeterminate = 0xFFFFFFFF
            };
```

Il est fort probable qu'un tel `enum` sera utilisé dans de nombreux endroits d'un système et qu'il sera donc inclus dans un fichier d'en-tête dont dépendra chaque partie de ce système. Supposons qu'une nouvelle valeur d'état soit ajoutée :

```
enum Status { good = 0,
              failed = 1,
              incomplete = 100,
```

Copyright © 2016 Dunod.
© Dunod – Toute reproduction non autorisée est un délit.

```
            corrupt = 200,
            audited = 500,
            indeterminate = 0xFFFFFFFF
        };
```

Cette modification imposera certainement une nouvelle compilation de l'intégralité du système, même si un seul sous-système, voire une seule fonction, se sert de l'énumérateur ajouté. C'est vraiment le genre de choses que les développeurs haïssent. Et c'est le genre de choses que les déclarations anticipées d'`enum` en C++11 permettent d'éviter. Par exemple, voici une déclaration parfaitement valide d'une énumération délimitée et d'une fonction qui l'attend en paramètre :

```
enum class Status;                  // Déclaration anticipée.

void continueProcessing(Status s);  // Utiliser l'enum déclaré en amont.
```

L'en-tête qui contient ces déclarations n'a pas besoin d'être recompilé lorsque la *définition* de `Status` évolue. Par ailleurs, si `Status` est modifié (par exemple pour ajouter l'énumérateur `audited`) et si le comportement de `continueProcessing` n'en est pas affecté (par exemple parce que `continueProcessing` n'utilise pas `audited`), il n'est pas utile de recompiler son implémentation.

Mais, si le compilateur a besoin de connaître la taille d'un `enum` avant que l'énumération puisse être utilisée, comment peut-il s'en sortir avec les `enum` C++11 déclarés en amont alors qu'il n'y parvient pas avec les `enum` C++98. La réponse est simple : le type sous-jacent d'une énumération délimitée est toujours connu et, pour une énumération non délimitée, nous pouvons le préciser.

Pour les `enum` délimités, le type sous-jacent par défaut est `int` :

```
enum class Status;                  // Le type sous-jacent est int.
```

Si le type par défaut ne convient pas, nous pouvons le changer :

```
enum class Status: std::uint32_t;  // Le type sous-jacent pour
                                   // Status est std::uint32_t
                                   // (extrait de <cstdint>).
```

Pour une énumération délimitée, le compilateur connaît donc toujours la taille des énumérateurs.

Dans le cas d'une énumération non délimitée, nous pouvons préciser le type sous-jacent de la même manière et le résultat peut faire l'objet d'une déclaration anticipée :

```
enum Color: std::uint8_t;          // Déclaration anticipée d'un enum
                                   // non délimité ; le type sous-jacent
                                   // est std::uint8_t
```

Copyright © 2016 Dunod.

La spécification du type sous-jacent peut également se faire sur une définition d'`enum` :

```
enum class Status: std::uint32_t { good = 0,
                                   failed = 1,
                                   incomplete = 100,
                                   corrupt = 200,
                                   audited = 500,
                                   indeterminate = 0xFFFFFFFF
                                 };
```

Étant donné que les `enum` délimités évitent la pollution de l'espace de noms et qu'ils ne sont pas sujets aux conversions de type implicites ineptes, vous risquez d'être surpris d'apprendre qu'il existe au moins un cas où les `enum` non délimités présentent un intérêt : dans les références à des champs dans des `std::tuple` de C++11. Supposons par exemple que nous ayons un tuple qui contient des valeurs représentant le nom, l'adresse électronique et la renommée d'un utilisateur sur un site de réseau social :

```
using UserInfo =                // Alias de type (voir le conseil 9)
  std::tuple<std::string,       // nom
             std::string,       // adresse électronique
             std::size_t>;      // renommée
```

Même si les commentaires expliquent ce que chaque champ du tuple représente, ces informations ne seront probablement pas très utiles lors de la lecture d'un code qui se trouve dans un fichier source distinct :

```
UserInfo uInfo;                 // Objet du type du tuple.
…

auto val = std::get<1>(uInfo);  // Obtenir la valeur du champ 1.
```

En tant que programmeurs, nous avons de nombreux aspects à suivre. Comment pouvons-nous nous rappeler que le champ 1 correspond à l'adresse électronique de l'utilisateur ? La solution passe par une énumération non délimitée de façon à associer des noms aux numéros des champs :

```
enum UserInfoFields { uiName, uiEmail, uiReputation };

UserInfo uInfo;                         // Comme précédemment.
…

auto val = std::get<uiEmail>(uInfo);    // Obtenir la valeur du champ de
                                        // l'adresse électronique.
```

Cela fonctionne en raison de la conversion implicite de `UserInfoFields` en `std::size_t`, précisément le type demandé par `std::get`.

Le code équivalent avec des énumérations délimitées est un tantinet plus verbeux :

Copyright © 2016 Dunod.
© Dunod – Toute reproduction non autorisée est un délit.

```
enum class UserInfoFields { uiName, uiEmail, uiReputation };

UserInfo uInfo;                         // Comme précédemment.
...

auto val =
  std::get<static_cast<std::size_t>(UserInfoFields::uiEmail)>
    (uInfo);
```

Il est possible de faire plus court en écrivant une fonction qui prend en paramètre un énumérateur et qui retourne la valeur `std::size_t` correspondante, mais la solution est un peu tirée par les cheveux. `std::get` est un template et la valeur que nous fournissons est un argument de template (notez l'utilisation de crochets obliques, non de parenthèses). Par conséquent, la fonction qui transforme un énumérateur en un `std::size_t` doit générer son résultat *pendant la compilation*. Comme l'explique le conseil 15, elle doit donc être une fonction `constexpr`.

En réalité, elle doit être un template de fonction `constexpr` car elle doit opérer avec n'importe quel `enum`. Et si nous devons effectuer cette généralisation, elle doit aussi concerner le type de retour. Au lieu de renvoyer `std::size_t`, nous retournons le type sous-jacent de l'`enum`. Il est disponible *via* les traits de type `std::underlying_type` (voir le conseil 9 pour de plus amples informations sur les traits de type). Enfin, nous la déclarons `noexcept` (voir le conseil 14), car nous savons qu'elle ne lancera jamais d'exception. Nous obtenons un template de fonction `toUType` qui prend en argument un énumérateur quelconque et retourne sa valeur au moment de la compilation :

```
template<typename E>
constexpr typename std::underlying_type<E>::type
  toUType(E enumerator) noexcept
{
  return
    static_cast<typename
                std::underlying_type<E>::type>(enumerator);
}
```

En C++14, il est possible de simplifier `toUType` en remplaçant `typename std::underlying_type<E>::type` par l'écriture plus courte `std::underlying_type_t` (voir le conseil 9) :

```
template<typename E>                                  // C++14.
constexpr std::underlying_type_t<E>
  toUType(E enumerator) noexcept
{
  return static_cast<std::underlying_type_t<E>>(enumerator);
}
```

Le type de retour, encore plus court, `auto` (voir le conseil 3) est également valide en C++14 :

Copyright © 2016 Dunod.

```
template<typename E>                                    // C++14.
constexpr auto
  toUType(E enumerator) noexcept
{
  return static_cast<std::underlying_type_t<E>>(enumerator);
}
```

Quelle que soit la manière de le décrire, `toUType` nous permet d'accéder à un champ du tuple de la manière suivante :

```
auto val = std::get<toUType(UserInfoFields::uiEmail)>(uInfo);
```

Si cela reste encore plus long qu'avec une énumération non délimitée, nous évitons la pollution de l'espace de noms et les conversions accidentelles sur les énumérateurs. Dans de nombreux cas, la saisie de quelques caractères supplémentaires est un prix raisonnable à payer pour éviter les pièges associés à une technologie d'énumération qui date de l'époque où le modem à 2 400 bauds représentait l'état de l'art des télécommunications numériques.

À retenir

- Les `enum` de style C++98 sont appelés énumérations non délimitées.
- Dans les énumérations délimitées, les énumérateurs sont visibles uniquement à l'intérieur de l'`enum`. Ils sont convertis dans d'autres types uniquement par des conversions explicites.
- Les énumérations délimitées ou non acceptent la spécification du type sous-jacent. Pour les `enum` délimités, le type sous-jacent par défaut est `int`. Les `enum` non délimités n'ont pas de type sous-jacent par défaut.
- Il est toujours possible de déclarer de façon anticipée des énumérations délimitées. Cela reste possible avec les énumérations non délimitées, mais le type sous-jacent doit être précisé.

CONSEIL N° 11. PRÉFÉRER LES FONCTIONS SUPPRIMÉES AUX FONCTIONS INDÉFINIES PRIVÉES

Si nous fournissons du code à d'autres développeurs et souhaitons les empêcher d'appeler une fonction précise, il suffit de ne pas déclarer cette fonction. Sans fonction déclarée, il ne peut pas y avoir de fonction à appeler. Facile comme tout. Mais il arrive que C++ déclare des fonctions à notre place et si nous voulons éviter que les clients n'appellent ces fonctions, ce n'est plus du tout aussi simple.

Ce cas se produit uniquement pour les « fonctions membres spéciales », c'est-à-dire les fonctions membres générées automatiquement par C++ lorsqu'elles sont requises. Le conseil 17 détaille ces fonctions mais, pour le moment, nous allons nous intéresser uniquement au constructeur de copie et à l'opérateur d'affectation par copie. Ce

Copyright © 2016 Dunod.
© Dunod – Toute reproduction non autorisée est un délit.

chapitre est largement consacré aux pratiques courantes en C++98 remplacées par de meilleures pratiques en C++11. Et, en C++98, lorsque l'on souhaite supprimer l'usage d'une fonction, il s'agit presque toujours du constructeur de copie, de l'opérateur d'affectation, ou des deux.

En C++98, la solution consiste à déclarer ces fonctions `private` et à ne pas les définir. Par exemple, près du début de la hiérarchie des iostream dans la bibliothèque standard de C++, nous trouvons le template de classe `basic_ios`. Toutes les classes d'istream et d'ostream dérivent, parfois indirectement, de celle-ci. La copie d'istream et d'ostream n'est pas souhaitable car le fonctionnement de telles opérations n'est pas parfaitement clair. Par exemple, un objet `istream` représente un flux de valeurs d'entrée, dont certaines peuvent déjà avoir été lues, d'autres l'étant potentiellement plus tard. Si un istream devait être copié, faudrait-il copier toutes les valeurs qui ont déjà été lues ainsi que celles qui seront lues par la suite ? La manière la plus simple de répondre à ces questions est de les faire disparaître. C'est précisément ce qui se passe en interdisant la copie des flux.

Pour que les classes d'istream et d'ostream ne puissent pas être copiées, `basic_ios` est spécifié de la manière suivante en C++98 (commentaires compris) :

```
template <class charT, class traits = char_traits<charT> >
class basic_ios : public ios_base {
public:
  ...

private:
  basic_ios(const basic_ios& );            // not defined
  basic_ios& operator=(const basic_ios&);  // not defined
};
```

Puisque ces fonctions sont déclarées `private`, elles ne peuvent pas être appelées depuis du code client. Si du code a néanmoins accès à ces fonctions (fonctions membres ou classes `friend`) et les utilise, l'édition de liens échouera car elles ne sont pas définies.

En C++11, il existe une meilleure manière d'obtenir les mêmes résultats : faire du constructeur de copie et de l'opérateur d'affectation par copie des *fonctions supprimées* en les marquant avec « `= delete` ». Voici la même partie de `basic_ios`, cette fois-ci dans sa version C++11 :

```
template <class charT, class traits = char_traits<charT> >
class basic_ios : public ios_base {
public:
  ...
  basic_ios(const basic_ios& ) = delete;
  basic_ios& operator=(const basic_ios&) = delete;
  ...
};
```

On pourrait croire que la différence entre supprimer ces fonctions et les déclarer `private` n'est qu'une question de mode, mais elle est en réalité beaucoup plus profonde.

Copyright © 2016 Dunod.

Les fonctions supprimées ne peuvent pas être utilisées de quelque façon que ce soit, et même le code présent dans des fonctions membres ou `friend` ne compilera pas s'il tente de copier des objets `basic_ios`. Voilà une amélioration par rapport à C++98, où une telle utilisation impropre ne serait découverte qu'à l'édition de liens.

Par convention, les fonctions supprimées sont déclarées non pas `private` mais `public`. En effet, lorsque du code client tente d'utiliser une fonction membre, C++ vérifie l'accessibilité avant l'état supprimé. Si un code client tente d'utiliser une fonction supprimée `private`, certains compilateurs indiqueront uniquement qu'elle est privée, même si son accessibilité est en réalité sans rapport avec la possibilité de l'utiliser. Il est préférable de tenir compte de ce point lorsque du code ancien est revu de façon à remplacer les fonctions membres `private` non définies par des fonctions supprimées. En déclarant ces nouvelles fonctions `public`, les messages d'erreurs seront généralement plus précis.

Les fonctions supprimées ont également un autre avantage important : n'importe quelle fonction peut être supprimée, tandis que seules les fonctions membres peuvent être `private`. Par exemple, supposons que nous ayons une fonction non membre qui prend un entier et renvoie un chiffre porte-bonheur :

```
bool isLucky(int number);
```

Les origines C du C++ font que pratiquement n'importe quel type peut être considéré comme plus ou moins numérique, avec une conversion implicite en `int`. Cependant certains appels, qui passent à la compilation, pourraient avoir peu de sens :

```
if (isLucky('a')) …              // 'a' est-il un chiffre porte-bonheur ?

if (isLucky(true)) …             // Et "true" ?

if (isLucky(3.5)) …              // Faut-il tronquer à 3 avant de
                                 // vérifier si on est chanceux ?
```

Si les chiffres porte-bonheur doivent réellement être des entiers, il faut que ces appels ne puissent pas être compilés.

Une solution consiste à créer des surcharges supprimées pour les types à écarter :

```
bool isLucky(int number);          // Fonction d'origine.

bool isLucky(char) = delete;       // Rejeter les caractères.

bool isLucky(bool) = delete;       // Rejeter les booléens.

bool isLucky(double) = delete;     // Rejeter les double et les float.
```

Le commentaire sur la surcharge avec `double` indique que les `double` et les `float` seront écartés. Cela pourrait vous surprendre, mais n'oubliez pas que, lorsqu'un `float` peut être converti en `int` ou en `double`, C++ choisit la conversion vers un `double`. En appelant `isLucky` avec un `float`, la surcharge pour un `double`, non celle pour un `int`,

Copyright © 2016 Dunod.
© Dunod – Toute reproduction non autorisée est un délit.

est appelée. En réalité, elle ne sera pas appelée car elle est supprimée, ce qui interdit la compilation de l'appel.

Même si les fonctions supprimées ne peuvent pas être utilisées, elles font partie du programme. Elles sont donc prises en compte lors de la résolution de la surcharge. C'est pourquoi, avec les déclarations de fonctions supprimées précédentes, les appels indésirables à `isLucky` seront rejetés :

```
if (isLucky('a')) ...          // Erreur ! Appel à une fonction supprimée.

if (isLucky(true)) ...         // Erreur !

if (isLucky(3.5f)) ...         // Erreur !
```

De plus, les fonctions supprimées, contrairement aux fonctions membres `private`, permettent d'empêcher l'instanciation de template. Par exemple, supposons que nous ayons besoin d'un template qui manipule des pointeurs intégrés (le chapitre 4 conseille d'adopter les pointeurs intelligents plutôt que les pointeurs bruts) :

```
template<typename T>
void processPointer(T* ptr);
```

Dans le monde des pointeurs, il existe deux cas particuliers. Le premier concerne les pointeurs `void*`, car il n'existe aucun moyen de les déréférencer, de les incrémenter ou de les décrémenter, etc. Le second concerne les pointeurs `char*`, car ils représentent souvent des pointeurs sur des chaînes de caractères de type C, non des pointeurs sur des caractères individuels. Ces cas spéciaux nécessitent souvent un traitement particulier et, dans le cas du template `processPointer`, nous supposons que l'objectif est de rejeter les appels qui utilisent ces types. Autrement dit, il faut que les appels à `processPointer` avec des pointeurs `void*` ou `char*`ne soient pas autorisés.

Pour cela, il suffit de marquer les instanciations suivantes par `delete` :

```
template<>
void processPointer<void>(void*) = delete;

template<>
void processPointer<char>(char*) = delete;
```

Si un appel à `processPointer` avec un `void*` ou un `char*` est désormais invalide, il est probable qu'un appel avec un `const void*` ou un `const char*` devrait l'être également. Par conséquent, ces instanciations doivent elles aussi être supprimées :

```
template<>
void processPointer<const void>(const void*) = delete;

template<>
void processPointer<const char>(const char*) = delete;
```

Copyright © 2016 Dunod.

Pour être véritablement rigoureux, nous devons également supprimer les surcharges avec `const volatile void*` et `const volatile char*`, puis travailler sur celles avec des pointeurs des autres types de caractère standard : `std::wchar_t`, `std::char16_t` et `std::char32_t`.

Si nous avons un template de fonction dans une classe et si nous souhaitons interdire certaines instanciations en les déclarant `private` (à la manière classique de C++98), ce n'est tout simplement pas possible. En effet, on ne peut pas donner à une spécialisation de template de fonction membre un niveau d'accès différent de celui du template principal. Supposons, par exemple, que `processPointer` soit un template de fonction membre dans `Widget` et que nous voulions interdire les appels avec des pointeurs `void*`. Même si elle ne compilera pas, voici l'approche C++98 :

```
public:
  …
  template<typename T>
void processPointer(T* ptr)
  { … }

private:
  template<>                                    // Erreur !
  void processPointer<void>(void*);

};
```

Le problème vient du fait que les spécialisations de template doivent être écrites non pas dans une portée de classe mais dans la portée de l'espace de noms. Ce problème n'existe pas avec les fonctions supprimées, car elles n'ont pas besoin d'un niveau d'accès différent. Elles peuvent être supprimées en dehors de la classe (donc dans la portée de l'espace de noms) :

```
class Widget {
public:
  …
  template<typename T>
  void processPointer(T* ptr)
  { … }
  …

};

template<>                                          // Toujours publique,
void Widget::processPointer<void>(void*) = delete;  // mais supprimée.
```

En vérité, la déclaration de fonctions `private` sans les définir était la tentative C++98 d'obtenir ce que les fonctions supprimées permettent d'accomplir en C++11. L'approche de C++98 n'est pas aussi bonne. Elle ne fonctionne pas en dehors des classes, elle ne fonctionne pas toujours à l'intérieur des classes et, lorsqu'elle fonctionne, ce n'est souvent qu'au moment de l'édition de liens. Il est donc préférable de s'en tenir aux fonctions supprimées.

Copyright © 2016 Dunod.
© Dunod – Toute reproduction non autorisée est un délit.

À retenir

- Les fonctions supprimées doivent être préférées aux fonctions privées non définies.
- N'importe quelle fonction peut être supprimée, y compris les fonctions non membres et les instanciations de template.

CONSEIL N° 12. DÉCLARER LES FONCTIONS DE SUBSTITUTION AVEC OVERRIDE

En C++, le monde de la programmation orientée objet tourne autour des classes, de l'héritage et des fonctions virtuelles. L'une des idées fondamentales qui régissent ce monde est que les implémentations des fonctions virtuelles dans des classes dérivées *redéfinissent* les implémentations de leurs homologues dans la classe de base. Il est donc assez décourageant de réaliser à quel point la redéfinition d'une fonction virtuelle peut aisément mal se passer. C'est un peu comme si cette partie du langage était conçue avec l'idée qu'on ne devait pas respecter la loi de Murphy, simplement l'honorer.

Puisque « redéfinition » sonne beaucoup comme « surcharge », rappelons clairement que c'est grâce à la redéfinition d'une fonction virtuelle que nous pouvons invoquer une fonction d'une classe dérivée au travers d'une interface de la classe de base :

```
class Base {
public:
  virtual void doWork();    // Fonction virtuelle de la classe de base.
  …
};

class Derived: public Base {
public:
  virtual void doWork();        // Redéfinition de Base::doWork
  …                             // ("virtual" est facultatif dans
};                              // ce cas).

std::unique_ptr<Base> upb =     // Créer un pointeur de la classe de base
  std::make_unique<Derived>();  // vers un objet de la classe dérivée ;
                                // voir le conseil 21 pour des infos
…                               // sur std::make_unique.

upb->doWork();                  // Appeler doWork via le pointeur ptr
                                // de la classe de base ; la fonction
                                // de la classe dérivée est invoquée.
```

La mise en place de la redéfinition requiert plusieurs conditions :

- La fonction de la classe de base doit être virtuelle.

Copyright © 2016 Dunod.

- Les noms de la fonction dans la classe de base et dans la classe dérivée doivent être identiques (excepté dans le cas des destructeurs).
- Les types des paramètres de la fonction dans la classe de base et dans la classe dérivée doivent être identiques.
- Les caractères `const` de la fonction dans la classe de base et dans la classe dérivée doivent correspondre.
- Les types de retour et les spécifications des exceptions de la fonction dans la classe de base et dans la classe dérivée doivent être compatibles.

À ces contraintes, qui font partie de C++98, C++11 en ajoute une autre :

- Les *qualificatifs de référence* des fonctions doivent être identiques. Les qualificatifs de référence sur les fonctions membres sont parmi les fonctionnalités les moins connues de C++11 ; ne soyez pas surpris si vous n'en avez jamais entendu parler. Ils permettent de restreindre l'utilisation d'une fonction membre uniquement aux lvalues ou aux rvalues. Les fonctions membres n'ont pas besoin d'être virtuelles pour les utiliser :

```
class Widget {
public:
  …
  void doWork() &;       // Cette version de doWork est utilisée
                         // uniquement lorsque *this est une lvalue.

  void doWork() &&;      // Cette version de doWork est utilisée
};                       // uniquement lorsque *this est une rvalue.

…

Widget makeWidget();     // Fonction fabrique (retourne une rvalue).

Widget w;                // Objet normal (une lvalue).

…

w.doWork();              // Appeler Widget::doWork pour les lvalues
                         // (c'est-à-dire Widget::doWork &).

makeWidget().doWork();   // Appeler Widget::doWork pour les rvalues
                         // (c'est-à-dire Widget::doWork &&).
```

Nous reviendrons ultérieurement sur les fonctions avec des qualificatifs de référence mais, pour le moment, notez simplement que si une fonction virtuelle dans une classe de base est marquée d'un qualificatif de référence, la redéfinition de cette fonction dans la classe dérivée doit comprendre exactement le même qualificatif de référence. Dans le cas contraire, les fonctions déclarées existeront dans la classe dérivée, mais elles ne redéfiniront pas celles de la classe de base.

Toutes ces exigences sur la redéfinition signifient que de petites erreurs peuvent faire une grande différence. En général, le code qui contient des erreurs de redéfinition ne posera pas de problème de compilation, mais il n'aura pas le fonctionnement

Copyright © 2016 Dunod.
© Dunod – Toute reproduction non autorisée est un délit.

attendu. Nous ne pouvons donc pas compter sur le compilateur pour nous prévenir. Par exemple, le code suivant est parfaitement valide et, à première vue, semble raisonnable, mais il ne contient aucune redéfinition de fonction virtuelle – aucune fonction de classe dérivée n'est liée à une fonction de la classe de base. Pouvez-vous identifier le problème de chaque cas, c'est-à-dire indiquer pourquoi chaque fonction de la classe dérivée ne redéfinit pas la fonction de même nom dans la classe de base ?

```
class Base {
public:
  virtual void mf1() const;
  virtual void mf2(int x);
  virtual void mf3() &;
  void mf4() const;
};

class Derived: public Base {
public:
  virtual void mf1();
  virtual void mf2(unsigned int x);
  virtual void mf3() &&;
  void mf4() const;
};
```

Voici un peu d'aide :

- `mf1` est déclarée `const` dans `Base`, mais pas dans `Derived`.
- `mf2` prend un paramètre de type `int` dans `Base`, mais de type `unsigned int` dans `Derived`.
- `mf3` a un qualificatif de lvalue dans `Base`, mais de rvalue dans `Derived`.
- `mf4` n'est pas déclarée `virtual` dans `Base`.

Vous pourriez penser que, dans la pratique, tous ces points conduiront à des avertissements du compilateur et que vous n'avez donc pas à vous en inquiéter. C'est peut-être vrai, mais pas certain. Deux des compilateurs que nous avons testés ont accepté le code sans broncher, alors même que le niveau d'avertissement était à son maximum. (D'autres compilateurs ont affiché des avertissements uniquement pour certains problèmes.)

Puisque la déclaration des redéfinitions dans une classe dérivée est importante pour que le code fonctionne correctement, mais qu'il est facile de se tromper, C++11 apporte une solution pour indiquer explicitement qu'une fonction d'une classe dérivée est *supposée* redéfinir une version de la classe de base : il suffit de la déclarer avec `override`. Appliquons cette solution à l'exemple précédent :

```
class Derived: public Base {
public:
  virtual void mf1() override;
  virtual void mf2(unsigned int x) override;
  virtual void mf3() && override;
  virtual void mf4() const override;
};
```

Copyright © 2016 Dunod.

Ce code ne compilera pas car, écrit de cette manière, le compilateur se plaindra de problèmes associés à la redéfinition. C'est précisément ce que nous souhaitons et c'est pourquoi il faut déclarer toutes les fonctions de substitution avec `override`.

Voici la version valide du code fondé sur `override` (en supposant que l'objectif soit que toutes les fonctions de `Derived` redéfinissent les fonctions virtuelles de `Base`) :

```
class Base {
public:
  virtual void mf1() const;

  virtual void mf2(int x);

  virtual void mf3() &;

  virtual void mf4() const;
};

class Derived: public Base {
public:
  virtual void mf1() const override;

  virtual void mf2(int x) override;

  virtual void mf3() & override;

  void mf4() const override;        // Ajouter "virtual" est facultatif.
```

Vous aurez noté que, dans cet exemple, déclarer `mf4` virtuelle dans `Base` fait partie des corrections. En général, les problèmes de redéfinition proviennent des classes dérivées, mais il est également possible que les erreurs se trouvent dans les classes de base.

En appliquant `override` à toutes les redéfinitions dans une classe dérivée, nous obtenons bien plus que des messages du compilateur en cas de problème. En effet, cela nous aide à mesurer les ramifications d'un changement de la signature d'une fonction virtuelle dans une classe de base. Si les classes dérivées emploient systématiquement `override`, nous pouvons simplement modifier la signature, recompiler l'ensemble et constater les dommages que nous pouvons avoir causés (c'est-à-dire le nombre de classes dérivées qui ne compilent plus). Ensuite, il ne reste plus qu'à décider si la modification de la signature en vaut la peine. Sans `override`, nous devrions nous reposer sur l'exhaustivité des tests unitaires car, nous l'avons vu, le compilateur ne nous préviendra pas si des fonctions virtuelles d'une classe dérivée ne redéfinissent pas des fonctions de la classe de base alors que nous le supposons.

Les mots clés existent depuis toujours en C++, mais C++11 ajoute deux *mots clés contextuels*, `override` et `final`[1]. Ces mots clés ont une signification réservée, mais

1. Appliquer `final` à une fonction virtuelle empêche la redéfinition de cette fonction dans les classes dérivées. `final` peut également être appliqué à une classe, auquel cas celle-ci ne pourra pas servir de classe de base.

Copyright © 2016 Dunod.
© Dunod – Toute reproduction non autorisée est un délit.

uniquement dans certains contextes. Ainsi, `override` ne prend son sens réservé que s'il se trouve à la fin de la déclaration d'une fonction membre. Autrement dit, si du code ancien utilise déjà le nom `override`, il est inutile de le modifier pour le rendre compatible avec C++11 :

```
class Warning {           // Ancienne classe de C++98.
public:
  …
  void override();        // Valide en C++98 et en C++11
  …                       // (avec le même sens).
};
```

Nous en avons fini avec `override`, mais ce n'est pas le cas des qualificatifs de référence sur les fonctions membres. Voici le complément d'information que nous avions promis.

Si nous souhaitons écrire une fonction qui accepte uniquement des arguments lvalue, nous déclarons un paramètre de référence lvalue non `const` :

```
void doSomething(Widget& w);  // Accepter uniquement des Widget lvalue.
```

Pour écrire une fonction qui accepte uniquement des arguments rvalue, nous déclarons un paramètre de référence rvalue :

```
void doSomething(Widget&& w); // Accepter uniquement des Widget rvalue.
```

Grâce aux qualificatifs de référence sur les fonctions membres, nous pouvons appliquer la même distinction sur l'objet sur lequel une fonction membre est invoquée, c'est-à-dire sur `*this`. On pourrait comparer cela au `const` placé à la fin de la déclaration d'une fonction membre pour indiquer que l'objet sur lequel la fonction membre est invoquée (c'est-à-dire `*this`) est `const`.

Les fonctions membres avec des qualificatifs de référence sont rarement nécessaires, mais cela arrive. Par exemple, supposons que notre classe `Widget` comprenne une donnée membre `std::vector` et que nous proposions une méthode accesseur qui donne au code client un accès direct à cette donnée :

```
class Widget {
public:
  using DataType = std::vector<double>;       // Voir le conseil 9 pour
  …                                           // des infos sur "using".

  DataType& data() { return values; }
  …

private:
  DataType values;
};
```

Copyright © 2016 Dunod.

On peut difficilement dire que cette encapsulation soit parfaite, mais mettons cela de côté et voyons ce qui se passe dans le code client suivant :

```
Widget w;
…

auto vals1 = w.data();              // Copier w.values dans vals1.
```

Le type de la valeur de retour de `Widget::data` est une référence lvalue (plus précisément `std::vector<double>&`) et puisque ces références sont définies comme des lvalues, nous initialisons `vals1` à partir d'une lvalue. Par conséquent, `vals1` est construit par copie à partir de `w.values`, comme l'indique le commentaire.

Supposons à présent que nous ayons une fonction fabrique qui crée des `Widget` :

```
Widget makeWidget();
```

Et que nous souhaitions initialiser une variable avec le `std::vector` qui se trouve à l'intérieur du `Widget` renvoyé par `makeWidget` :

```
auto vals2 = makeWidget().data();     // Copier dans vals2 les values
                                      // présents dans le Widget.
```

`Widget::data` retourne à nouveau une référence lvalue et, à nouveau, cette référence est une lvalue. Par conséquent, notre nouvel objet (`vals2`) est encore construit par copie à partir du `values` qui se trouve dans le `Widget`. Toutefois, le `Widget` est cette fois-ci l'objet temporaire renvoyé par `makeWidget` (c'est-à-dire une rvalue) et la copie du `std::vector` qu'il contient constitue une perte de temps. Il serait préférable de le déplacer car `data` retourne une référence lvalue et les règles du C++ exigent que le compilateur génère du code pour une copie. (Il serait possible de mettre en place une optimisation au travers de ce que l'on appelle la règle « as if », mais il serait stupide de se fonder sur le compilateur pour trouver une façon d'en tirer parti.)

Ce dont nous avons besoin, c'est une manière de préciser que l'invocation de `data` sur un `Widget` rvalue doit produire une rvalue. Nous l'avons, en utilisant les qualificatifs de référence pour surcharger `data` selon que le `Widget` est une lvalue ou une rvalue :

```
class Widget {
public:
  using DataType = std::vector<double>;
  …

  DataType& data() &                // Pour les Widget lvalue,
  { return values; }                // retourner une lvalue.

  DataType data() &&                // Pour les Widget rvalue,
  { return std::move(values); }     // retourner une rvalue.
```

© Dunod – Toute reproduction non autorisée est un délit.
Copyright © 2016 Dunod.

```
  ...

private:
  DataType values;
};
```

Notez la différence dans les types de retour de surcharge de `data`. La surcharge pour les références lvalue renvoie une référence lvalue (c'est-à-dire une lvalue), tandis que la surcharge pour les références rvalue renvoie un objet temporaire (c'est-à-dire une rvalue). Le code client se comporte à présent comme nous le souhaitons :

```
auto vals1 = w.data();              // Appelle la surcharge lvalue de
                                    // Widget::data, construit vals1
                                    // par copie.

auto vals2 = makeWidget().data();   // Appelle la surcharge rvalue de
                                    // Widget::data, construit vals2
                                    // par déplacement.
```

Tout cela est très bien, mais ne laissons pas cette fin heureuse nous distraire du véritable sujet de ce conseil : dès lors que nous déclarons dans une classe dérivée une fonction qui doit redéfinir une fonction virtuelle de la classe de base, nous devons la déclarer avec `override`.

À retenir

- Les fonctions de substitution doivent être déclarées avec `override`.
- Les qualificatifs de référence sur les fonctions membres permettent de traiter les objets lvalue et rvalue (`*this`) de façon différente.

CONSEIL N° 13. PRÉFÉRER LES `const_iterator` AUX `iterator`

Les `const_iterator` sont les équivalents STL des pointeurs sur `const`. Ils pointent sur des valeurs qui ne peuvent pas être modifiées. La bonne pratique qui veut que `const` soit utilisé dès que c'est possible engage à utiliser les `const_iterator` chaque fois qu'un itérateur est requis et que les éléments ciblés par cet itérateur n'ont pas besoin d'être modifiés.

C'est vrai en C++98 comme en C++11, mais, en C++98, la prise en charge des `const_iterator` était médiocre. Il n'était pas facile de les créer et, une fois cela fait, les façons de les utiliser étaient limitées. Par exemple, supposons que nous voulions rechercher dans un `std::vector<int>` la première occurrence de 1983 (l'année où « C++ » est devenu le nom du langage de programmation à la place de « C avec des classes »), puis insérer la valeur 1998 (l'année où la première norme ISO du C++ a été adoptée) à cet emplacement. Si le vecteur ne contient pas 1983, l'insertion doit se faire à la fin du vecteur. Avec les `iterator` de C++98, la solution est simple :

Copyright © 2016 Dunod.

```
std::vector<int> values;

...

std::vector<int>::iterator it =
  std::find(values.begin(),values.end(), 1983);
values.insert(it, 1998);
```

Cependant, les `iterator` ne sont pas vraiment adaptés dans ce cas, car ce code ne modifie jamais l'élément pointé. La modification du code de façon à utiliser des `const_iterator` devrait être immédiate, mais ce n'est absolument pas le cas en C++98. Voici une approche conceptuellement appropriée, mais non valide :

```
typedef std::vector<int>::iterator IterT;             // Définitions des
typedef std::vector<int>::const_iterator ConstIterT;  // types.

std::vector<int> values;

...

ConstIterT ci =
  std::find(static_cast<ConstIterT>(values.begin()),  // Conversions
            static_cast<ConstIterT>(values.end()),    // des types.
            1983);

values.insert(static_cast<IterT>(ci), 1998);  // Peut ne pas compiler ;
                                              // voir ci-après.
```

Bien entendu, les `typedef` ne sont pas obligatoires, mais ils simplifient l'écriture des conversions de types. (Vous vous demandez peut-être pourquoi nous utilisons `typedef` au lieu de suivre le conseil 9, qui préconise l'usage des déclarations d'alias. La raison en est simple : cet exemple montre du code C++98 et les déclarations d'alias sont une nouvelle fonctionnalité de C++11.)

Nous utilisons des conversions de type dans l'appel à `std::find` car `values` est un conteneur non `const` et, en C++98, obtenir un `const_iterator` à partir d'un conteneur non `const` est compliqué. Elles ne sont pas strictement nécessaires, car il est possible d'obtenir des `const_iterator` par d'autres moyens, comme lier `values` à une variable de type référence à un `const`, puis utiliser cette variable à la place de `values` dans le code. Toutefois, quelle que soit la méthode qui sera mise en place, obtenir des `const_iterator` sur des éléments d'un conteneur non `const` restera complexe.

Après avoir obtenu les `const_iterator`, les choses se corsent car, en C++98, les emplacements des insertions (et des suppressions) ne peuvent être précisés que par des `iterator` ; les `const_iterator` ne sont pas acceptés. C'est pourquoi, dans le code précédent, nous convertissons le `const_iterator` (obtenu à partir de `std::find`) en un `iterator` : passer un `const_iterator` à `insert` ne compilerait pas.

Pour être honnête, le code montré ne compilera pas car il n'existe aucune conversion portable d'un `const_iterator` en un `iterator`, pas même à l'aide d'un `static_cast`. La solution lourde appelée `reinterpret_cast` n'y parviendra pas plus. (Il ne

Copyright © 2016 Dunod.
© Dunod – Toute reproduction non autorisée est un délit.

s'agit pas d'une limite de C++98, C++11 est également concerné. Les `const_iterator` ne peuvent tout simplement pas être convertis en `iterator`.) Il existe des solutions portables pour générer des `iterator` qui pointent aux mêmes emplacements que des `const_iterator`, mais elles sont complexes, elles ne s'appliquent pas dans tous les cas et ne valent pas la peine d'être présentées dans cet ouvrage. Quoi qu'il en soit, nous espérons que vous avez à présent compris que les `const_iterator` posaient tellement de difficultés en C++98 qu'ils méritaient rarement que l'on s'y intéresse. En fin de compte, les développeurs n'utilisent pas `const` dès que c'est *possible*, mais lorsque cela se révèle *pratique* et, en C++98, les `const_iterator` n'étaient pas très pratiques.

Tout cela a changé en C++11. Les `const_iterator` sont désormais faciles à obtenir et à utiliser. Les fonctions membres `cbegin` et `cend` d'un conteneur génèrent des `const_iterator`, même dans le cas d'un conteneur non `const`, et les fonctions membres de STL qui identifient des emplacements avec des itérateurs (comme `insert` et `erase`) se servent de `const_iterator`. Pour employer les `const_iterator` de C++11 dans le code C++98 original qui utilise des `iterator`, rien n'est plus simple :

```
std::vector<int> values;                              // Comme précédemment.

...

auto it =                                             // Utiliser cbegin
  std::find(values.cbegin(),values.cend(), 1983);     // et cend.

values.insert(it, 1998);
```

Ce code emploie maintenant des `const_iterator`, qui se révèlent pratiques !

Il existe un cas où la prise en charge des `const_iterator` en C++11 atteint ses limites : lors de l'écriture d'une bibliothèque aussi générique que possible. Un tel code tient compte du fait que certains conteneurs et structures de données de type conteneur fournissent `begin` et `end` (ainsi que `cbegin`, `cend`, `rbegin`, etc.) sous forme de fonctions *non membres* plutôt que membres. C'est par exemple le cas des tableaux intégrés et de certaines bibliothèques tierces dont les interfaces sont constituées uniquement de fonctions indépendantes. Un code totalement générique emploie donc des fonctions non membres, sans supposer l'existence de versions membres.

Nous pouvons ainsi généraliser le code sur lequel nous avons travaillé sous forme d'un template `findAndInsert` :

```
template<typename C, typename V>
void findAndInsert(C& container,              // Dans container, rechercher
                   const V& targetVal,        // la première occurrence de
                   const V& insertVal)        // targetVal, puis insérer
{                                             // insertVal à l'emplacement
  using std::cbegin;                          // trouvé.
  using std::cend;

  auto it = std::find(cbegin(container),      // cbegin non membre.
                      cend(container),        // cend non membre.
                      targetVal);
```

Copyright © 2016 Dunod.

```
  container.insert(it, insertVal);
}
```

Cela fonctionne en C++14, mais pas en C++11. En raison d'une omission pendant la normalisation, C++11 a ajouté les fonctions non membres `begin` et `end`, mais `cbegin`, `cend`, `rbegin`, `rend`, `crbegin` et `crend` ont été oubliées. C++14 a corrigé le tir.

Si nous utilisons C++11 et voulons obtenir le code le plus générique possible, mais si aucune des bibliothèques que nous utilisons ne fournit les templates manquants pour les versions non membres de `cbegin` et ses amies, nous pouvons proposer facilement nos propres implémentations. Par exemple, voici celle d'une fonction `cbegin` non membre :

```
template <class C>
auto cbegin(const C& container)->decltype(std::begin(container))
{
  return std::begin(container);       // Voir les explications ci-après.
}
```

N'êtes-vous pas surpris de constater que la fonction non membre `cbegin` n'appelle pas la fonction membre `cbegin` ? Nous l'avons également été, mais suivons la logique. Ce template de `cbegin` accepte n'importe quel type d'argument qui représente une structure de données de type conteneur, `C`, et y accède au travers de son paramètre `container`, de type référence sur `const`. Si `C` est un type de conteneur classique (par exemple un `std::vector<int>`), `container` sera une référence à une version `const` de ce conteneur (par exemple un `const std::vector<int>&`). L'invocation de la fonction non membre `begin` (fournie par C++11) sur un conteneur `const` donne un `const_iterator`, et cet itérateur est celui retourné par ce template. Cette manière d'implémenter les choses a un avantage : elle fonctionne même avec les conteneurs qui offrent une fonction membre `begin` (qui, pour les conteneurs, est appelée par la fonction non membre `begin` de C++11), mais sans proposer de fonction membre `cbegin`. Nous pouvons ainsi utiliser cette fonction non membre `cbegin` avec des conteneurs qui reconnaissent uniquement `begin`.

Ce template fonctionne également lorsque `C` est un type tableau intégré. Dans ce cas, `container` devient une référence à un tableau `const`. C++11 fournit une version de la fonction non membre `begin` adaptée aux tableaux et retournant un pointeur sur le premier élément du tableau. Les éléments d'un tableau `const` sont de type `const`, et le pointeur que la fonction non membre `begin` renvoie pour un tableau `const` est un pointeur sur un `const`. Ce type de pointeur est en réalité un `const_iterator` pour un tableau. (Pour plus de détails sur la spécialisation d'un template pour les tableaux intégrés, consulter le conseil 1 qui traite de la déduction de type dans les templates qui prennent en paramètres des références sur des tableaux.)

Revenons à l'essentiel. L'objectif de ce conseil est de vous encourager à utiliser les `const_iterator` dès que vous le pouvez. La motivation de base – employer `const` dès que cela a un sens – date d'avant C++11 mais, lors de la manipulation d'itérateurs en

Copyright © 2016 Dunod.
© Dunod – Toute reproduction non autorisée est un délit.

C++98, elle n'était pas très facile à mettre en pratique. En C++11, ce n'est plus le cas, et C++14 va plus loin en terminant le travail que C++11 a laissé derrière lui.

À retenir

- Préférer les `const_iterator` aux `iterator`.
- Dans un code aussi générique que possible, préférer les versions non membres de `begin`, `end`, `rbegin`, etc., aux fonctions membres équivalentes.

CONSEIL N° 14. DÉCLARER `NOEXCEPT` LES FONCTIONS QUI NE LANCENT PAS D'EXCEPTIONS

En C++98, les exceptions étaient plutôt difficiles à dompter. Il fallait récapituler les types des exceptions qu'une fonction pouvait lancer et, si l'implémentation de cette fonction était modifiée, les spécifications d'exceptions devaient également être revues. Un tel changement pouvait remettre en question le fonctionnement du code client car les différents appels pouvaient dépendre de la spécification initiale des exceptions. Le compilateur apportait généralement peu d'aide pour maintenir la cohérence entre l'implémentation des fonctions, les spécifications des exceptions et le code client. De nombreux programmeurs ont fini par décider que les spécifications d'exceptions en C++98 ne valaient pas la peine qu'ils s'y attardent.

Pendant les réflexions sur C++11, un consensus a émergé : l'information véritablement intéressante est de savoir si une fonction en lance ou non des exceptions. Tout blanc ou tout noir : soit une fonction peut lancer une exception, soit il est certain qu'elle n'en lancera pas. Cette dichotomie éventuellement/jamais forme le socle des spécifications d'exceptions en C++11, qui remplacent celles de C++98. (La variante C++98 reste prise en charge, mais elle est déclarée obsolète.) En C++11, la version non conditionnelle de `noexcept` est destinée aux fonctions qui ne lanceront jamais d'exceptions.

La décision de déclarer une fonction `noexcept` se prend lors de la conception de l'interface. Pour le code client, il est très important de connaître le comportement d'une fonction vis-à-vis des exceptions. Il lui est possible de savoir si une fonction est déclarée `noexcept` et la réponse peut avoir un impact sur son efficacité et sur sa sûreté face aux exceptions. Il est donc aussi important de savoir si une fonction est `noexcept` que de savoir si une fonction membre est `const`. Ne pas déclarer une fonction `noexcept` alors qu'elle est réputée ne pas lancer d'exceptions révèle une spécification médiocre de l'interface.

Il existe un autre avantage à déclarer `noexcept` les fonctions qui ne produisent aucune exception : le compilateur est capable de générer un meilleur code objet. Pour en comprendre la raison, nous allons examiner la différence entre les façons C++98 et C++11 d'indiquer qu'une fonction ne génère pas d'exceptions. Prenons une fonction `f` qui assure aux appelants qu'ils ne recevront jamais d'exceptions. Voici les deux façons d'exprimer cette affirmation :

Copyright © 2016 Dunod.

```
int f(int x) throw();  // f ne lance pas d'exception : variante C++98.

int f(int x) noexcept; // f ne lance pas d'exception : variante C++11.
```

Si, au moment de l'exécution, une exception sort de `f`, la spécification d'exception de `f` n'est pas respectée. Dans le contexte de C++98, la pile des appels est déroulée jusqu'à l'appelant de `f` et, après quelques actions qui ne nous intéressent pas, l'exécution du programme se termine. Dans le contexte de C++11, le comportement à l'exécution est légèrement différent : la pile est *peut-être* déroulée avant que l'exécution du programme ne soit terminée.

La différence entre dérouler systématiquement et éventuellement la pile des appels a un impact important sur la génération du code. Avec une fonction `noexcept`, l'optimiseur n'a pas besoin de conserver la pile d'exécution dans un état déroulable, ni de s'assurer que les objets qu'elle contient sont détruits dans l'ordre inverse de leur construction, juste pour le cas où une exception se propagerait en dehors de la fonction. Les fonctions qui utilisent « `throw()` », ainsi que celles dépourvues de spécifications d'exceptions, n'autorisent pas une telle souplesse d'optimisation. Voici comment nous pouvons résumer la situation :

```
TypeRetour fonction (params) noexcept;     // Optimisable au maximum.

TypeRetour fonction (params) throw();      // Peu optimisable.

TypeRetour fonction (params);              // Peu optimisable.
```

Ce bénéfice justifie à lui seul le choix de déclarer les fonctions `noexcept` dès lors que l'on est certain qu'elles ne produisent aucune exception.

Pour certaines fonctions, le cas est plus complexe. Les opérations de déplacement en sont de bons exemples. Supposons que nous ayons une base de code C++98 qui utilise un `std::vector<Widget>`. Des `Widget` sont ajoutés de temps en temps au `std::vector` à l'aide de `push_back` :

```
std::vector<Widget> vw;

...

Widget w;

...                          // Manipuler w.

vw.push_back(w);             // Ajouter w à vw.

...
```

Supposons que ce code fonctionne parfaitement et que nous n'ayons aucun intérêt à le modifier pour le convertir en C++11. Pourtant, nous souhaitons tirer parti de la sémantique de déplacement de C++11 qui peut améliorer les performances du code ancien lorsque des types compatibles avec le déplacement sont impliqués. Nous

Copyright © 2016 Dunod.
© Dunod – Toute reproduction non autorisée est un délit.

nous assurons donc que `Widget` dispose des opérations de déplacement, que ce soit en les écrivant nous-mêmes ou en vérifiant que les conditions de leur génération automatique sont remplies (voir le conseil 17).

Lors de l'ajout d'un nouvel élément à un `std::vector`, il est possible de se trouver dans le cas où il n'y a plus de place pour cet élément, autrement dit que la taille du `std::vector` est égale à sa capacité. Lorsque cela se produit, le `std::vector` alloue une nouvelle zone de mémoire plus vaste pour contenir ses éléments. Il transfère ensuite des éléments depuis la zone de mémoire actuelle vers la nouvelle. En C++98, ce transfert se faisait en copiant chaque élément depuis l'ancienne zone de mémoire vers la nouvelle, puis en détruisant les objets dans l'ancienne zone. Cette solution permettait à `push_back` d'offrir une forte garantie de sécurité vis-à-vis des exceptions : si une exception était lancée au cours de la copie des éléments, l'état du `std::vector` restait inchangé car aucun des éléments qui se trouvaient dans l'ancienne zone de mémoire n'était détruit tant qu'ils n'avaient pas tous étés copiés dans la nouvelle zone.

En C++11, une optimisation naturelle serait de remplacer la copie des éléments du `std::vector` par des déplacements. Malheureusement, cette solution fait courir un risque sur la sécurité de `push_back` vis-à-vis des exceptions. Si `n` éléments ont été déplacés depuis l'ancienne zone de mémoire et si une exception est lancée au cours du déplacement de l'élément `n+1`, le travail de `push_back` ne peut pas être mené à son terme. Cependant, le `std::vector` d'origine a été modifié : `n` de ses éléments ont été déplacés. Sa restauration dans l'état précédent n'est pas toujours possible, car déplacer chaque élément dans la zone de mémoire initiale peut également déclencher une exception.

Ce problème est sérieux, car le comportement du code ancien peut dépendre d'une garantie forte de sécurité de `push_back` face aux exceptions. Par conséquent, les implémentations de C++11 ne peuvent pas simplement remplacer les opérations de copie de `push_back` par des déplacements, à moins qu'il ne soit possible d'assurer que ces déplacements ne génèrent aucune exception. Dans ce cas, ils pourront remplacer les copies et le seul effet secondaire sera des performances améliorées.

`std::vector::push_back` exploite cette stratégie « de déplacement si c'est possible, mais de copie si c'est nécessaire », et ce n'est pas la seule fonction de la bibliothèque standard à procéder ainsi. C'est notamment le cas de celles qui offrent une garantie forte de sécurité vis-à-vis des exceptions en C++98 (comme `std::vector::reserve`, `std::deque::insert`, etc.). Elles remplacent toutes les appels aux opérations de copie en C++98 par des appels à des opérations de déplacement en C++11, mais uniquement si celles-ci sont certaines de ne pas générer d'exception. Mais, comment une fonction peut-elle savoir qu'une opération de déplacement ne produira pas d'exception ? Elle vérifie évidemment si l'opération est déclarée `noexcept`[1].

1. En général, le contrôle est plutôt indirect. Les fonctions comme `std::vector::push_back` appellent `std::move_if_noexcept`, c'est-à-dire une variante de `std::move` qui, sous condition, est convertie en une rvalue (voir le conseil 23), selon que le constructeur de déplacement du type est déclaré `noexcept` ou non. À son tour, `std::move_if_noexcept` consulte `std::is_nothrow_move_constructible` et la valeur de ce trait de type (voir le conseil 9) est fixée par

Copyright © 2016 Dunod.

Les fonctions `swap` constituent un autre cas où `noexcept` est particulièrement attrayant. L'échange est une opération essentielle dans de nombreux algorithmes de la STL et on le rencontre également très souvent dans les opérations d'affectation par copie. En raison de sa généralisation, les optimisations permises par `noexcept` deviennent extrêmement intéressantes. Notez que la déclaration `noexcept` des fonctions `swap` de la bibliothèque standard dépend parfois de la déclaration `noexcept` des fonctions `swap` définies par l'utilisateur. Voici, par exemple, les déclarations des `swap` pour les tableaux et les `std::pair` dans la bibliothèque standard :

```
template <class T, size_t N>
void swap(T (&a)[N],                                    // Voir
          T (&b)[N]) noexcept(noexcept(swap(*a, *b)));  // ci-après.

template <class T1, class T2>
struct pair {
  ...
  void swap(pair& p) noexcept(noexcept(swap(first, p.first)) &&
                              noexcept(swap(second, p.second)));
  ...
};
```

Ces fonctions sont déclarées avec un `noexcept` *conditionnel* : elles seront `noexcept` uniquement si les expressions indiquées dans les clauses de leur `noexcept` sont `noexcept`. Prenons par exemple deux tableaux de `Widget`. L'invocation de `swap` pour ces deux tableaux sera `noexcept` uniquement si l'échange par `swap` des éléments individuels des tableaux est `noexcept`, autrement dit, si `swap` pour `Widget` est `noexcept`. L'auteur de la fonction `swap` de `Widget` détermine donc si l'échange de tableaux de `Widget` est `noexcept`. Ce statut détermine si d'autres `swap`, comme l'échange de tableaux de tableaux de `Widget`, sont aussi `noexcept`. De la même manière, l'échange de deux objets `std::pair` qui contiennent des `Widget` sera `noexcept` si `swap` pour des `Widget` est `noexcept`. Vous le constatez, l'échange de structures de données de haut niveau ne sera en général `noexcept` que si l'échange de leurs composants de niveau inférieur est `noexcept`. Cela devrait vous inciter à proposer des fonctions `swap` `noexcept` dès que vous le pouvez.

Nous espérons que vous êtes à présent enthousiasmé par les opportunités d'optimisation offertes par `noexcept`. Nous allons malheureusement vous décevoir. Si l'optimisation est importante, l'exactitude l'est plus encore. Nous avons indiqué au début de ce conseil que `noexcept` fait partie de l'interface d'une fonction, qui pourra être déclarée `noexcept` uniquement si nous sommes prêts à garder son implémentation `noexcept` sur le long terme. Si nous déclarons une fonction `noexcept` et regrettons ensuite ce choix, les conséquences sont peu réjouissantes. Nous pouvons retirer `noexcept` de la déclaration de la fonction (c'est-à-dire changer son interface), mais les risques de dysfonctionnement dans le code client ne sont pas négligeables. Nous pouvons modifier l'implémentation de façon qu'une exception puisse être générée,

---

le compilateur, selon que le constructeur de déplacement a une désignation `noexcept` (ou `throw()`) ou non.

Copyright © 2016 Dunod.
© Dunod – Toute reproduction non autorisée est un délit.

tout en gardant la spécification d'exception d'origine (mais alors inexacte). Dans ce cas, le programme sera terminé si une exception sort de la fonction. Nous pouvons également nous résigner à conserver la version existante, en oubliant ce qui nous avait initialement conduit à vouloir modifier l'implémentation. Aucune de ces options n'est séduisante.

Le fait est que la plupart des fonctions sont *neutres envers les exceptions*. Elles ne lancent aucune exception elles-mêmes, mais les fonctions qu'elles appellent peuvent en générer. Lorsque cela se produit, la fonction neutre permet à l'exception qui a été levée de remonter la chaîne des appels jusqu'à un gestionnaire qui saura la traiter. Les fonctions neutres envers les exceptions ne sont jamais `noexcept`, car elles peuvent émettre des exceptions « qui ne font que passer ». Par conséquent, la désignation `noexcept` est absente de la plupart des fonctions.

Cependant, certaines fonctions ont des implémentations naturelles qui ne génèrent aucune exception. Pour quelques autres, comme les opérations de déplacement et `swap`, être `noexcept` apporte un tel bénéfice qu'il vaut la peine de les mettre en œuvre de manière `noexcept`, si tant est que ce soit possible[1]. Lorsque nous pouvons affirmer qu'une fonction ne produira jamais d'exception, nous devons la déclarer `noexcept`.

Nous avons indiqué que certaines fonctions ont une implémentation `noexcept` *naturelle*. Détourner la mise en œuvre d'une fonction afin de permettre une déclaration `noexcept`, c'est un peu le monde à l'envers. C'est placer la charrue avant les bœufs. C'est l'arbre qui cache la forêt. Choisissez votre métaphore préférée... Si l'implémentation normale d'une fonction peut lancer des exceptions (par exemple en invoquant une fonction qui peut elle-même en lever), tout le travail que nous devrons effectuer pour masquer ce fait au code appelant (par exemple intercepter toutes les exceptions et les remplacer par des codes d'état ou des valeurs de retour particulières) va non seulement compliquer l'implémentation de la fonction mais également le code appelant. Par exemple, il faudra que celui-ci vérifie les codes d'état ou les valeurs de retour spéciales. Le coût d'exécution de ces complications (par exemple des branchements supplémentaires, des fonctions plus longues qui sollicitent énormément les caches d'instructions, etc.) peut remettre en question l'amélioration des performances que nous pensions obtenir grâce à `noexcept`, sans oublier que le code source risque d'être plus difficile à comprendre et à maintenir. Voilà une ingénierie logicielle plutôt médiocre.

Il est tellement important que certaines fonctions soient `noexcept` qu'elles le sont par défaut. En C++98, les bonnes pratiques interdisaient aux fonctions de libération de la mémoire (c'est-à-dire `operator delete` et `operator delete[]`) et aux destructeurs de générer des exceptions. En C++11, cette règle de style est devenue une règle

1. Les interfaces des opérations de déplacement sur les conteneurs de la bibliothèque standard ne sont pas spécifiées `noexcept`. Toutefois, les programmeurs peuvent renforcer les spécifications d'exceptions sur les fonctions de la bibliothèque standard et, en pratique, il est courant qu'au moins quelques opérations de déplacement sur les conteneurs soient déclarées `noexcept`. Cette pratique illustre parfaitement le conseil prodigué ici. En ayant découvert que des opérations de déplacement sur les conteneurs pouvaient être écrites sans déclencher des exceptions, les programmeurs les déclarent souvent `noexcept` même si la norme ne les y oblige pas.

Copyright © 2016 Dunod.

du langage. Par défaut, toutes les fonctions de désallocation de la mémoire et les destructeurs, qu'ils soient définis par l'utilisateur ou générés par le compilateur, sont implicitement `noexcept`. Il est donc inutile de les déclarer `noexcept` (le contraire ne fera pas de mal mais sera juste peu conventionnel). Il existe un seul cas où un destructeur n'est pas implicitement `noexcept` : lorsqu'une donnée membre de la classe (y compris les membres hérités et ceux contenus à l'intérieur d'autres données membres) a un type qui stipule expressément que son destructeur peut générer des exceptions (par exemple le déclare « `noexcept(false)` »). De tels destructeurs sont plutôt rares. La bibliothèque standard n'en contient aucun et si le destructeur d'un objet utilisé par cette bibliothèque (par exemple, parce qu'il se trouve dans un conteneur ou qu'il a été passé à un algorithme) génère une exception, le comportement du programme est indéfini.

Il est bon de savoir que certains concepteurs d'interfaces de bibliothèques font une différence entre les fonctions ayant des *contrats étendus* et celles ayant des *contrats restreints*. Une fonction avec un contrat étendu ne fixe aucune condition préalable. Elle peut être appelée quel que soit l'état du programme et elle n'impose aucune contrainte sur les arguments transmis par l'appelant[1]. Ces fonctions n'ont jamais un comportement indéfini.

Les fonctions qui n'ont pas de contrat étendu ont un contrat restreint. Dans ce cas, si une condition préalable n'est pas respectée, le résultat de l'appel à la fonction n'est pas défini.

Si nous développons une fonction avec un contrat étendu et savons qu'elle ne générera aucune exception, il est facile de suivre le présent conseil et de la déclarer `noexcept`. Le cas d'une fonction avec un contrat restreint est plus complexe. Supposons, par exemple, que nous écrivions une fonction `f` qui prend un `std::string` en paramètre et supposons que son implémentation naturelle ne déclenche jamais d'exception. Tout est réuni pour que `f` soit déclarée `noexcept`.

Supposons à présent que `f` définisse une précondition : la longueur du paramètre `std::string` ne doit pas dépasser 32 caractères. Si `f` est appelée avec un `std::string` de longueur supérieure, son comportement est indéfini car, *par définition*, le non-respect d'une précondition conduit à un comportement indéfini. `f` n'est pas obligé de vérifier la longueur du paramètre, car les fonctions peuvent supposer que leurs préconditions sont remplies. (C'est à l'appelant de respecter les hypothèses.) Même avec une précondition, il semble approprié de déclarer `f noexcept` :

```
void f(const std::string& s) noexcept;      // Précondition :
                                            // s.length() <= 32.
```

1. « Quel que soit l'état du programme » et « aucune contrainte » ne légitiment en aucun cas les programmes dont le comportement est déjà indéfini. Par exemple, `std::vector::size` a un contrat étendu, mais cela ne veut pas dire que son comportement sera correct si elle est invoquée avec une zone de mémoire quelconque qui a été convertie de façon forcée en un `std::vector`. Le résultat de la conversion de type est indéfini et il n'y a aucune garantie de comportement pour le programme qui effectue cette conversion.

Copyright © 2016 Dunod.
© Dunod – Toute reproduction non autorisée est un délit.

Mais supposons que le développeur de `f` décide de contrôler le respect de la précondition. Cette vérification n'est pas obligatoire, mais elle n'est pas interdite et peut même être utile, par exemple pendant les tests du système. Il est en général plus facile de déboguer une exception qui a été lancée que d'essayer de découvrir l'origine d'un comportement indéfini. Comment peut-on signaler le non-respect d'une précondition afin que l'outil de test ou le gestionnaire d'erreurs du code client puisse le détecter ? Une solution simple serait de lancer une exception « non-respect d'une précondition », mais, si `f` est déclarée `noexcept`, cela n'est pas possible. En effet, lancer une exception conduirait alors à la terminaison du programme. C'est pourquoi les concepteurs de bibliothèques qui distinguent les contrats étendus et les contrats restreints réservent généralement `noexcept` aux fonctions qui ont un contrat étendu.

Pour finir, revenons sur le fait que le compilateur n'apporte en général aucune aide pour l'identification des incohérences entre l'implémentation d'une fonction et sa spécification d'exception. Examinons le code suivant, qui est parfaitement valide :

```
void setup();           // Fonctions définies ailleurs.
void cleanup();

void doWork() noexcept
{
  setup();              // Mettre en place les actions à réaliser.

  …                     // Effectuer les actions.

  cleanup();            // Faire le ménage après les actions.
}
```

Bien qu'elle appelle les fonctions `setup` et `cleanup` qui ne sont pas spécifiées `noexcept`, `doWork` est déclarée `noexcept`. Cela peut sembler contradictoire, mais il est possible que la documentation de `setup` et de `cleanup` précise qu'elles ne lancent jamais d'exception même si elles ne sont pas déclarées `noexcept`. Il peut y avoir de bonnes raisons à cela, par exemple le fait qu'elles se trouvent dans une bibliothèque écrite en C. (Même des fonctions de la bibliothèque standard C qui ont été déplacées dans l'espace de noms `std` n'ont pas de spécifications d'exception. Par exemple, `std::strlen` n'est pas déclarée `noexcept`.) Elles peuvent également faire partie d'une bibliothèque C++98 pour laquelle il avait été décidé de ne pas utiliser les spécifications d'exceptions de C++98 et qui n'a pas été revue pour C++11.

Puisque les fonctions `noexcept` peuvent avoir de bonnes raisons de se fonder sur du code sans garantie `noexcept`, C++ autorise l'écriture d'un tel code et les compilateurs n'y voient généralement rien à redire.

À retenir

- `noexcept` fait partie de l'interface d'une fonction et le code appelant peut donc en dépendre.
- Les fonctions `noexcept` présentent de plus grandes possibilités d'optimisation que les fonctions non `noexcept`.

Copyright © 2016 Dunod.

- `noexcept` est particulièrement intéressant avec les opérations de déplacement, `swap`, les fonctions de désallocation de la mémoire et les destructeurs.
- La plupart des fonctions affichent une neutralité envers les exceptions plutôt que `noexcept`.

CONSEIL N° 15. UTILISER `CONSTEXPR` DÈS QUE POSSIBLE

Si nous devions élire le nouveau mot clé le plus déroutant de C++11, `constexpr` serait probablement le vainqueur. Appliqué à des objets, il s'agit essentiellement d'une version renforcée de `const`. En revanche, appliqué à des fonctions, sa signification est assez différente. Nous allons couper court à toute confusion car, lorsque `constexpr` correspond à ce que nous voulons exprimer, il devient indispensable.

Conceptuellement, `constexpr` indique non seulement qu'une valeur est une constante mais aussi qu'elle est connue au moment de la compilation. Néanmoins, le concept n'est pas tout car, lorsque `constexpr` est appliquée aux fonctions, les choses sont plus nuancées. De peur de dévoiler la scène finale, nous allons pour le moment nous contenter de préciser qu'il ne faut pas supposer que les résultats des fonctions `constexpr` sont des `const`, ni que leurs valeurs sont connues au moment de la compilation. Plus étrangement, ces deux aspects sont des *fonctionnalités*. En réalité, il est *préférable* que les fonctions `constexpr` ne soient pas obligées de produire des résultats `const` ou connus à la compilation !

Commençons tout d'abord par les objets `constexpr`. De tels objets sont bien `const` et ils ont bien des valeurs connues au moment de la compilation. (D'un point de vue technique, leurs valeurs sont déterminées au cours de la *traduction*, qui regroupe la compilation et l'édition de liens. Cependant, à moins que vous ne développiez des compilateurs ou des éditeurs de liens pour C++, cela ne vous concerne pas et vous pouvez continuer à écrire vos programmes comme si les valeurs des objets `constexpr` étaient déterminées au moment de la compilation.)

Les valeurs connues pendant la compilation bénéficient de privilèges. Par exemple, elles peuvent être placées dans une zone de mémoire en lecture seule, ce qui, pour les développeurs de systèmes embarqués notamment, peut avoir une importance considérable. Dans un champ d'application plus large, les valeurs entières qui sont constantes et connues à la compilation peuvent être employées dans tous les contextes où C++ a besoin d'une *expression constante entière*. Il s'agit notamment de la spécification de la taille d'un tableau, d'un argument entier d'un template (y compris la dimension d'un objet `std::array`), des valeurs d'un énumérateur, des spécificateurs d'alignement, etc. Si nous souhaitons employer une variable dans toutes ces utilisations, nous la déclarerons avec `constexpr` car le compilateur s'assurera ensuite qu'elle a une valeur pendant la compilation :

```
int sz;                          // Variable non constexpr.

...
```

Copyright © 2016 Dunod.
© Dunod – Toute reproduction non autorisée est un délit.

```
constexpr auto arraySize1 = sz;      // Erreur ! La valeur de sz n'est
                                     // pas connue à la compilation.

std::array<int, sz> data1;           // Erreur ! Problème identique.

constexpr auto arraySize2 = 10;      // Parfait, 10 est une constante
                                     // connue à la compilation.

std::array<int, arraySize2> data2;   // Parfait, arraySize2 est déclarée
                                     // constexpr.
```

Notez que `const` n'offre pas la même garantie que `constexpr`. En effet, les objets `const` ne sont pas nécessairement initialisés avec des valeurs connues à la compilation :

```
int sz;                              // Comme précédemment.

...

const auto arraySize = sz;           // Parfait, arraySize est une copie
                                     // constant de sz.

std::array<int, arraySize> data;     // Erreur ! La valeur de arraySize
                                     // n'est pas connue à la compilation.
```

Pour faire simple, tous les objets `constexpr` sont des `const`, mais tous les objets `const` ne sont pas des `constexpr`. Si nous voulons que le compilateur garantisse qu'une variable possède une valeur pouvant être utilisée dans des contextes qui exigent des constantes connues à la compilation, nous devons employer non pas `const` mais `constexpr`.

Les scénarios d'utilisation des objets `constexpr` deviennent plus intéressants lorsque des fonctions `constexpr` entrent en scène. De telles fonctions produisent des constantes connues à la compilation *lorsqu'elles sont appelées avec des constantes connues à la compilation*. Si elles sont appelées avec des valeurs connues uniquement au moment de l'exécution, elles produisent des valeurs connues à l'exécution. Vous pourriez penser que cela revient à ignorer ce qu'elles feront, mais cette réflexion est erronée. Voici la bonne vision :

- Les fonctions `constexpr` peuvent être employées dans les contextes qui demandent des constantes connues à la compilation. Si les valeurs des arguments transmis à une fonction `constexpr` employée dans un tel contexte sont connues à la compilation, le résultat sera déterminé pendant la compilation. Si l'une des valeurs des arguments n'est pas connue à la compilation, le code sera refusé.
- Lorsqu'une fonction `constexpr` est appelée avec une ou plusieurs valeurs inconnues au moment de la compilation, elle se comporte comme une fonction normale, en déterminant son résultat au cours de l'exécution. Nous n'avons donc pas besoin de deux fonctions, l'une pour des constantes connues à la compilation et l'autre pour les autres valeurs, pour effectuer la même opération. La seule fonction `constexpr` suffit.

Copyright © 2016 Dunod.

Supposons que nous ayons besoin d'une structure de données pour mémoriser les résultats d'une expérience qui peut être menée sous différentes conditions. Par exemple, le niveau d'éclairage peut être élevé, faible ou nul au cours de l'expérience, tout comme la vitesse du ventilateur, la température, etc. S'il existe `n` conditions environnementales pertinentes pour l'expérience, chacune avec trois états possibles, le nombre de combinaisons est égal à 3^n. Pour stocker les résultats obtenus avec toutes ces combinaisons, il nous faut une structure de données avec une place suffisante pour 3^n valeurs. En supposant que chaque résultat soit un `int` et que `n` est connu (ou puisse être calculé) à la compilation, un `std::array` peut être une structure de données convenable. Nous devons cependant calculer 3^n pendant la compilation. La bibliothèque standard de C++ fournit `std::pow`, qui correspond à la fonction mathématique requise, mais, dans notre cas, elle pose deux problèmes. Premièrement, `std::pow` manipule des types à virgule flottante alors que nous avons besoin d'un résultat entier. Deuxièmement, `std::pow` n'est pas `constexpr` (autrement dit, rien ne garantit qu'elle retournera un résultat au moment de la compilation si elle est appelée avec une valeur connue à la compilation). Nous ne pouvons donc pas l'employer pour fixer la taille d'un `std::array`.

Heureusement, nous sommes capables d'écrire la fonction `pow` dont nous avons besoin, mais nous la montrerons plus loin. Commençons par présenter sa déclaration et son utilisation :

```
constexpr                                   // pow est une fonction constexpr
int pow(int base, int exp) noexcept         // qui ne lève pas d'exception.
{
  ...                                       // Implémentation ci-après.
}

constexpr auto numConds = 5;                      // Nombre de conditions.

std::array<int, pow(3, numConds)> results;        // results comprend
                                                  // 3^numConds éléments.
```

Rappelons que le mot clé `constexpr` placé devant `pow` signifie non pas que cette fonction retourne une valeur `const`, mais que si `base` et `exp` sont des constantes connues à la compilation, le résultat de `pow` peut être utilisé comme une constante au moment de la compilation. Si `base` et/ou `exp` ne sont pas des constantes connues à la compilation, le résultat de `pow` sera déterminé à l'exécution. Autrement dit, la fonction `pow` peut non seulement être appelée à la compilation pour effectuer des opérations comme calculer la taille d'un `std::array`, mais également pendant l'exécution :

```
auto base = readFromDB("base");         // Obtenir ces valeurs
auto exp = readFromDB("exponent");      // à l'exécution.

auto baseToExp = pow(base, exp);        // Appeler la fonction pow
                                        // à l'exécution.
```

Puisque les fonctions `constexpr` doivent être en mesure de renvoyer des résultats pendant la compilation lorsqu'elles sont appelées avec des valeurs connues à ce

Copyright © 2016 Dunod.
© Dunod – Toute reproduction non autorisée est un délit.

moment-là, leur implémentation est soumise à plusieurs contraintes. Ces restrictions diffèrent entre C++11 et C++14.

En C++11, les fonctions `constexpr` ne doivent contenir qu'une seule instruction exécutable : un `return`. Cette contrainte n'est pas trop forte, car deux astuces permettent d'étendre les possibilités d'expression dans les fonctions `constexpr`. Premièrement, l'opérateur conditionnel « `?:` » peut remplacer des instructions `if-else` et, deuxièmement, la récursion peut remplacer les boucles. Voici donc une manière d'implémenter `pow` :

```
constexpr int pow(int base, int exp) noexcept
{
  return (exp == 0 ? 1 : base * pow(base, exp - 1));
}
```

Ce code fonctionne, mais il est difficile d'imaginer qu'un programmeur autre qu'un amateur pur et dur de la programmation fonctionnelle puisse l'apprécier. En C++14, les restrictions sur les fonctions `constexpr` sont moindres et l'implémentation suivante devient possible :

```
constexpr int pow(int base, int exp) noexcept       // C++14.
{
  auto result = 1;
  for (int i = 0; i < exp; ++i) result *= base;

  return result;
}
```

Les fonctions `constexpr` ont pour obligation de prendre et de retourner des *littéraux*, autrement dit des types dont il est possible de déterminer la valeur au moment de la compilation. En C++11, c'est le cas de tous les types intégrés, à l'exception de `void`. Les types définis par l'utilisateur peuvent également être des littéraux, car les constructeurs et d'autres fonctions membres peuvent être `constexpr` :

```
class Point {
public:
  constexpr Point(double xVal = 0, double yVal = 0) noexcept
  : x(xVal), y(yVal)
  {}

  constexpr double xValue() const noexcept { return x; }
  constexpr double yValue() const noexcept { return y; }

  void setX(double newX) noexcept { x = newX; }
  void setY(double newY) noexcept { y = newY; }

private:
  double x, y;
};
```

Copyright © 2016 Dunod.

Dans cet exemple, le constructeur de `Point` peut être déclaré `constexpr` car, si les arguments qui lui sont passés sont connus au moment de la compilation, les valeurs des données membres de l'objet `Point` construit peuvent également être connues à ce moment-là. Les `Point` initialisés de cette manière peuvent donc être `constexpr` :

```
constexpr Point p1(9.4, 27.7);   // Parfait, "exécuter" le constructeur
                                 // constexpr à la compilation.

constexpr Point p2(28.8, 5.3);   // Également parfait.
```

De même, les accesseurs `xValue` et `yValue` peuvent être `constexpr`, car si ces méthodes sont invoquées sur un objet `Point` avec une valeur connue à la compilation (par exemple un objet `constexpr Point`), les valeurs des données membres `x` et `y` sont également connues pendant cette phase. Il est donc possible d'écrire des fonctions `constexpr` qui invoquent les accesseurs de `Point` et d'initialiser des objets `constexpr` avec les résultats obtenus :

```
constexpr
Point midpoint(const Point& p1, const Point& p2) noexcept
{
  return { (p1.xValue() + p2.xValue()) / 2,    // Appeler les fonctions
           (p1.yValue() + p2.yValue()) / 2 };  // membres constexpr.
}

constexpr auto mid = midpoint(p1, p2);       // Initialiser un objet
                                             // constexpr avec le résultat
                                             // d'une fonction constexpr.
```

Tout cela est très intéressant. En effet, même si l'initialisation de l'objet `mid` implique des appels à des constructeurs, des accesseurs et une fonction non membre, il peut être créé dans une zone de mémoire en lecture seule ! Nous pouvons donc employer une expression comme `mid.xValue() * 10` dans un argument de template ou dans une expression qui précise la valeur d'un énumérateur[1] ! Par ailleurs, la démarcation habituellement nette entre le travail effectué à la compilation et celui effectué au cours de l'exécution devient plus floue. Certains calculs traditionnellement effectués pendant l'exécution peuvent à présent être réalisés à la compilation. Plus la quantité de code concernée par ce changement sera importante, plus l'exécution du programme sera rapide. (En revanche, la compilation risque de prendre plus de temps.)

En C++11, deux contraintes empêchent de déclarer `constexpr` les fonctions membres `setX` et `setY` de `Point`. Premièrement, elles modifient l'objet qu'elles manipulent et, en C++11, les fonctions membres `constexpr` sont implicitement `const`.

1. Puisque `Point::xValue` renvoie un `double`, `mid.xValue() * 10` est aussi de type `double`. Les nombres à virgule flottante ne peuvent pas servir à instancier des templates ni à spécifier les valeurs d'un énumérateur, mais ils peuvent être intégrés à des expressions plus longues qui impliquent des entiers. Par exemple, nous pouvons employer `static_cast<int>(mid.xValue() * 10)` pour instancier un template ou fixer la valeur d'un énumérateur.

© Dunod – Toute reproduction non autorisée est un délit.
Copyright © 2016 Dunod.

Deuxièmement, elles spécifient un type de retour `void`, qui n'est pas un littéral en C++11. Puisque ces deux restrictions sont levées en C++14, dans cette version du langage même les mutateurs de `Point` peuvent être `constexpr` :

```
class Point {
public:
  …

  constexpr void setX(double newX) noexcept     // C++14.
  { x = newX; }

  constexpr void setY(double newY) noexcept     // C++14.
  { y = newY; }

  …

};
```

Cela nous autorise à écrire des fonctions telles que la suivante :

```
// Retourner l'opposé de p par rapport à l'origine (C++14).
constexpr Point reflection(const Point& p) noexcept
{
  Point result;                         // Créer un Point non const.

  result.setX(-p.xValue());             // Fixer ses valeurs x et y.
  result.setY(-p.yValue());

  return result;                        // En renvoyer une copie.
}
```

Voici un exemple de code client :

```
constexpr Point p1(9.4, 27.7);          // Comme précédemment.
constexpr Point p2(28.8, 5.3);
constexpr auto mid = midpoint(p1, p2);

constexpr auto reflectedMid =           // La valeur de reflectedMid
  reflection(mid);                      // est (-19.1 -16.5) et connue
                                        // à la compilation.
```

Ce conseil recommande d'utiliser `constexpr` dès que c'est possible et nous espérons que vous comprenez à présent pourquoi : les objets et les fonctions `constexpr` peuvent être employés dans un plus grand nombre de contextes que les objets et les fonctions non `constexpr`.

Il est important de noter que `constexpr` fait partie de l'interface d'un objet d'une fonction. Ce mot clé signifie « je peux être utilisé dans tout contexte où C++ a besoin d'une expression constante ». Si nous déclarons un objet ou une fonction `constexpr`, le code client peut l'employer dans de tels contextes. Si nous décidons ensuite que la déclaration `constexpr` était une erreur et que nous la supprimons, il est possible que de grandes quantités de code client ne compilent plus. (Le simple ajout d'entrées-sorties

Copyright © 2016 Dunod.

à une fonction pour le débogage ou le réglage des performances peut conduire un tel problème, car les instructions d'entrées-sorties sont généralement interdites dans les fonctions `constexpr`.) C'est à vous de décider si, en utilisant `constexpr`, vous acceptez de respecter sur le long terme les contraintes que cela impose sur les objets et les fonctions concernés.

À retenir

- Les objets `constexpr` sont `const` et sont initialisés avec des valeurs connues au moment de la compilation.
- Les fonctions `constexpr` peuvent générer des résultats pendant la compilation si elles sont appelées avec des arguments dont les valeurs sont connues à ce moment-là.
- Les objets et les fonctions `constexpr` sont utilisables dans un éventail de contextes plus large que les objets et les fonctions non `constexpr`.
- `constexpr` fait partie de l'interface de l'objet ou de la fonction.

CONSEIL N° 16. RENDRE LES FONCTIONS MEMBRES `CONST` SÛRES VIS-À-VIS DES THREADS

Les personnes qui travaillent dans un domaine mathématique pourraient trouver commode de disposer d'une classe qui représente des polynômes. Elle pourrait offrir une fonction qui calcule les racines d'un polynôme, c'est-à-dire les valeurs pour lesquelles l'évaluation du polynôme est égale à zéro. Puisque cette fonction ne modifierait pas le polynôme, il serait naturel de la déclarer `const` :

```
class Polynomial {
public:
  using RootsType =           // Structure de données qui contient
    std::vector<double>;      // les valeurs pour lesquelles le
  …                           // polynôme est égal à 0 (voir le
                              // conseil 9 pour des infos sur "using").

  RootsType roots() const;

  …

};
```

Déterminer les racines d'un polynôme peut être un calcul lourd, qui ne doit donc pas être effectué inutilement. S'il doit être lancé, il est préférable de ne pas le répéter plusieurs fois. Nous allons donc placer les racines du polynôme dans un cache après leur calcul et nous allons implémenter `roots` de façon qu'elle retourne ces valeurs en cache. Voici l'approche de base :

Copyright © 2016 Dunod.
© Dunod – Toute reproduction non autorisée est un délit.

```
class Polynomial {
public:
  using RootsType = std::vector<double>;

  RootsType roots() const
  {
    if (!rootsAreValid) {            // Si le cache est invalide

      …                              // déterminer les racines, et
                                     // les mémoriser dans rootVals.
      rootsAreValid = true;
    }

    return rootVals;
  }

private:
  mutable bool rootsAreValid{ false };    // Voir le conseil 7 pour des
  mutable RootsType rootVals{};           // infos sur les initialiseurs.
};
```

Conceptuellement, `roots` ne modifie pas l'objet `Polynomial` qu'elle manipule, mais, dans sa gestion du cache, elle peut modifier `rootVals` et `rootsAreValid`. Il s'agit d'un cas classique d'utilisation de `mutable` et c'est pourquoi nous l'utilisons dans la déclaration de ces données membres.

Imaginons à présent que deux threads appellent simultanément `roots` sur un objet `Polynomial` :

```
Polynomial p;

…

/*- - - - -  Thread 1  - - - - - */     /*- - - - -  Thread 2  - - - - - */

auto rootsOfP = p.roots();              auto valsGivingZero = p.roots();
```

Ce code client est parfaitement envisageable. `roots` est une fonction membre `const`, qui représente donc une opération de lecture. Plusieurs threads peuvent effectuer en toute sécurité une opération de lecture sans synchronisation. Tout au moins, c'est ainsi que cela devrait se passer. Ce n'est pas le cas de notre exemple, car, dans la méthode `roots`, l'un ou l'autre de ces threads peut essayer de modifier les données membres `rootsAreValid` et `rootVals`. Autrement dit, ce code peut présenter des threads différents qui lisent et écrivent la même zone de mémoire sans synchronisation ; voilà la définition d'une situation de concurrence sur les données. Ce code a donc un comportement indéfini.

Le problème vient du fait que `roots` est déclarée `const` sans qu'elle soit sûre vis-à-vis des threads. Puisque la déclaration `const` est aussi correcte en C++11 qu'elle le serait en C++98 (obtenir les racines d'un polynôme ne change pas la valeur du polynôme), la rectification doit se faire au niveau de la sécurité vis-à-vis des threads.

Copyright © 2016 Dunod.

Pour résoudre ce problème, l'approche classique consiste à mettre en place un `mutex` :

```
class Polynomial {
public:
  using RootsType = std::vector<double>;

  RootsType roots() const
  {
    std::lock_guard<std::mutex> g(m);   // Verrouiller le mutex.

    if (!rootsAreValid) {               // Si le cache est invalide

      …                                 // calculer/mémoriser les racines.

      rootsAreValid = true;
    }

    return rootVals;
  }                                     // Libérer le mutex.

private:
  mutable std::mutex m;
  mutable bool rootsAreValid{ false };
  mutable RootsType rootVals{};
};
```

Le `std::mutex m` est déclaré `mutable`, car son verrouillage et sa libération se font avec des fonctions membres non `const` et, dans `roots` (une fonction membre `const`), `m` serait sinon considéré comme un objet `const`.

Notez que `std::mutex` étant un *type réservé au déplacement* (c'est-à-dire un type qui accepte les déplacements mais pas les copies), l'ajout de `m` à `Polynomial` conduit cette classe à perdre sa faculté à être copiée. En revanche, elle peut toujours être déplacée.

Dans certains cas, la solution du mutex est exagérée. Par exemple, si nous comptons simplement le nombre d'appels à une fonction membre, un compteur `std::atomic` (c'est-à-dire un compteur dont les opérations sont vues par d'autres threads comme indivisibles ; voir le conseil 40) sera souvent plus économe. (Le coût réel de cette solution dépend du matériel utilisé et de l'implémentation des mutex dans la bibliothèque standard.) Voici comment se servir d'un `std::atomic` pour compter des appels :

```
class Point {                                   // Point en 2D.
public:
  …

  double distanceFromOrigin() const noexcept    // Voir le conseil 14
  {                                             // pour noexcept.

    ++callCount;                                // Incrémentation atomique.

    return std::sqrt((x * x) + (y * y));
```

© Dunod – Toute reproduction non autorisée est un délit.
Copyright © 2016 Dunod.

```
  }

private:
  mutable std::atomic<unsigned> callCount{ 0 };
  double x, y;
};
```

À l'instar des `std::mutex`, les `std::atomic` peuvent uniquement être déplacés. La présence de `callCount` dans `Point` signifie donc que `Point` est un type réservé au déplacement.

Puisque les opérations sur les variables `std::atomic` sont souvent moins onéreuses que l'acquisition et la libération d'un mutex, nous pourrions être tentés de compter sur les `std::atomic` bien plus qu'il ne le faudrait. Par exemple, dans une classe qui met en cache un `int` long à calculer, nous pourrions imaginer utiliser deux variables `std::atomic` à la place d'un mutex :

```
class Widget {
public:
  ...

  int magicValue() const
  {
    if (cacheValid) return cachedValue;
    else {
      auto val1 = expensiveComputation1();
      auto val2 = expensiveComputation2();
      cachedValue = val1 + val2;                    // Houlà, partie 1.
      cacheValid = true;                            // Houlà, partie 2.
      return cachedValue;
    }
  }

private:
  mutable std::atomic<bool> cacheValid{ false };
  mutable std::atomic<int> cachedValue;
};
```

Ce code fonctionne, mais il travaille parfois plus qu'il ne le devrait. Examinons la situation suivante :

- Un thread appelle `Widget::magicValue`, voit que `cacheValid` est `false`, effectue deux calculs intensifs et affecte la somme de leurs résultats à `cachedValue`.
- À ce stade, un second thread appelle `Widget::magicValue`, voit également que `cacheValid` est `false` et réalise donc les mêmes calculs que le premier thread vient de terminer. (Ce « second thread » pourrait en réalité correspondre à *plusieurs* autres threads.)

Un tel fonctionnement va à l'encontre de l'objectif d'un cache. Inverser l'ordre des affectations de `cachedValue` et de `CacheValid` supprime ce problème, mais le résultat est pire encore :

Copyright © 2016 Dunod.

```
class Widget {
public:
  …

  int magicValue() const
  {
    if (cacheValid) return cachedValue;
    else {
      auto val1 = expensiveComputation1();
      auto val2 = expensiveComputation2();
      cacheValid = true;                          // Houlà, partie 1.
      return cachedValue = val1 + val2;           // Houlà, partie 2.
    }
  }

  …

};
```

Imaginons que `cacheValid` soit `false` et qu'ensuite :

- Un thread appelle `Widget::magicValue` et poursuit son exécution jusqu'à la ligne où `cacheValid` est fixé à `true`.
- À ce moment-là, un second thread appelle `Widget::magicValue` et vérifie `cacheValid`. Puisqu'il vaut `true`, le thread renvoie `cachedValue`, même si le premier thread n'a pas encore fixé sa valeur. La valeur retournée est donc incorrecte.

Voici la leçon à retenir. Lorsqu'une seule variable ou zone de mémoire a besoin d'une synchronisation, il est possible d'employer un `std::atomic`. En revanche, lorsque deux variables ou zones de mémoire sont impliquées et doivent être manipulées comme une seule unité, il faut opter pour un mutex. Appliquons ce principe à `Widget::magicValue` :

```
class Widget {
public:
  …

  int magicValue() const
  {
    std::lock_guard<std::mutex> guard(m);      // Verrouiller m.

    if (cacheValid) return cachedValue;
    else {
      auto val1 = expensiveComputation1();
      auto val2 = expensiveComputation2();
      cachedValue = val1 + val2;
      cacheValid = true;
      return cachedValue;
    }
  }                                            // Libérer m.
  …
```

Copyright © 2016 Dunod.
© Dunod – Toute reproduction non autorisée est un délit.

```
private:
  mutable std::mutex m;
  mutable int cachedValue;                    // N'est plus atomique.
  mutable bool cacheValid{ false };           // N'est plus atomique.
};
```

Ce conseil se fonde sur l'hypothèse que plusieurs threads peuvent invoquer simultanément une fonction membre `const` sur un objet. Si nous écrivons une fonction membre `const` pour laquelle ce n'est pas le cas – pour laquelle nous pouvons *garantir* que jamais plusieurs threads ne l'invoqueront sur un même objet –, sa sécurité vis-à-vis des threads n'a pas d'importance. Par exemple, les fonctions membres d'une classe conçue pour une utilisation exclusivement monothread n'ont pas besoin d'être sûres vis-à-vis des threads. Dans ce cas, nous pouvons éviter les coûts associés aux mutex et aux `std::atomic`, et oublier que les classes qui les utilisent ne peuvent plus être copiées mais uniquement déplacées. Toutefois, de tels scénarios sont de moins en moins fréquents et vont même se raréfier. Il vaut mieux parier sur le fait que les fonctions membres `const` seront sujettes aux exécutions concurrentes et qu'elles doivent donc être sûres vis-à-vis des threads.

À retenir

- Les fonctions membres `const` doivent être sûres vis-à-vis des threads, sauf s'il est *certain* qu'elles ne seront jamais utilisées dans un contexte de concurrence.
- Les variables `std::atomic` pourront conduire à de meilleures performances qu'un mutex, mais elles ne conviennent que lors de la manipulation d'une seule variable ou zone de mémoire.

CONSEIL N° 17. COMPRENDRE LA GÉNÉRATION D'UNE FONCTION MEMBRE SPÉCIALE

Dans le jargon C++ officiel, les *fonctions membres spéciales* désignent les fonctions que le compilateur C++ est disposé à générer de lui-même. En C++98, ces fonctions sont au nombre de quatre : le constructeur par défaut, le destructeur, le constructeur de copie et l'opérateur d'affectation par copie. Elles sont générées uniquement en cas de besoin, c'est-à-dire si du code les utilise sans qu'elles soient déclarées explicitement dans la classe. Un constructeur par défaut est généré uniquement si la classe n'en déclare aucun. (Cela évite que le compilateur ne crée un constructeur par défaut alors que nous avons spécifié que le constructeur de la classe doit avoir des arguments.) Les fonctions membres spéciales générées sont implicitement publiques et `inline`. Elles ne sont pas virtuelles, sauf si la fonction en question est le destructeur d'une classe qui dérive d'une classe de base dont le destructeur est virtuel. Dans ce cas, le destructeur créé par le compilateur pour la classe dérivée est également virtuel.

Mais vous savez déjà tout cela ; c'est de l'histoire ancienne. Mais les temps ont changé et les règles de génération d'une fonction membre spéciale en C++ ont évolué. Il est important de les connaître, car savoir quand le compilateur insère discrètement

Copyright © 2016 Dunod.

des fonctions membres dans nos classes est essentiel à une programmation efficace en C++.

Depuis C++11, le club des fonctions spéciales comprend deux membres supplémentaires : le constructeur de déplacement et l'opérateur d'affectation par déplacement. Voici leur signature :

```
class Widget {
public:
  …
  Widget(Widget&& rhs);                // Constructeur de déplacement.

  Widget& operator=(Widget&& rhs);     // Opérateur d'affectation
                                       // par déplacement.
  …
};
```

Les règles qui gouvernent leur génération et comportement sont comparables à celles de leurs homologues pour la copie. Les opérations de déplacement sont générées uniquement si elles sont nécessaires et, lorsque c'est le cas, elles réalisent des « déplacements de niveau membre » sur les données membres non statiques de la classe. Autrement dit, le constructeur de déplacement construit par déplacement chaque donnée membre non statique de la classe à partir du membre correspondant dans son paramètre `rhs`, et l'opérateur d'affectation par déplacement effectue une affectation par déplacement pour chaque donnée membre non statique de son paramètre. Le constructeur de déplacement et l'opérateur d'affectation par déplacement traitent également les éventuels éléments de la classe de base.

Lorsque nous parlons d'opération de déplacement, de construction par déplacement ou d'affectation par déplacement d'une donnée membre ou d'une classe de base, rien ne garantit qu'un déplacement aura réellement lieu. Les « déplacements de niveau membre » sont en réalité des « demandes » de déplacement de niveau membre car les types qui ne sont pas compatibles avec le déplacement (autrement dit qui n'offrent aucune prise en charge particulière pour les opérations de déplacement, par exemple la plupart des classes anciennes de C++98) seront « déplacés » par des opérations de copie. Le « déplacement » de niveau membre se fait en appliquant `std::move` à l'objet qui sert de source et le résultat est utilisé pendant la résolution de la surcharge de fonction pour déterminer si une copie ou un déplacement doit être effectué. Le conseil 23 revient en détail sur ce processus. Pour le moment, il suffit de retenir qu'un déplacement de niveau membre correspond à des opérations de déplacement sur les données membres et sur les classes de base qui prennent en charge ces opérations, sinon il correspond à des opérations de copie.

À l'instar des opérations de copie, les opérations de déplacement ne sont pas générées si nous les déclarons nous-mêmes. En revanche, les conditions précises de leur génération automatique diffèrent légèrement.

Les deux opérations de copie sont indépendantes : déclarer l'une n'empêche pas le compilateur de générer l'autre. Par conséquent, si nous déclarons un constructeur de copie sans déclarer d'opérateur d'affectation par copie, puis écrivons du code qui a

Copyright © 2016 Dunod.
© Dunod – Toute reproduction non autorisée est un délit.

besoin de l'affectation par copie, le compilateur générera cet opérateur à notre place. Il en va de même pour la génération du constructeur de copie. Ce comportement était valide en C++98 et le reste en C++11.

En revanche, les deux opérations de déplacement ne sont pas indépendantes. Si nous déclarons l'une, le compilateur ne génère pas l'autre. En voici la raison : en déclarant, par exemple, un constructeur de déplacement pour notre classe, nous indiquons implicitement que l'implémentation de cette opération est différente du déplacement de niveau membre par défaut que le compilateur générerait. Si la construction par déplacement de niveau membre ne convient pas, il est probable que l'affectation par déplacement de niveau membre ne convienne pas non plus. C'est pourquoi déclarer un constructeur de déplacement empêche la génération d'un opérateur d'affectation par déplacement, et vice versa.

Par ailleurs, les opérations de déplacement ne seront pas générées si la classe déclare explicitement une opération de copie. En effet, déclarer une opération de copie (construction ou affectation) indique que la méthode normale de copie d'un objet (copie de niveau membre) n'est pas adaptée à la classe et le compilateur suppose donc que le déplacement de niveau membre n'est pas adapté aux opérations de déplacement.

Ce comportement est également valable dans l'autre sens. Déclarer une opération de déplacement (construction ou affectation) dans une classe conduit le compilateur à désactiver les opérations de copie. (La désactivation se fait en supprimant les opérations de copie ; voir le conseil 11.) En effet, si le déplacement de niveau membre ne convient pas au déplacement d'un objet, on peut supposer que la copie de niveau membre n'est pas la bonne manière de le copier. On pourrait penser que cela remet en cause le code C++98, car les conditions sous lesquelles les opérations de copie sont activées sont plus contraignantes en C++11 qu'en C++98, mais ce n'est pas le cas. Puisque la notion de déplacement d'objet n'existe pas en C++98, un code C++98 ne peut pas inclure des opérations de déplacement. Pour qu'une ancienne classe puisse offrir des opérations de déplacement déclarées par l'utilisateur, il faut les ajouter pour C++11 et cela doit se faire conformément aux règles C++11 de la génération d'une fonction membre spéciale.

Vous avez peut-être entendu parler de la recommandation dite de la *Règle des trois*. Elle explique que s'il nous faut déclarer un constructeur de copie, un opérateur d'affectation par copie ou un destructeur, nous devons déclarer les trois. Elle découle d'une observation : si nous avons besoin de modifier le sens d'une opération de copie, cela signifie presque toujours que la classe effectue une forme de gestion des ressources. Et cela implique presque toujours que (1) quelle que soit la gestion de ressources effectuée dans une opération de copie, elle devra probablement être réalisée dans l'autre opération de copie, et que (2) le destructeur de la classe doit également participer à cette gestion (habituellement libérer la ressource). La ressource gérée est souvent la mémoire et c'est pourquoi toutes les classes de la bibliothèque standard qui gèrent des zones de mémoire (par exemple les conteneurs STL qui effectuent une gestion dynamique de la mémoire) suivent cette Règle des trois : elles déclarent les deux opérations de copie et un destructeur.

Copyright © 2016 Dunod.

En conséquence de la Règle des trois, la présence d'un destructeur déclaré par l'utilisateur indique qu'une copie de niveau membre simple est probablement inadaptée aux opérations de copie dans la classe. Cela suggère donc que si une classe déclare un destructeur, il est fort probable que les opérations de copie qui seraient générées automatiquement n'auraient pas le fonctionnement approprié. Au moment de l'adoption de C++98, le sens d'un tel raisonnement n'était pas pleinement apprécié et l'existence d'un destructeur déclaré par l'utilisateur n'avait pas d'impact sur la volonté des compilateurs à générer des opérations de copie. Cela reste le cas en C++11, mais uniquement parce que le durcissement des conditions de génération des opérations de copie rendrait inopérante une trop grande quantité de code ancien.

Cependant, les fondements de la Règle des trois restent valides et, en y ajoutant le fait que la déclaration d'une opération de copie exclut la génération implicite des opérations de déplacement, nous avons les raisons pour que C++11 ne génère pas des opérations de déplacement lorsque la classe dispose d'un destructeur déclaré par l'utilisateur.

Les opérations de déplacement sont donc générées (si nécessaire) uniquement lorsque les trois conditions suivantes sont satisfaites :

- Aucune opération de copie n'est déclarée dans la classe.
- Aucune opération de déplacement n'est déclarée dans la classe.
- Aucun destructeur n'est déclaré dans la classe.

Il est possible que des règles analogues puissent un jour s'appliquer aux opérations de copie, car C++11 a rendu obsolète leur génération automatique lorsque la classe déclare des opérations de copie ou un destructeur. Par conséquent, si nous avons du code qui dépend de la génération des opérations de copie dans des classes qui déclarent un destructeur ou l'une des opérations de copie, nous devons les revoir afin de supprimer cette dépendance. En supposant que le comportement des fonctions générées par le compilateur soit correct (autrement dit que la copie de niveau membre des données membres non statiques de la classe corresponde aux besoins), notre travail reste simple en C++11. Il suffit d'ajouter « `= default` » pour l'exprimer explicitement :

```
class Widget {
public:
  ...
  ~Widget();                                  // Destructeur déclaré
  .                                           // par l'utilisateur.

  ...                                         // Le comportement du constructeur
  Widget(const Widget&) = default;            // de copie par défaut est OK.

  Widget&                                     // Le comportement de l'affectation
    operator=(const Widget&) = default;       // par copie par défaut est OK.
  ...
};
```

Copyright © 2016 Dunod.
© Dunod – Toute reproduction non autorisée est un délit.

Cette approche est souvent utile avec les classes de base polymorphes, c'est-à-dire celles qui définissent des interfaces de manipulation des objets des classes dérivées. Les classes de base polymorphes possèdent généralement un destructeur virtuel, car, lorsque ce n'est pas le cas, certaines opérations (par exemple l'utilisation de `delete` ou de `typeid` sur un objet d'une classe dérivée au travers d'un pointeur ou d'une référence sur une classe de base) peuvent conduire à des résultats indéfinis ou erronés. À moins qu'une classe n'hérite d'un destructeur qui soit déjà virtuel, la seule manière de rendre un destructeur virtuel est de le déclarer de la sorte. L'implémentation par défaut sera souvent correcte et appliquer « `= default` » est une bonne manière de l'exprimer. Cependant, un destructeur déclaré par l'utilisateur désactive la génération des opérations de déplacement et, si ces opérations doivent être prises en charge, « `= default` » trouve alors une seconde application. La déclaration des opérations de déplacement désactive les opérations de copie et, si les possibilités de copie sont également souhaitées, une nouvelle utilisation de « `= default` » résout la question :

```
class Base {
public:
  virtual ~Base() = default;        // Rendre le destructeur virtuel.

  Base(Base&&) = default;           // Prise en charge du déplacement.
  Base& operator=(Base&&) = default;

  Base(const Base&) = default;      // Prise en charge de la copie.
  Base& operator=(const Base&) = default;

  ...

};
```

En réalité, même si nous avons une classe pour laquelle le compilateur est disposé à générer des opérations de copie et de déplacement, et pour laquelle les fonctions générées auront le comportement approprié, nous pouvons choisir de les déclarer nous-mêmes et d'utiliser « `= default` » dans leur définition. Cela nous demande plus de travail, mais nos intentions sont plus claires et certains bogues subtils peuvent être plus faciles à découvrir. Par exemple, supposons que nous ayons une classe qui représente une table de chaînes de caractères, c'est-à-dire une structure de données qui permet des recherches rapides de chaînes de caractères *via* un identifiant entier :

```
class StringTable {
public:
  StringTable() {}
  ...             // Fonctions d'insertion, de suppression, de recherche,
                  // etc., mais pas de copie/déplacement/destructeur.

private:
  std::map<int, std::string> values;
};
```

Copyright © 2016 Dunod.

En supposant que la classe ne déclare aucune opération de copie et de déplacement ni de destructeur, le compilateur générera automatiquement ces fonctions si elles sont utilisées ; une approche très pratique.

Mais supposons que nous décidions plus tard que la journalisation de la construction et de la destruction par défaut de ces objets serait bien utile. L'ajout de cette fonctionnalité est simple :

```
class StringTable {
public:
  StringTable()
  { makeLogEntry("Objet StringTable créé"); }      // Ajoutée.

  ~StringTable()                                   // Également
  { makeLogEntry("Objet StringTable détruit"); }   // ajoutée.

  ...                                        // Autres fonctions précédentes.

private:
  std::map<int, std::string> values;     // Comme précédemment.
};
```

Cela semble raisonnable, mais la déclaration d'un destructeur a un effet secondaire potentiellement important : elle empêche la génération des opérations de déplacement. Toutefois, la création des opérations de copie de la classe n'est pas touchée. Le code va donc certainement compiler, s'exécuter et passer les tests fonctionnels. Cela inclut les tests du déplacement car, même si cette classe n'est plus compatible avec le déplacement, les requêtes de déplacement compileront et s'exécuteront. Comme nous l'avons indiqué précédemment dans ce conseil, elles déclencheront des copies. Autrement dit, le code qui « déplace » des objets `StringTable` effectue en réalité des copies, c'est-à-dire des copies des objets `std::map<int, std::string>` sous-jacents. Malheureusement, la copie d'un `std::map<int, std::string>` risque d'être beaucoup plus lente que son déplacement. Le simple fait d'ajouter un destructeur à la classe peut donc mener à un problème de performance significatif ! Si nous définissons explicitement les opérations de copie et de déplacement avec « `= default` », ce problème disparaît.

Maintenant que vous avez enduré toutes nos explications sur les règles qui gouvernent les opérations de copie et de déplacement en C++11, vous vous demandez peut-être quand nous allons enfin nous intéresser aux deux autres fonctions membres spéciales, le constructeur par défaut et le destructeur. Nous y voilà, mais uniquement par cette phrase, car presque rien n'a changé pour ces fonctions membres : les règles de C++11 sont quasi identiques à celles de C++98.

Voici donc les règles qui gouvernent les fonctions membres spéciales en C++11 :

- **Constructeur par défaut** : les mêmes règles qu'en C++98. Il est généré uniquement si la classe ne contient aucun constructeur déclaré par l'utilisateur.
- **Destructeur** : globalement les mêmes règles qu'en C++98, la seule différence étant que ces destructeurs sont par défaut `noexcept` (voir le conseil 14). Comme en C++98, il est virtuel uniquement si celui de la classe de base est virtuel.

Copyright © 2016 Dunod.
© Dunod – Toute reproduction non autorisée est un délit.

- **Constructeur de copie** : même comportement à l'exécution qu'en C++98, c'est-à-dire construction par copie de niveau membre de données membres non statiques. Il est généré uniquement si la classe ne possède pas de constructeur de copie déclaré par l'utilisateur. Il est supprimé si la classe déclare une opération de déplacement. La génération de cette fonction dans une classe qui dispose d'un opérateur d'affectation par copie ou d'un destructeur déclaré par l'utilisateur est obsolète.
- **Opérateur d'affectation par copie** : même comportement à l'exécution qu'en C++98, c'est-à-dire affectation par copie de niveau membre des données membres non statiques. Il est généré uniquement si la classe ne possède pas d'opérateur d'affectation par copie déclaré par l'utilisateur. Il est supprimé si la classe déclare une opération de déplacement. La génération de cette fonction dans une classe qui dispose d'un constructeur de copie ou d'un destructeur déclaré par l'utilisateur est obsolète.
- **Constructeur de déplacement** et **opérateur d'affectation par déplacement** : chacun effectue un déplacement de niveau membre des données membres non statiques. Ils sont générés uniquement si la classe ne contient aucune opération de copie, opération de déplacement ou destructeur déclaré par l'utilisateur.

Notez que les règles ne disent rien sur l'existence d'un *template* de fonction membre qui empêcherait le compilateur de générer les fonctions membres spéciales. Supposons donc que la classe `Widget` soit déclarée ainsi :

```
class Widget {
  …
  template<typename T>                  // Construire un Widget à partir
  Widget(const T& rhs);                 // de n'importe quoi.

  template<typename T>                  // Affecter un Widget à partir
  Widget& operator=(const T& rhs);      // de n'importe quoi.
  …
};
```

Le compilateur va alors générer les opérations de copie et de déplacement pour `Widget` (en supposant que les conditions habituelles qui régissent leur génération soient satisfaites), même si l'instanciation de ces templates permet d'obtenir la signature du constructeur de copie et de l'opérateur d'affectation par copie (c'est le cas lorsque `T` est `Widget`). Selon toute vraisemblance, cela vous apparaîtra comme un cas secondaire dont il faut rarement se préoccuper, mais nous le mentionnons pour une bonne raison. Le conseil 26 montrera en effet qu'il peut avoir des conséquences importantes.

Copyright © 2016 Dunod.

À retenir

- Les fonctions membres spéciales sont celles que le compilateur peut générer de lui-même : constructeur par défaut, destructeur, opérations de copie et opérations de déplacement.
- Les opérations de déplacement sont générées uniquement lorsque la classe ne déclare explicitement aucune opération de déplacement, opération de copie et destructeur.
- Le constructeur de copie est généré uniquement lorsque la classe ne déclare explicitement aucun constructeur de copie et il est supprimé si une opération de déplacement est déclarée. L'opérateur d'affectation par copie est généré uniquement lorsque la classe ne déclare explicitement aucun opérateur d'affectation par copie et il est supprimé si une opération de déplacement est déclarée. La génération des opérations de copie dans une classe qui possède un destructeur déclaré explicitement est obsolète.
- Les templates de fonctions membres n'empêchent jamais la génération des fonctions membres spéciales.

Copyright © 2016 Dunod.

4

Pointeurs intelligents

Les poètes et les auteurs-compositeurs ont un faible pour l'amour. Et parfois pour le comptage. Quelquefois les deux. Inspirés par les différents essais sur l'amour et le comptage d'Elizabeth Barrett Browning (« Comment t'aimé-je ? Laisse-moi t'en compter les façons ») et Paul Simon (« *There must be 50 ways to leave your lover* »), nous pouvons tenter d'énumérer les raisons du désamour pour le pointeur brut :

1. Sa déclaration n'indique pas s'il pointe sur un seul objet ou sur un tableau.
2. Sa déclaration ne précise pas si nous devons détruire l'élément sur lequel il pointe lorsque nous n'en avons plus besoin, autrement dit si le pointeur *détient* l'élément pointé.
3. Si nous déterminons que nous devons détruire l'élément pointé par le pointeur, rien ne nous dit comment procéder. Devons-nous utiliser `delete` ou employer un autre mécanisme de destruction (par exemple une fonction de destruction particulière à laquelle le pointeur doit être transmis) ?
4. Si nous réussissons à savoir que nous devons utiliser `delete`, la raison 1 fait qu'il peut être impossible de choisir entre la version pour un seul objet (« `delete` ») et celle pour un tableau (« `delete []` »). En cas d'erreur, le résultat est indéfini.
5. En supposant que nous soyons certains que le pointeur détient l'élément sur lequel il pointe et que nous découvrions comment effectuer la destruction, il est difficile d'être sûr que la destruction n'est faite qu'*une seule fois* dans l'ensemble de notre code (y compris dans la gestion des exceptions). Manquer une destruction conduit à une fuite de ressource et répéter une destruction conduit à un comportement indéfini.
6. Il n'y a en général aucun moyen de savoir si un pointeur pointe dans le vide, c'est-à-dire s'il pointe sur une zone de mémoire qui ne contient plus l'objet sur lequel le pointeur est supposé pointer. Les pointeurs « pendouillant »

Copyright © 2016 Dunod.
© Dunod – Toute reproduction non autorisée est un délit.

apparaissent lorsque des objets sont détruits alors que des pointeurs continuent à pointer dessus.

Les pointeurs bruts sont certes des outils puissants, mais des années d'expérience ont montré que la plus petite inattention risque de se retourner contre leurs prétendus maîtres.

Les *pointeurs intelligents* sont une manière de résoudre ces problèmes. Il s'agit d'enveloppes autour de pointeurs bruts, qui se comportent comme les pointeurs qu'ils enveloppent, mais en évitant bon nombre de leurs pièges. Il est donc préférable d'éviter les pointeurs bruts et d'adopter les pointeurs intelligents. Ils peuvent remplacer les pointeurs bruts dans quasiment tous les cas, en laissant peu de place aux erreurs.

En C++11, il existe quatre pointeurs intelligents : `std::auto_ptr`, `std::unique_ptr`, `std::shared_ptr` et `std::weak_ptr`. Ils sont tous conçus pour faciliter la gestion des objets alloués dynamiquement, autrement dit pour éviter les fuites de ressources en s'assurant que les objets sont détruits de la bonne manière au bon moment (y compris lors des exceptions).

`std::auto_ptr` est une relique du C++98. Son objectif était de normaliser ce qui est devenu `std::unique_ptr` en C++11. Pour cela, la sémantique de déplacement était nécessaire, mais elle n'existait pas en C++98. Pour contourner ce manque, `std::auto_ptr` faisait passer ses opérations de copie pour des déplacements. Cela conduisait à du code surprenant (la copie d'un `std::auto_ptr` le fixait à nul !) et à des restrictions d'usage frustrantes (en C++11, il n'était pas possible de stocker des `std::auto_ptr` dans des conteneurs).

`std::unique_ptr` assure toutes les fonctions de `std::auto_ptr`, et plus encore. Il est aussi efficace et opère sans travestir la copie d'un objet. Il est meilleur que `std::auto_ptr` sur tous les plans. Le seul cas d'utilisation légitime de `std::auto_ptr` réside dans l'obligation d'avoir un code compatible avec C++98. Dans tous les autres, `std::auto_ptr` doit être systématiquement remplacé par `std::unique_ptr`.

Les API des pointeurs intelligents sont extrêmement variées. Seule la construction par défaut est commune à l'ensemble d'entre elles. Leur description exhaustive étant largement disponible, nous allons plutôt nous focaliser sur ce qui manque dans ces présentations, par exemple les cas d'utilisation remarquables, l'analyse des coûts à l'exécution, etc. La maîtrise de ces aspects fera la différence entre une utilisation simple des pointeurs intelligents et leur utilisation *efficace*.

CONSEIL N° 18. UTILISER `STD::UNIQUE_PTR` POUR LA GESTION D'UNE RESSOURCE À PROPRIÉTÉ EXCLUSIVE

Lorsque nous souhaitons recourir à un pointeur intelligent, `std::unique_ptr` doit généralement avoir notre priorité. Nous pouvons supposer que, par défaut, les `std::unique_ptr` ont une taille identique à celle des pointeurs bruts et que, pour la plupart des opérations, y compris le déréférencement, ils exécutent exactement les mêmes instructions. Nous pouvons donc les employer même dans les cas où l'occupation de la mémoire et les temps d'exécution revêtent une grande importance.

Copyright © 2016 Dunod.

Si un pointeur brut est suffisamment petit et rapide pour l'application, alors un `std::unique_ptr` l'est certainement également.

`std::unique_ptr` incarne la sémantique de *propriété exclusive*. Un `std::unique_ptr` non nul détient toujours l'élément sur lequel il pointe. Le déplacement d'un `std::unique_ptr` transfère la propriété depuis le pointeur source vers le pointeur destination. (Le pointeur source devient nul.) La copie d'un `std::unique_ptr` est interdite car elle conduirait à deux `std::unique_ptr` sur la même ressource, chacun pensant qu'il la détient (et qu'il peut donc la détruire). `std::unique_ptr` est par conséquent un *type réservé au déplacement*. Lors de la destruction, un `std::unique_ptr` non nul libère sa ressource. Par défaut, la destruction de la ressource se fait en appliquant `delete` au pointeur brut qui se trouve dans le `std::unique_ptr`.

`std::unique_ptr` est souvent employé comme type de retour d'une fonction qui fabrique des objets d'une hiérarchie. Supposons que nous ayons une hiérarchie de types pour représenter des notions d'investissement (par exemple actions, obligations, immobilier, etc.) avec une classe de base nommée `Investment` (figure 4.1) :

```
class Investment { … };

class Stock:                        // Action.
  public Investment { … };

class Bond:                         // Obligation.
  public Investment { … };

class RealEstate:                   // Immobilier.
  public Investment { … };
```

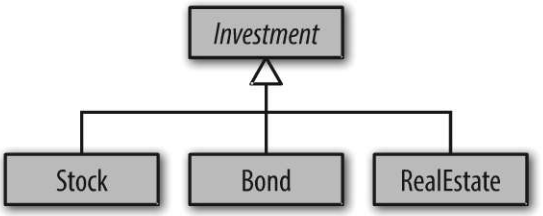


Figure 4.1 — Exemple d'une hiérarchie de types représentant des notions d'investissement.

Avec une telle hiérarchie, la fonction fabrique alloue généralement un objet sur le tas et retourne un pointeur sur cet objet, le code appelant étant responsable de la suppression de l'objet lorsqu'il ne lui est plus utile. Cette situation convient parfaitement à un `std::unique_ptr`, car l'appelant devient responsable de la ressource retournée par la fabrique (en devient le propriétaire exclusif) et le `std::unique_ptr` supprime automatiquement l'élément sur lequel il pointe lorsqu'il est détruit. Voici comment déclarer la fonction fabrique pour la hiérarchie `Investment` :

```
template<typename... Ts>            // Retourner un std::unique_ptr
std::unique_ptr<Investment>         // sur un objet créé à partir
makeInvestment(Ts&&... params);     // des arguments transmis.
```

Copyright © 2016 Dunod.
© Dunod – Toute reproduction non autorisée est un délit.

Dans le code appelant, le `std::unique_ptr` obtenu peut être employé dans une même portée :

```
{
  ...
  auto pInvestment =              // pInvestment est de type
    makeInvestment( arguments );  // std::unique_ptr<Investment>.

  ...

}                                 // Détruire *pInvestment.
```

Il peut également être utilisé dans des scénarios de transfert de la propriété. Par exemple, le `std::unique_ptr` retourné par la fabrique peut être déplacé dans un conteneur, puis l'élément du conteneur est déplacé dans une donnée membre d'un objet, et cet objet est ensuite détruit. Lorsque cela se produit, la donnée membre `std::unique_ptr` de l'objet est également détruite, ce qui déclenche la libération de la ressource renvoyée par la fabrique. Si la chaîne de propriété est interrompue en raison d'une exception ou d'un autre flux de contrôle inhabituel (par exemple le retour précoce d'une fonction ou un `break` dans une boucle), le destructeur du `std::unique_ptr` qui détient la ressource gérée finira par être invoqué[1], avec pour conséquence la destruction de cette ressource.

Par défaut, cette destruction se fait avec `delete`, mais, au cours de sa construction, un objet `std::unique_ptr` peut être configuré de façon à utiliser des *supprimeurs personnalisés* (*custom deleters*) : fonctions arbitraires (ou objets fonctions, y compris ceux provenant d'expressions lambda) qui sont invoquées lorsque leurs ressources doivent être supprimées. Si l'objet créé par `makeInvestment` ne doit pas être supprimé directement avec `delete` mais doit commencer par ajouter une entrée dans un journal, `makeInvestment` peut être implémentée de la manière suivante. (Si quelque chose vous choque, ne vous inquiétez pas, les explications suivent le code.)

```
auto delInvmt = [](Investment* pInvestment)       // Supprimeur
                {                                 // personnalisé
                  makeLogEntry(pInvestment);      // (une expression
                  delete pInvestment;             // lambda).
                };

template<typename... Ts>                          // Type de retour
std::unique_ptr<Investment, decltype(delInvmt)>   // revu.
makeInvestment(Ts&&... params)
{
  std::unique_ptr<Investment, decltype(delInvmt)> // Pointeur à
    pInv(nullptr, delInvmt);                      // retourner.
```

1. Il existe quelques exceptions à cette règle, la plupart étant liées à une terminaison anormale du programme. Si une exception se propage en dehors de la fonction principale d'un thread (par exemple `main` pour le thread initial du programme) ou si une spécification `noexcept` n'est pas respectée (voir le conseil 14), les objets locaux pourraient ne pas être détruits, et si `std::abort` ou une fonction de sortie (c'est-à-dire `std::_Exit`, `std::exit` ou `std::quick_exit`) est appelée, ils ne le seront pas.

Copyright © 2016 Dunod.

```
  if ( /* Si un objet Stock doit être créé. */ )
  {
    pInv.reset(new Stock(std::forward<Ts>(params)...));
  }
  else if ( /* Si un objet Bond doit être créé. */ )
  {
    pInv.reset(new Bond(std::forward<Ts>(params)...));
  }
  else if ( /* Si un objet RealEstate doit être créé. */ )
  {
    pInv.reset(new RealEstate(std::forward<Ts>(params)...));
  }

  return pInv;
}
```

Nous détaillerons le fonctionnement de ce code plus loin, mais commençons par étudier ce qui se passe du côté de l'appelant. En supposant que nous stockions le résultat de l'appel à `makeInvestment` dans une variable `auto`, nous vivons comme des bienheureux sans nous soucier du fait que la ressource exploitée nécessite un traitement particulier au moment de la suppression. En réalité, nous pouvons faire preuve d'insouciance car l'utilisation de `std::unique_ptr` nous permet d'éviter les tracas de la destruction de la ressource (à quel moment la détruire et comment être sûr qu'elle ne se produit une seule fois dans l'ensemble du programme). `std::unique_ptr` prend automatiquement en charge tous ces aspects. Du point de vue du client, l'interface de `makeInvestment` est agréable.

L'implémentation est également sympathique, dès lors que l'on comprend les points suivants :

- `delInvmt` est le supprimeur personnalisé de l'objet renvoyé par `makeInvestment`. Toutes les fonctions de suppression personnalisées prennent un pointeur brut sur l'objet à détruire, puis elles s'arrangent pour détruire cet objet. Dans ce cas, la procédure consiste à appeler `makeLogEntry`, puis à appliquer `delete`. La création de `delInvmt` avec une expression lambda se révèle pratique mais, nous le verrons plus loin, cette solution est également plus efficace que l'écriture d'une fonction classique.
- Lorsqu'un supprimeur personnalisé est requis, son type doit être indiqué dans le second argument de type de `std::unique_ptr`. Dans notre exemple, il s'agit du type de `delInvmt` et c'est pourquoi le type de retour de `makeInvestment` est `std::unique_ptr<Investment, decltype(delInvmt)>`. (Pour de plus amples informations sur `decltype`, consulter le conseil 3.)
- La stratégie de base de `makeInvestment` consiste à créer un `std::unique_ptr` nul, à le faire pointer sur un objet du type approprié, puis à le retourner. Pour associer le supprimeur personnalisé `delInvmt` à `pInv`, nous le passons en second argument du constructeur.
- L'affectation d'un pointeur brut (par exemple obtenu avec `new`) à un `std::unique_ptr` ne passera pas la compilation, car elle représente une conversion implicite depuis un pointeur brut vers un pointeur intelligent. Puisque de telles conversions implicites peuvent poser des problèmes, les

Copyright © 2016 Dunod.
© Dunod – Toute reproduction non autorisée est un délit.

pointeurs intelligents de C++11 les interdisent. Nous employons donc `reset` pour que `pInv` assume la propriété de l'objet créé *via* `new`.

- Pour chaque appel à `new`, nous utilisons `std::forward` de façon à transmettre parfaitement les arguments passés à `makeInvestment` (voir le conseil 25). De cette manière, toutes les informations fournies par les appelants sont disponibles dans les constructeurs des objets créés.
- Le supprimeur personnalisé prend un paramètre de type `Investment*`. Quel que soit le type réel de l'objet créé dans `makeInvestment` (c'est-à-dire `Stock`, `Bond` ou `RealEstate`), il finira par être détruit en tant qu'objet `Investment*` par un appel à `delete` dans l'expression lambda. Cela signifie que nous allons détruire un objet d'une classe dérivée au travers d'un pointeur sur la classe de base. Pour que cela fonctionne, la classe de base, `Investment`, doit disposer d'un destructeur virtuel.

```
class Investment {
public:
  …                                                  // Élément
  virtual ~Investment();                             // essentiel
  …                                                  // de la conception !
};
```

Grâce à la déduction du type de retour d'une fonction en C++14 (voir le conseil 3), nous pouvons avoir une implémentation de `makeInvestment` plus simple, avec une meilleure encapsulation :

```
template<typename... Ts>
auto makeInvestment(Ts&&... params)                  // C++14.
{
  auto delInvmt = [](Investment* pInvestment)        // Ce code se trouve
                  {                                  // à présent dans
                    makeLogEntry(pInvestment);       // makeInvestment.
                    delete pInvestment;
                  };

  std::unique_ptr<Investment, decltype(delInvmt)>    // Comme précédemment.
    pInv(nullptr, delInvmt);

  if ( … )                                           // Comme précédemment.
  {
    pInv.reset(new Stock(std::forward<Ts>(params)...));
  }
  else if ( … )                                      // Comme précédemment.
  {
    pInv.reset(new Bond(std::forward<Ts>(params)...));
  }
  else if ( … )                                      // Comme précédemment.
  {
    pInv.reset(new RealEstate(std::forward<Ts>(params)...));
  }
  return pInv;                                       // Comme précédemment.
}
```

Copyright © 2016 Dunod.

Nous avons indiqué précédemment qu'en utilisant le supprimeur par défaut (c'est-à-dire `delete`), nous pouvons raisonnablement supposer que les objets `std::unique_ptr` ont la même taille que les pointeurs bruts. En présence de supprimeurs personnalisés, ce n'est généralement plus le cas. Un supprimeur donné sous forme d'un pointeur de fonction augmente la taille d'un `std::unique_ptr` d'un ou deux mots. Lorsque le supprimeur est un objet fonction, la variation de la taille dépend des éléments d'état stockés dans l'objet fonction. Les objets fonctions sans état (par exemple provenant d'expressions lambda sans capture) n'ont aucune incidence sur la taille. Par conséquent, lorsque l'implémentation d'un supprimeur personnalisé peut se faire sous forme soit d'une fonction, soit d'une expression lambda sans capture, cette dernière option est préférable :

```
auto delInvmt1 = [](Investment* pInvestment)      // Supprimeur
                 {                                // personnalisé
                   makeLogEntry(pInvestment);     // sous forme
                   delete pInvestment;            // d'expression lambda
                 };                               // sans état.

template<typename... Ts>                          // Le type de retour
std::unique_ptr<Investment, decltype(delInvmt1)>  // a la taille de
makeInvestment(Ts&&... args);                     // Investment*.

void delInvmt2(Investment* pInvestment)           // Supprimeur
{                                                 // personnalisé
  makeLogEntry(pInvestment);                      // sous forme
  delete pInvestment;                             // de fonction.
}

template<typename... Ts>                  // Le type de retour a la
std::unique_ptr<Investment,               // taille de Investment*,
                void (*)(Investment*)>    // plus au moins la taille
makeInvestment(Ts&&... params);           // d'un pointeur de fonction !
```

Lorsque des objets fonctions possèdent un état important et sont utilisés comme supprimeurs, la taille des objets `std::unique_ptr` risque d'être significative. Si un supprimeur personnalisé conduit à un `std::unique_ptr` trop volumineux, il est préférable de revoir la conception du code.

Les fonctions fabriques ne représentent pas le seul cas d'utilisation classique des `std::unique_ptr`. Ils sont même plus connus comme mécanisme d'implémentation de l'idiome Pimpl. Le code correspondant n'est pas compliqué, mais il peut parfois ne pas être très évident. Nous y reviendrons au conseil 22, consacré à ce sujet.

Un `std::unique_ptr` existe sous deux formes : l'une pour les objets individuels (`std::unique_ptr<T>`), l'autre pour les tableaux (`std::unique_ptr<T[]>`). En conséquence, il n'y a jamais d'ambiguïté sur le type d'entité ciblée par un `std::unique_ptr`. L'API de `std::unique_ptr` est conçue pour correspondre à la variante employée. Par exemple, il n'existe pas d'opérateur d'indexation (`operator[]`) dans la forme adaptée aux objets individuels, tandis que celle réservée aux tableaux ne dispose pas des opérateurs de déréférencement (`operator*` et `operator->`).

Copyright © 2016 Dunod.
© Dunod – Toute reproduction non autorisée est un délit.

L'existence de `std::unique_ptr` pour les tableaux ne présente essentiellement qu'un intérêt intellectuel car `std::array`, `std::vector` et `std::string` sont en général des structures de données mieux adaptées aux tableaux bruts. Le seul cas où nous pouvons imaginer l'utilisation d'un `std::unique_ptr<T[]>` serait dans le contexte d'une API de type C qui retourne un pointeur brut sur un tableau alloué sur le tas et dont nous devons assurer la propriété.

`std::unique_ptr` est la manière C++11 d'exprimer une propriété exclusive. Mais l'une de ses fonctionnalités les plus attrayantes est qu'il se transforme facilement en un `std::shared_ptr` efficace :

```
std::shared_ptr<Investment> sp =      // Convertir un std::unique_ptr
  makeInvestment( arguments );        // en un std::shared_ptr.
```

C'est l'une des principales raisons pour lesquelles un `std::unique_ptr` convient parfaitement en type de retour d'une fonction fabrique. En effet, ces fonctions ne savent pas si le code appelant souhaite une propriété exclusive ou partagée (c'est-à-dire un `std::shared_ptr`) sur l'objet retourné. En retournant un `std::unique_ptr`, elles donnent donc aux appelants le pointeur intelligent le plus efficace, sans les empêcher de le remplacer par son homologue plus souple. (Pour de plus amples informations sur `std::shared_ptr`, consulter le conseil 19.)

À retenir

- Un `std::unique_ptr` est un pointeur intelligent concis, rapide et réservé au déplacement. Il convient à la gestion de ressources à propriété exclusive.
- Par défaut, la destruction d'une ressource se fait avec `delete`, mais il est possible de spécifier des supprimeurs (*deleters*) personnalisés. Les supprimeurs avec état et les pointeurs de fonctions utilisés comme supprimeurs augmentent la taille des objets `std::unique_ptr`.
- Convertir un `std::unique_ptr` en un `std::shared_ptr` est un jeu d'enfant.

CONSEIL N° 19. UTILISER `STD::SHARED_PTR` POUR LA GESTION D'UNE RESSOURCE À PROPRIÉTÉ PARTAGÉE

Les programmeurs qui utilisent des langages dotés d'un ramasse-miettes rigolent bien devant le travail que doivent effectuer les programmeurs C++ pour éviter les fuites de ressources. « Que ce langage est primitif !», raillent-ils. « N'avez-vous pas lu l'article sur Lisp dans les années 1960 ? La vie des ressources doit être gérée non pas par les humains mais par les machines. » Les développeurs C++ écarquillent les yeux. « Vous voulez parler de l'article dans lequel la seule ressource mentionnée était la mémoire et où la libération des ressources était non déterministe ? Merci bien, nous préférons le caractère plus général et prévisible des destructeurs. » Notre bravade

Copyright © 2016 Dunod.

est en réalité une fanfaronnade. Le ramasse-miettes est réellement très commode et la gestion manuelle du cycle de vie des ressources équivaut à construire un circuit de mémoire mnémonique à l'aide de couteaux en pierre en étant habillés d'une peau d'ours. Pourquoi ne pourrions-nous pas profiter du meilleur des deux mondes : un système qui opère de façon automatique (comme le ramasse-miettes), mais qui s'applique à toutes les ressources et de manière prévisible (comme les destructeurs) ?

Pour réunir ces deux mondes, la solution C++11 passe par `std::shared_ptr`. Lorsqu'un objet est manipulé au travers de `std::shared_ptr`, la gestion de son cycle de vie est assurée par ces pointeurs, avec une *propriété partagée*. L'objet n'est détenu par aucun `std::shared_ptr` en particulier. À la place, tous les `std::shared_ptr` qui pointent sur cet objet collaborent pour assurer sa destruction lorsqu'il n'est plus utile. Lorsque le dernier `std::shared_ptr` qui pointe sur un objet arrête de pointer dessus (par exemple en raison de la destruction du `std::shared_ptr` ou de sa redirection vers un autre objet), ce `std::shared_ptr` détruit l'objet concerné. Grâce au ramasse-miettes, le code client n'a pas à se préoccuper du cycle de vie des objets pointés et, grâce aux destructeurs, le moment de la destruction des objets est déterministe.

Pour savoir qu'il est le dernier à pointer sur une ressource, le `std::shared_ptr` consulte le *compteur de références* de cette ressource. Il s'agit d'une valeur associée à la ressource et dont l'objectif est de suivre le nombre de `std::shared_ptr` qui pointent sur elle. Les constructeurs de `std::shared_ptr` incrémentent ce compteur (en général, mais voir ci-après), les destructeurs le décrémentent, et les opérateurs d'affectation par copie effectuent les deux opérations. (Si `sp1` et `sp2` sont des `std::shared_ptr` sur des objets différents, l'affectation « `sp1 = sp2;` » modifie `sp1` de sorte qu'il pointe sur l'objet ciblé par `sp2`. Le compteur de références de l'objet initialement visé par `sp1` est décrémenté, tandis que celui de l'objet pointé par `sp2` est incrémenté.) Si un `std::shared_ptr` constate un compteur de références égal à zéro après avoir effectué sa décrémentation, cela signifie que plus aucun `std::shared_ptr` ne cible la ressource et ce `std::shared_ptr` la détruit donc.

L'existence du compteur de références a un impact sur les performances :

- **La taille des `std::shared_ptr` est deux fois plus importante que celle des pointeurs bruts** car, en interne, ils contiennent un pointeur brut sur la ressource et un pointeur brut sur le compteur de références associé à cette ressource[1].
- **La mémoire associée au compteur de références doit être allouée dynamiquement.** Conceptuellement, le compteur de références est associé à l'objet pointé, mais cet objet n'en a pas connaissance. Il ne dispose donc d'aucune place pour stocker un compteur de références. (Conséquence intéressante, cela signifie que n'importe quel objet, même un type intégré, peut être géré par un `std::shared_ptr`.) Le conseil 21 explique que le coût de l'allocation dynamique est évité lorsque le `std::shared_ptr` est créé par `std::make_shared`, mais cette

1. Cette implémentation n'est pas imposée par la norme, mais c'est elle que nous avons rencontrée partout.

Copyright © 2016 Dunod.
© Dunod – Toute reproduction non autorisée est un délit.

approche n'est pas toujours envisageable. Quoi qu'il en soit, le compteur de références est stocké sous forme de données allouées dynamiquement.

- **Les incrémentations et les décrémentations du compteur de références doivent être atomiques.** En effet, il est possible que des lectures et des écritures se produisent simultanément dans des threads différents. Par exemple, un `std::shared_ptr` qui pointe sur une ressource dans un thread pourrait être en train d'exécuter son destructeur (d'où une décrémentation du compteur de références associé à la ressource ciblée), pendant que, dans un thread différent, un `std::shared_ptr` sur le même objet pourrait être copié (d'où une incrémentation du même compteur de références). Puisque les opérations atomiques sont typiquement plus lentes que les opérations non atomiques, nous pouvons supposer que, même si les compteurs de références n'occupent en général qu'un mot, leur lecture et leur écriture sont plus coûteuses.

Avons-nous piqué votre curiosité en indiquant que les constructeurs de `std::shared_ptr` incrémentent « en général » le compteur de références de l'objet pointé ? Puisque la création d'un `std::shared_ptr` qui pointe sur un objet amène toujours un `std::shared_ptr` supplémentaire à pointer sur cet objet, pourquoi le compteur de références n'est-il pas *toujours* incrémenté ?

L'explication tient dans la construction par déplacement. En construisant un `std::shared_ptr` par déplacement à partir d'un autre `std::shared_ptr`, le `std::shared_ptr` d'origine est fixé à nul et ne cible donc plus sa ressource lorsque le nouveau `std::shared_ptr` entre en scène. Par conséquent, le compteur de références ne doit pas être modifié. Le déplacement d'un `std::shared_ptr` est donc plus rapide que sa copie, qui exige une incrémentation du compteur de références. Cela est également vrai pour l'affectation. Par conséquent, la construction et l'affectation par déplacement sont plus rapides que la construction et l'affectation par copie.

À l'instar de `std::unique_ptr` (voir le conseil 18), `std::shared_ptr` utilise par défaut `delete` pour la destruction de la ressource, mais il prend également en charge les supprimeurs personnalisés. La conception de cette prise en charge diffère toutefois de celle de `std::unique_ptr`. En effet, dans le cas de `std::unique_ptr`, le type du supprimeur fait partie du type du pointeur intelligent, ce qui n'est pas le cas pour `std::shared_ptr` :

```
auto loggingDel = [](Widget *pw)        // Supprimeur personnalisé
                  {                     // (comme au conseil 18).
                    makeLogEntry(pw);
                    delete pw;
                  };

std::unique_ptr<                        // Le type du supprimeur fait
  Widget, decltype(loggingDel)          // partie du type du pointeur.
  > upw(new Widget, loggingDel);

std::shared_ptr<Widget>                 // Le type du supprimeur ne fait
  spw(new Widget, loggingDel);          // pas partie du type du pointeur.
```

Copyright © 2016 Dunod.

La conception de `std::shared_ptr` est plus souple. Prenons deux `std::shared_ptr<Widget>`, chacun avec un supprimeur personnalisé de type différent (par exemple parce que les supprimeurs personnalisés sont spécifiés *via* des expressions lambda) :

```
auto customDeleter1 = [](Widget *pw) { … };    // Supprimeurs
auto customDeleter2 = [](Widget *pw) { … };    // personnalisés, chacun
                                               // de type différent.

std::shared_ptr<Widget> pw1(new Widget, customDeleter1);
std::shared_ptr<Widget> pw2(new Widget, customDeleter2);
```

Puisque `pw1` et `pw2` sont du même type, ils peuvent être placés dans un conteneur d'objets de ce type :

```
std::vector<std::shared_ptr<Widget>> vpw{ pw1, pw2 };
```

Ils peuvent également être affectés l'un à l'autre et être chacun transmis à une fonction qui prend un paramètre de type `std::shared_ptr<Widget>`. Aucune de ces opérations n'est possible avec des `std::unique_ptr` qui auraient des supprimeurs personnalisés de types différents, car le type du supprimeur personnalisé affecte le type du `std::unique_ptr`.

Il existe une différence entre `std::unique_ptr` et `std::shared_ptr`. La spécification d'un supprimeur personnalisé ne change pas la taille d'un objet `std::shared_ptr`. Quel que soit le supprimeur, la taille d'un objet `std::shared_ptr` est celle de deux pointeurs. C'est une bonne nouvelle, mais elle vous met peut-être mal à l'aise. En effet, les supprimeurs personnalisés peuvent être des objets fonctions, qui peuvent contenir une quantité de données quelconque. Autrement dit, ils peuvent avoir une taille arbitraire. Comment un `std::shared_ptr` qui fait référence à un supprimeur de taille arbitraire fait-il pour ne pas occuper un espace mémoire de taille supérieure ?

C'est impossible. Il doit utiliser une plus grande quantité de mémoire. Toutefois, cette mémoire ne fait pas partie de l'objet `std::shared_ptr`. Elle se trouve sur le tas ou, si le créateur du `std::shared_ptr` tire profit des allocateurs personnalisés, là où est placée la mémoire gérée par l'allocateur. Nous avons mentionné précédemment qu'un objet `std::shared_ptr` comprend un pointeur sur le compteur de références de l'objet ciblé. C'est vrai, mais un tantinet erroné, car le compteur de références fait partie d'une structure de données plus large appelée *bloc de contrôle*. Il existe un bloc de contrôle pour chaque objet géré par des `std::shared_ptr`. Ce bloc de contrôle comprend, outre le compteur de références, une copie du supprimeur personnalisé, s'il a été spécifié. Si un allocateur personnalisé a été défini, le bloc de contrôle en comprend également une copie. Des données supplémentaires peuvent compléter ces informations, par exemple, comme l'explique le conseil 21, un compteur de références secondaire, appelé *compteur de références faibles*, mais nous allons les ignorer dans ce conseil. La figure 4.2 montre comment nous pouvons voir la mémoire associée à un objet `std::shared_ptr<T>`.

Copyright © 2016 Dunod.
© Dunod – Toute reproduction non autorisée est un délit.

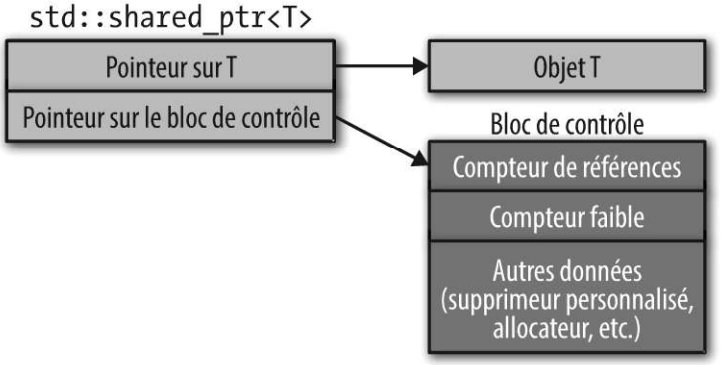


Figure 4.2 — Mémoire associée à un objet `std::shared_ptr<T>`.

Le bloc de contrôle d'un objet est mis en place par la fonction qui crée le premier `std::shared_ptr` sur l'objet. Tout au moins, c'est ainsi que cela est supposé se passer. Malheureusement, la fonction qui crée un `std::shared_ptr` sur un objet ne peut en général pas savoir si d'autres `std::shared_ptr` pointent déjà sur cet objet. Par conséquent, voici les règles de la création du bloc de contrôle :

- `std::make_shared` **(voir le conseil 21) crée toujours un bloc de contrôle.** Puisqu'elle fabrique un nouvel objet pointé, il ne devrait donc pas exister de bloc de contrôle pour cet objet au moment où elle est appelée.
- Un bloc de contrôle est créé lorsqu'un `std::shared_ptr` est construit à partir d'un pointeur à propriété exclusive (c'est-à-dire un `std::unique_ptr` ou un `std::auto_ptr`). Puisque les pointeurs à propriété exclusive ne se servent pas des blocs de contrôle, il ne devrait donc pas exister de bloc de contrôle pour l'objet pointé. (Pendant sa construction, le `std::shared_ptr` assume la propriété de l'objet pointé et le pointeur à propriété exclusive est donc fixé à nul.)
- **Lorsqu'un constructeur de `std::shared_ptr` est appelé avec un pointeur brut, il crée un bloc de contrôle.** Si nous voulions créer un `std::shared_ptr` à partir d'un objet pour lequel il existe déjà un bloc de contrôle, nous passerions non pas un pointeur bruit mais certainement un `std::shared_ptr` ou un `std::weak_ptr` (voir le conseil 20) en argument du constructeur. Les constructeurs de `std::shared_ptr` qui prennent en arguments des `std::shared_ptr` ou des `std::weak_ptr` ne créent pas de nouveaux blocs de contrôle, car ils peuvent compter sur les pointeurs intelligents transmis pour disposer des blocs de contrôle requis.

Ces règles ont une conséquence malheureuse. La construction de plusieurs `std::shared_ptr` à partir d'un même pointeur brut conduit à un comportement indéfini, car plusieurs blocs de contrôle seront associés à l'objet pointé. Ces multiples blocs de contrôle signifient plusieurs compteurs de références, ce qui signifie plusieurs destructions de l'objet (une pour chaque compteur). Autrement dit, le code suivant est on ne peut plus mauvais :

Copyright © 2016 Dunod.

```
auto pw = new Widget;                          // pw est un pointeur brut.

…

std::shared_ptr<Widget> spw1(pw, loggingDel);  // Bloc de contrôle
                                               // créé pour *pw.
…

std::shared_ptr<Widget> spw2(pw, loggingDel);  // 2e bloc de contrôle
                                               // créé pour *pw !
```

La création du pointeur brut `pw` sur un objet alloué dynamiquement est une mauvaise idée. En effet, elle va à l'encontre du conseil prodigué tout au long de ce chapitre : préférer les pointeurs intelligents aux pointeurs bruts. (Si vous avez oublié ce qui motive cette préférence, revenez au début de ce chapitre.) Mais laissons cela de côté. La ligne qui crée `pw` est une abomination stylistique, mais au moins elle ne cause pas le comportement indéfini du programme.

Le constructeur de `spw1` est appelé avec un pointeur brut et crée donc un bloc de contrôle (et par conséquent un compteur de références) pour l'élément pointé. Dans ce cas, il s'agit de `*pw` (c'est-à-dire l'objet pointé par `pw`). Cela ne pose pas de problème en soi, mais le constructeur de `spw2` est appelé avec le même pointeur brut, ce qui conduit à la création d'un bloc de contrôle (et par conséquent d'un compteur de références) pour `*pw`. Il existe donc deux compteurs de références pour `*pw`. Chacun finira par être égal à zéro, ce qui déclenchera deux tentatives de destruction de `*pw`. La seconde est responsable du comportement indéfini.

Nous pouvons tirer au moins deux leçons de cet exemple d'utilisation de `std::shared_ptr`. Premièrement, il faut éviter de passer un pointeur brut à un constructeur de `std::shared_ptr`. L'alternative consiste généralement à utiliser `std::make_shared` (voir le conseil 21), mais notre exemple précédent met en œuvre des supprimeurs personnalisés, ce qui est incompatible avec `std::make_shared`. Deuxièmement, si nous devons transmettre un pointeur brut à un constructeur de `std::shared_ptr`, il faut que ce soit directement le résultat de `new`, sans passer par l'intermédiaire d'une variable. Nous pouvons ainsi réécrire la première partie du code précédent :

```
std::shared_ptr<Widget> spw1(new Widget,       // Utilisation directe
                             loggingDel);      // de new.
```

Nous serons alors moins tentés de créer un second `std::shared_ptr` à partir du même pointeur brut. À la place, il semblera plus naturel de créer `spw2` en passant `spw1` en argument d'initialisation (autrement dit appeler le constructeur de copie de `std::shared_ptr`), ce qui évitera tout problème :

```
std::shared_ptr<Widget> spw2(spw1);      // spw2 se sert du même bloc
                                         // de contrôle que spw1.
```

Copyright © 2016 Dunod.
© Dunod – Toute reproduction non autorisée est un délit.

Une autre utilisation particulièrement surprenante d'un pointeur brut en argument du constructeur de `std::shared_ptr` menant à la création de plusieurs blocs de contrôle implique le pointeur `this`. Supposons que notre programme fonde la gestion d'objets `Widget` sur des `std::shared_ptr` et que la trace des `Widget` traités est conservée dans une structure de données :

```
std::vector<std::shared_ptr<Widget>> processedWidgets;
```

Supposons également que le traitement soit réalisé par une fonction membre de `Widget` :

```
class Widget {
public:
  ...
  void process();
  ...
};
```

Voici une façon *a priori* convenable d'implémenter `Widget::process` :

```
void Widget::process()
{
  ...                                          // Traiter le Widget.

  processedWidgets.emplace_back(this);         // L'ajouter à la liste
}                                              // des Widget traités ;
                                               // mauvaise idée !
```

Le commentaire indique que la façon de procéder est mauvaise. (Le problème concerne le passage de `this`, non l'utilisation de `emplace_back`. Pour de plus amples informations sur `emplace_back`, consulter le conseil 42.) La compilation de ce code ne pose aucune difficulté, mais il transmet un pointeur brut (`this`) à un conteneur de `std::shared_ptr`. Le `std::shared_ptr` ainsi construit va créer un nouveau bloc de contrôle pour le `Widget` pointé (`*this`). Cela n'a rien de choquant, jusqu'à ce que nous réalisions que si des `std::shared_ptr` extérieurs à la fonction membre pointent déjà sur ce `Widget`, le comportement indéfini décrit précédemment est de retour.

L'API `std::shared_ptr` apporte une solution à cette situation. Son nom est probablement le plus bizarre de tous ceux de la bibliothèque C++ standard : `std::enable_shared_from_this`. Il s'agit d'un template pour une classe de base dont nous devons hériter pour qu'une classe gérée par des `std::shared_ptr` puisse créer en toute sécurité un `std::shared_ptr` à partir d'un pointeur `this`. Voici son utilisation dans notre exemple de `Widget` :

```
class Widget: public std::enable_shared_from_this<Widget> {
public:
  ...
  void process();
  ...
};
```

Copyright © 2016 Dunod.

Nous l'avons dit, `std::enable_shared_from_this` est un template de classe de base. Son paramètre de type est toujours le nom de la classe dérivée. Par conséquent, `Widget` hérite donc de `std::enable_shared_from_this<Widget>`. Si l'idée d'une classe dérivée qui hérite d'un template d'une classe de base sur la classe dérivée vous donne le tournis, essayez de ne pas trop y penser. Le code est parfaitement valide et le design pattern sur lequel il se fonde est parfaitement connu. Son nom est établi, même s'il est aussi étrange que `std::enable_shared_from_this` : *Curiously Recurring Template Pattern* (CRTP). Si vous souhaitez en apprendre plus sur ce pattern, tournez-vous vers votre moteur de recherche car nous devons revenir à `std::enable_shared_from_this`.

`std::enable_shared_from_this` définit une fonction membre qui crée un `std::shared_ptr` sur l'objet courant, mais sans dupliquer les blocs de contrôle. Cette fonction se nomme `shared_from_this` et nous devons l'appeler depuis les fonctions membres pour obtenir un `std::shared_ptr` qui cible le même objet que le pointeur `this`. Voici une implémentation fiable de `Widget::process` :

```
void Widget::process()
{
  // Comme précédemment, traiter le Widget.
  …

  // Ajouter le std::shared_ptr sur l'objet courant
  // à processedWidgets.
  processedWidgets.emplace_back(shared_from_this());
}
```

De façon interne, `shared_from_this` recherche le bloc de contrôle associé à l'objet courant et crée un nouveau `std::shared_ptr` qui fait référence à ce bloc de contrôle. Ce comportement suppose qu'un bloc de contrôle soit associé à l'objet courant. Pour cela, il doit déjà exister un `std::shared_ptr` (par exemple en dehors de la fonction membre qui appelle `shared_from_this`) qui pointe sur l'objet courant. Si ce n'est pas le cas (autrement dit si aucun bloc de contrôle n'est lié à l'objet courant), le comportement est indéfini, mais `shared_from_this` lance en général une exception.

Pour éviter que du code client n'appelle des fonctions membres qui invoquent `shared_from_this` avant qu'un `std::shared_ptr` ne pointe sur l'objet, les classes qui dérivent de `std::enable_shared_from_this` déclarent souvent leurs constructeurs `private` et permettent la création des objets au travers de fonctions fabriques qui retournent des `std::shared_ptr`. Par exemple, `Widget` pourrait être déclarée de la manière suivante :

```
class Widget: public std::enable_shared_from_this<Widget> {
public:
  // Fonction fabrique qui transmet parfaitement les arguments
  // à un constructeur privé.
  template<typename... Ts>
  static std::shared_ptr<Widget> create(Ts&&... params);

  …
  void process();             // Comme précédemment.
```

© Dunod – Toute reproduction non autorisée est un délit.
Copyright © 2016 Dunod.

```
  ...

private:
  ...                              // Constructeurs.
};
```

À ce stade, vous avez peut-être vaguement oublié que nos propos sur les blocs de contrôle étaient motivés par une volonté de comprendre les coûts associés aux `std::shared_ptr`. Puisque nous savons à présent comment éviter la création d'un trop grand nombre de blocs de contrôle, revenons au sujet initial.

La taille d'un bloc de contrôle est en général de quelques mots, bien que les supprimeurs et les allocateurs personnalisés puissent l'augmenter. L'implémentation classique d'un bloc de contrôle est donc plus sophistiquée qu'on l'aurait imaginé. L'héritage et une fonction virtuelle sont de la partie. (La fonction virtuelle est utilisée pour s'assurer que la destruction de l'objet pointé se fait correctement.) Autrement dit, l'utilisation des `std::shared_ptr` subit également le coût de toute la mécanique associée à la fonction virtuelle employée par le bloc de contrôle.

Nous avons donc des blocs de contrôle alloués dynamiquement, des supprimeurs et des allocateurs de taille arbitraire, la mécanique des fonctions virtuelles et les manipulations atomiques d'un compteur de références. Nous pourrions comprendre que votre enthousiasme pour les `std::shared_ptr` ait quelque peu disparu. Certes, ils n'apportent pas la meilleure solution à chaque problème de gestion d'une ressource, mais ils ont un coût très raisonnable. Dans des conditions classiques, lorsque le supprimeur et l'allocateur par défaut sont employés et lorsque le `std::shared_ptr` est créé avec `std::make_shared`, le bloc de contrôle occupe simplement trois mots et son allocation est quasiment gratuite. (Elle est incluse dans l'allocation de la mémoire destinée à l'objet pointé ; voir le conseil 21.) Déréférencer un `std::shared_ptr` n'est pas plus coûteux que déréférencer un pointeur brut. Effectuer une opération qui demande la manipulation d'un compteur de références (par exemple, une construction ou une affectation par copie, une destruction) entraîne une ou deux opérations atomiques, mais celles-ci correspondent généralement à des instructions machine en propre, qui ne seront pas nécessairement plus coûteuses que des instructions non atomiques (ce sont des instructions simples). La mécanique des fonctions virtuelles dans le bloc de contrôle est généralement activée une seule fois pour chaque objet géré *via* des `std::shared_ptr` : au moment où l'objet est détruit.

En échange de ces coûts plutôt modestes, nous obtenons une gestion automatique du cycle de vie des ressources allouées dynamiquement. La plupart du temps, l'emploi d'un `std::shared_ptr` est beaucoup plus intéressant que la gestion manuelle d'un objet dont la propriété est partagée. En cas de doute sur l'intérêt de l'utilisation d'un `std::shared_ptr`, il faut réfléchir à la nécessité d'une propriété partagée. Si une propriété exclusive fait ou pourrait faire l'affaire, `std::unique_ptr` représente un meilleur choix. Ses performances sont proches de celles obtenues avec des pointeurs bruts et le passage de `std::unique_ptr` à `std::shared_ptr` reste facile, car nous pouvons créer un `std::shared_ptr` à partir d'un `std::unique_ptr`.

L'inverse est faux. Dès lors que la gestion d'une ressource a été confiée à un `std::shared_ptr`, il n'est plus possible de revenir en arrière. Même si le compteur de

Copyright © 2016 Dunod.

références reste à un, nous ne pouvons pas réclamer la propriété de la ressource, par exemple pour la gérer à l'aide d'un `std::unique_ptr`. Le contrat de propriété entre une ressource et les `std::shared_ptr` qui pointent dessus est signé « jusqu'à ce que la mort nous sépare ».

Par ailleurs, les `std::shared_ptr` sont incompatibles avec les tableaux. Au contraire de `std::unique_ptr`, `std::shared_ptr` dispose d'une API conçue uniquement pour des pointeurs sur des objets. `std::shared_ptr<T[]>` n'existe pas. De temps à autre, des programmeurs « astucieux » se mettent en tête d'utiliser un `std::shared_ptr<T>` pour pointer sur un tableau, en définissant un supprimeur personnalisé pour la destruction du tableau (c'est-à-dire `delete []`). Il est possible que leur code compile, mais l'idée est très mauvaise. Tout d'abord, puisque `std::shared_ptr` ne propose pas `operator[]`, l'indexation dans le tableau exige des expressions délicates à base d'arithmétique sur les pointeurs. Ensuite, `std::shared_ptr` prend en charge les conversions de pointeurs entre classe de base et classe dérivée qui ont un sens pour les objets simples, mais qui ouvrent des brèches dans le système de typage lorsque les tableaux sont concernés. (C'est pour cette raison que l'API de `std::unique_ptr<T[]>` interdit ces conversions.) Mais plus important encore, en raison de la diversité des méthodes de création d'un tableau en C++11 (par exemple `std::array`, `std::vector`, `std::string`), la déclaration d'un pointeur intelligent sur un tableau de base est toujours signe d'une mauvaise conception.

À retenir

- Les `std::shared_ptr` ont une utilité proche de celle d'un ramasse-miettes pour la gestion du cycle de vie de ressources quelconques partagées.
- En comparaison de `std::unique_ptr`, les objets `std::shared_ptr` ont une taille généralement deux fois plus importante, impliquent un surcoût lié aux blocs de contrôle et exigent des manipulations atomiques du compteur de références.
- La destruction de la ressource se fait par défaut avec `delete`, mais les supprimeurs personnalisés sont pris en charge. Le type du supprimeur n'a pas d'incidence sur celui du `std::shared_ptr`.
- Il faut éviter de créer des `std::shared_ptr` à partir de variables de type pointeur brut.

CONSEIL N° 20. UTILISER STD::WEAK_PTR POUR DES POINTEURS DE TYPE STD::SHARED_PTR QUI PEUVENT PENDOUILLER

Cela pourrait paraître paradoxal, mais il peut être commode d'avoir un pointeur intelligent qui opère à la manière d'un `std::shared_ptr` (voir le conseil 19) sans qu'il participe au partage de la propriété de la ressource pointée. Autrement dit, un pointeur du type de `std::shared_ptr` qui n'affecte pas le compteur de références d'un objet. Cette forme de pointeur intelligent doit affronter un problème inconnu

Copyright © 2016 Dunod.
© Dunod – Toute reproduction non autorisée est un délit.

des `std::shared_ptr` : la possibilité de pointer sur un élément qui a été détruit. Un pointeur véritablement intelligent traitera cette question en détectant le moment où il *pendouille* (pointe dans le vide), c'est-à-dire lorsque l'objet sur lequel il est supposé pointer n'existe plus. C'est précisément le type de pointeur mis en œuvre par `std::weak_ptr`.

Vous vous demandez peut-être à quoi peut bien servir `std::weak_ptr`. Votre interrogation sera même plus grande encore si vous examinez l'API de `std::weak_ptr`. Elle a l'air de tout, sauf intelligent. Les `std::weak_ptr` ne peuvent pas être déréférencés, ni comparés à nul. En effet, un `std::weak_ptr` n'est pas un pointeur intelligent autonome, mais une extension de `std::shared_ptr`.

Les liens sont établis dès la naissance. Les `std::weak_ptr` sont en général créés à partir de `std::shared_ptr`. Ils pointent sur les mêmes emplacements que les `std::shared_ptr` ayant servi à leur initialisation, mais ils n'affectent pas le compteur de références de l'objet ciblé :

```
auto spw =                          // Après la construction de spw,
  std::make_shared<Widget>();       // le compteur de références du
                                    // Widget pointé est égal à 1.
                                    // (Voir le conseil 21 pour des
                                    // infos sur std::make_shared.)
…

std::weak_ptr<Widget> wpw(spw);     // wpw pointe sur le même Widget
                                    // que spw. Le compteur reste à 1.
…

spw = nullptr;                      // Le compteur passe à 0
                                    // et le Widget est détruit.
                                    // wpw pendouille à présent.
```

Les `std::weak_ptr` qui pointent dans le vide sont dits *périmés*. Il est possible de tester directement cet état :

```
if (wpw.expired()) …                // Si wpw ne pointe pas
                                    // sur un objet…
```

Mais nous souhaitons le plus souvent vérifier si un `std::weak_ptr` est périmé et, dans la négative (autrement dit, s'il ne pendouille pas), accéder à l'objet pointé. C'est plus facile à souhaiter qu'à réaliser. Puisque les `std::weak_ptr` ne peuvent pas être déréférencés, il n'est pas possible d'écrire ce code. Et même s'ils disposaient de cet opérateur, la séparation du contrôle et du déréférencement introduirait une condition de concurrence : entre l'appel à `expired` et le déréférencement, un autre thread pourrait réaffecter ou détruire le dernier `std::shared_ptr` qui pointe sur l'objet, provoquant ainsi la destruction de celui-ci. Le déréférencement mènerait alors à un comportement indéfini.

Nous avons donc besoin d'une opération atomique qui vérifie si le `std::weak_ptr` est périmé et, dans le cas contraire, nous donne accès à l'objet pointé. Pour cela, nous

Copyright © 2016 Dunod.

créons un `std::shared_ptr` à partir du `std::weak_ptr`. Cette opération existe sous deux formes, dont le choix dépend du comportement souhaité si le `std::weak_ptr` est périmé lorsque nous l'utilisons pour créer un `std::shared_ptr`. La première variante est `std::weak_ptr::lock`, qui retourne un `std::shared_ptr`. Ce `std::shared_ptr` est nul si le `std::weak_ptr` est périmé :

```
std::shared_ptr<Widget> spw1 = wpw.lock();  // Si wpw est périmé,
                                            // spw1 est nul.

auto spw2 = wpw.lock();                     // Comme ci-dessus,
                                            // mais avec auto.
```

La seconde variante est apportée par le constructeur de `std::shared_ptr` qui prend un `std::weak_ptr` en argument. Dans ce cas, si le `std::weak_ptr` est périmé, une exception est levée :

```
std::shared_ptr<Widget> spw3(wpw);    // Si wpw est périmé,
                                      // lancer std::bad_weak_ptr.
```

Mais tout cela n'explique pas l'utilité des `std::weak_ptr`. Examinons une fonction fabrique qui produit des pointeurs intelligents sur des objets en lecture seule à partir d'un identifiant unique. Conformément au conseil 18, elle retourne un `std::unique_ptr` :

```
std::unique_ptr<const Widget> loadWidget(WidgetID id);
```

Si `loadWidget` est une opération coûteuse (par exemple elle effectue des entrées-sorties sur un fichier ou une base de données) et si les identifiants sont souvent utilisés de façon répétée, une optimisation normale serait d'écrire une fonction qui réalise le traitement de `loadWidget` et place ses résultats dans un cache. Cependant, encombrer le cache avec chaque `Widget` qui aura été demandé risque de poser des problèmes de performances. Une optimisation raisonnable serait donc de détruire les `Widget` du cache lorsqu'ils ne sont plus utilisés.

Pour cette fonction fabrique avec cache, le type de retour `std::unique_ptr` ne convient pas. Le code appelant doit effectivement recevoir des pointeurs intelligents sur les objets mis dans le cache et il doit fixer le cycle de vie de ces objets, mais le cache a également besoin d'un pointeur sur les objets. Les pointeurs du cache doivent être en mesure de détecter lorsqu'ils pendouillent. En effet, lorsque les clients de la fabrique n'ont plus besoin d'un objet qu'elle a retourné, celui-ci doit être détruit. L'entrée correspondante dans le cache pointe alors dans le vide. Les pointeurs mis dans le cache doivent donc être des `std::weak_ptr`, c'est-à-dire des pointeurs qui détectent s'ils pendouillent. Cela signifie que la fabrique doit retourner un `std::shared_ptr`, car les `std::weak_ptr` ne peuvent détecter cet état que si la vie d'un objet est gérée par des `std::shared_ptr`.

Voici une implémentation rapide de `loadWidget` avec la gestion d'un cache :

Copyright © 2016 Dunod.
© Dunod – Toute reproduction non autorisée est un délit.

```
std::shared_ptr<const Widget> fastLoadWidget(WidgetID id)
{
  static std::unordered_map<WidgetID,
                            std::weak_ptr<const Widget>> cache;

  auto objPtr = cache[id].lock();   // objPtr est un std::shared_ptr
                                    // sur un objet en cache (ou nul
                                    // si l'objet ne s'y trouve pas).

  if (!objPtr) {                    // Si absent du cache,
    objPtr = loadWidget(id);        // le charger,
    cache[id] = objPtr;             // le mettre en cache.
  }
  return objPtr;
}
```

Cette version se fonde sur une table de hachage de C++11 (`std::unordered_map`), même si elle ne montre pas les fonctions de hachage et de comparaison d'égalité de `WidgetID` qui sont requises.

L'implémentation de `fastLoadWidget` ignore le fait que le cache puisse avoir accumulé des `std::weak_ptr` périmés qui correspondent à des `Widget` qui ne sont plus utilisés (et qui ont donc été détruits). La mise en œuvre peut être améliorée, mais au lieu de passer du temps sur un problème qui n'apporte aucune information supplémentaire sur les `std::weak_ptr`, intéressons-nous à un second cas d'utilisation : le design pattern Observateur. Les principaux composants de ce pattern sont les observables (objets qui peuvent changer d'état) et les observateurs (objets avertis d'un changement d'état). Dans la plupart des implémentations, chaque observable contient une donnée membre qui mémorise des pointeurs sur ses observateurs. La génération des notifications de changement d'état est ainsi aisée. Les observables n'ont aucun intérêt à contrôler le cycle de vie de leurs observateurs (c'est-à-dire lorsqu'ils sont détruits), mais ils ont tout intérêt à s'assurer que si un observateur est détruit, ils ne tentent plus d'y accéder. Nous pouvons alors concevoir chaque observable en lui attribuant un conteneur de `std::weak_ptr` sur ses observateurs, pour qu'il puisse déterminer si un pointeur pendouille avant de l'utiliser.

Examinons un dernier exemple de l'utilité des `std::weak_ptr`. Prenons une structure de données qui contient des objets `A`, `B` et `C`, et où `A` et `C` partagent la propriété de `B` et possèdent donc des `std::shared_ptr` sur cet objet (figure 4.3).

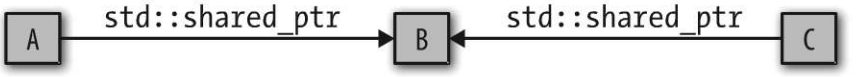


Figure 4.3 — Partage d'un objet *via* des `std::shared_ptr`

Copyright © 2016 Dunod.

Supposons que nous ayons également besoin d'un pointeur de `B` vers `A`. Quelle forme de pointeur doit-on choisir (figure 4.4) ?

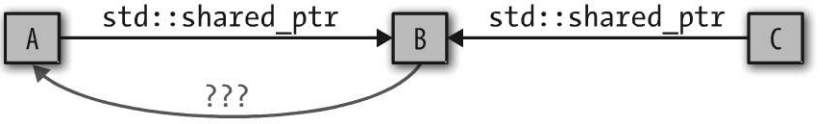


Figure 4.4 — Quel type de pointeur de retour pour éviter les cycles ?

Nous avons trois possibilités :

- **Un pointeur brut.** Dans cette approche, si `A` est détruit et si `C` continue à pointer sur `B`, le pointeur de `B` sur `A` pendouillera. `B` ne pourra pas détecter ce fait et risque de déréférencer par inadvertance un pointeur dans le vide. Cela mènera à un comportement indéfini.
- **Un `std::shared_ptr`.** Avec cette conception, `A` et `B` contiennent des pointeurs `std::shared_ptr` sur chacun d'eux. Le cycle de `std::shared_ptr` résultant (`A` pointe sur `B` et `B` pointe sur `A`) empêche la destruction de `A` et de `B`. Même si `A` et `B` ne peuvent plus être atteints à partir des autres structures de données du programme (par exemple parce que `C` ne pointe plus sur `B`), chacun aura un compteur de références égal à un. Dans ce cas, `A` et `B` sont à l'origine d'une fuite de ressources : le programme ne peut plus y accéder et leurs ressources ne peuvent pas être libérées.
- **Un `std::weak_ptr`.** Cette solution évite les deux problèmes précédents. Si `A` est détruit, le pointeur de `B` sur `A` pendouille, mais `B` est capable de le détecter. Par ailleurs, même si `A` et `B` pointent l'un sur l'autre, le pointeur de `B` n'affecte pas le compteur de références de `A`, et `A` peut donc être détruit lorsque plus aucun `std::shared_ptr` ne pointe dessus.

Les `std::weak_ptr` constituent manifestement le meilleur choix. Toutefois, nous devons préciser que la nécessité d'employer des `std::weak_ptr` pour éviter des cycles éventuels de `std::shared_ptr` est peu courante. Dans les structures de données strictement hiérarchiques, comme les arbres, les nœuds enfants sont généralement détenus uniquement par leurs parents. Lorsqu'un nœud parent est détruit, ses nœuds enfants doivent l'être également. Les `std::unique_ptr` conviennent donc à la représentation des liens depuis les parents vers les enfants. Les liens de retour des enfants vers les parents peuvent être mis en œuvre avec des pointeurs bruts, car un nœud enfant ne doit pas avoir une durée de vie plus longue que celle de son parent. Il n'y a donc aucun risque qu'un nœud enfant ne déréférence un pointeur dans le vide sur son parent.

Toutes les structures de données à base de pointeurs ne sont évidemment pas strictement hiérarchiques et, lorsque ce n'est le cas, ainsi que dans d'autres situations comme celles du cache et de la liste des observateurs, il est bon de savoir que les `std::weak_ptr` sont prêts à servir.

Sur le plan de l'efficacité, un `std::weak_ptr` équivaut à un `std::shared_ptr`. La taille des objets `std::weak_ptr` est identique à celle des objets `std::shared_ptr`, ils utilisent les mêmes blocs de contrôle (voir le conseil 19), et les opérations comme

Copyright © 2016 Dunod.
© Dunod – Toute reproduction non autorisée est un délit.

leur construction, leur destruction et leur affectation imposent des manipulations atomiques du compteur de références. Cela vous surprend peut-être, car nous avons indiqué au début de ce conseil que les `std::weak_ptr` n'interviennent pas dans le comptage des références. Mais nous n'avons pas dit exactement cela. Nous avons précisé que les `std::weak_ptr` ne participaient pas au *partage de la propriété* des objets et qu'ils n'avaient donc pas d'incidence sur le *compteur de références de l'objet pointé*. En réalité, le bloc de contrôle comprend un second compteur de références et c'est celui-ci que les `std::weak_ptr` manipulent. Le conseil 21 revient en détail sur ce sujet.

À retenir

- Utiliser des `std::weak_ptr` lorsque des pointeurs de type `std::shared_ptr` peuvent pendouiller.
- Les cas d'utilisation des `std::weak_ptr` comprennent notamment la mise en cache, les listes d'observateurs et la prévention de cycle de `std::shared_ptr`.

CONSEIL N° 21. PRÉFÉRER `STD::MAKE_UNIQUE` ET `STD::MAKE_SHARED` À UNE UTILISATION DIRECTE DE `NEW`

Commençons par établir les terrains de jeu de `std::make_unique` et de `std::make_shared`. `std::make_shared` fait partie de C++11, contrairement à `std::make_unique`. Il a rejoint la bibliothèque standard avec C++14. Si vous utilisez C++11, ne paniquez pas car l'écriture d'une version basique de `std::make_unique` n'a rien de complexe. La voici :

```
template<typename T, typename... Ts>
std::unique_ptr<T> make_unique(Ts&&... params)
{
  return std::unique_ptr<T>(new T(std::forward<Ts>(params)...));
}
```

Vous le constatez, `make_unique` se contente de relayer parfaitement ses paramètres au constructeur de l'objet à créer, construit un `std::unique_ptr` à partir du pointeur brut obtenu de `new`, et renvoie ce `std::unique_ptr` créé. Une telle version ne prend pas en charge les tableaux ni les supprimeurs personnalisés (voir le conseil 18), mais elle montre que, en cas de besoin, peu d'efforts suffisent à écrire une fonction `make_unique`[1]. Faites attention à ne pas placer votre version dans l'espace de noms `std`, car elle entrerait en conflit avec celle de la bibliothèque standard de C++14 au moment de la mise à niveau.

1. Pour créer une fonction `make_unique` qui offre toutes les fonctionnalités, toujours avec le minimum d'effort possible, recherchez son implémentation dans le document de normalisation qui la met en exergue, puis copiez-la. Ce document rédigé par Stephan T. Lavavej possède le numéro N3656, en date du 18 avril 2013.

Copyright © 2016 Dunod.

`std::make_unique` et `std::make_shared` sont deux des trois *fonctions* `make` : des fonctions qui prennent un jeu d'arguments quelconque, les retransmettent parfaitement au constructeur d'un objet alloué dynamiquement, et retournent un pointeur intelligent sur cet objet. La troisième fonction `make` se nomme `std::allocate_shared`. Elle opère comme `std::make_shared`, mais son premier argument est un objet d'allocation qui se charge de la réservation dynamique de la mémoire.

Une comparaison simple de la création d'un pointeur intelligent avec et sans passer par une fonction `make` révèle déjà l'intérêt d'une telle fonction :

```
auto upw1(std::make_unique<Widget>());      // Avec une fonction make.

std::unique_ptr<Widget> upw2(new Widget);   // Sans fonction make.


auto spw1(std::make_shared<Widget>());      // Avec une fonction make.

std::shared_ptr<Widget> spw2(new Widget);   // Sans fonction make.
```

Nous avons mis en exergue la différence principale : les versions fondées sur `new` répètent le type créé, contrairement à celles basées sur les fonctions `make`. La répétition des types va à l'encontre d'un principe essentiel de l'ingénierie logicielle : la duplication du code doit être évitée. En effet, elle augmente le temps de compilation, produit du code objet lourd et, généralement, rend la manipumation de la base de code plus difficile. Elle mène souvent à du code incohérent, et les incohérences dans une base de code conduisent souvent à des bogues. Par ailleurs, la saisie en double demande plus d'efforts, alors que l'objectif des programmeurs est plutôt de réduire cette charge de travail.

La seconde raison de préférer les fonctions `make` est en lien avec la sécurité vis-à-vis des exceptions. Supposons que nous ayons une fonction qui traite un `Widget` en fonction de la priorité indiquée :

```
void processWidget(std::shared_ptr<Widget> spw, int priority);
```

Le passage par valeur du `std::shared_ptr` pourrait sembler douteux, mais le conseil 41 explique que si `processWidget` effectue toujours une copie du `std::shared_ptr` (par exemple en le stockant dans une structure de données qui sert au suivi des `Widget` traités), ce choix de conception peut être approprié.

Nous disposons également d'une fonction qui calcule la priorité requise :

```
int computePriority();
```

Supposons que nous l'utilisions dans un appel à `processWidget` fondé sur `new` à la place de `std::make_shared` :

Copyright © 2016 Dunod.
© Dunod – Toute reproduction non autorisée est un délit.

```
processWidget(std::shared_ptr<Widget>(new Widget),  // Fuite de
              computePriority());                   // ressource
                                                    // potentielle !
```

Comme l'indique le commentaire, ce code peut perdre le `Widget` produit par `new`. Par quel processus ? En effet, le code appelant et la fonction appelée emploient des `std::shared_ptr` et les `std::shared_ptr` sont conçus pour éviter les fuites de ressources. Ils détruisent automatiquement l'élément ciblé lorsque le dernier `std::shared_ptr` qui pointe dessus disparaît. Puisque tout le monde utilise systématiquement des `std::shared_ptr`, d'où vient la fuite ?

La réponse se trouve dans la manière dont le compilateur convertit le code source en code objet. À l'exécution, les arguments d'une fonction doivent être évalués avant que celle-ci ne puisse être invoquée. Par conséquent, dans l'appel à `processWidget`, voici les opérations qui doivent avoir lieu avant que l'exécution de `processWidget` ne débute :

- L'expression « `new Widget` » est évaluée, ce qui déclenche la création d'un `Widget` sur le tas.
- Le constructeur de `std::shared_ptr<Widget>` responsable de la gestion du pointeur fourni par `new` est exécuté.
- La fonction `computePriority` est exécutée.

Le compilateur n'est pas obligé de générer un code qui effectue ces opérations dans cet ordre. « `new Widget` » doit se produire avant l'appel au constructeur de `std::shared_ptr`, car le résultat de `new` sert d'argument à ce constructeur, mais il est possible d'exécuter `computePriority` avant, après ou, point essentiel, entre ces appels. Autrement dit, le code produit par le compilateur peut effectuer des opérations dans l'ordre suivant :

1. Exécuter « `new Widget` ».
2. Appeler `computePriority`.
3. Invoquer le constructeur de `std::shared_ptr`.

Supposons que ce code soit généré et que, pendant l'exécution, `computePriority` lève une exception. Dans ce cas, l'objet `Widget` alloué dynamiquement à l'étape 1 sera perdu, car il ne sera jamais stocké dans le `std::shared_ptr` qui est censé le gérer à l'étape 3.

En utilisant `std::make_shared`, nous évitons ce problème. Voici le code correspondant :

```
processWidget(std::make_shared<Widget>(),     // Aucune fuite de
              computePriority());             // ressource.
```

À l'exécution, soit `std::make_shared`, soit `computePriority` est appelée en premier. S'il s'agit de `std::make_shared`, le pointeur brut sur le `Widget` alloué dynamiquement est stocké en toute sécurité dans le `std::shared_ptr` renvoyé avant que `computePriority` ne soit appelée. Dans le cas où `computePriority` lance une exception, le

Copyright © 2016 Dunod.

destructeur de `std::shared_ptr` veillera à ce que le `Widget` qu'il détient soit détruit. Si `computePriority` est appelée en premier et lève une exception, `std::make_shared` ne sera pas invoquée et aucun `Widget` ne sera alloué dynamiquement.

Si nous remplaçons `std::shared_ptr` et `std::make_shared` par `std::unique_ptr` et `std::make_unique`, nous pouvons tenir le même raisonnement. Pour écrire du code sûr vis-à-vis des exceptions, il est tout aussi important d'utiliser `std::make_unique` à la place de `new` que d'utiliser `std::make_shared`.

En comparaison d'une utilisation directe de `new`, `std::make_shared` présente l'avantage d'améliorer l'efficacité du code. En effet, le compilateur peut générer un code plus concis et plus rapide qui se fonde sur des structures de données plus légères. Examinons un appel direct à `new` :

```
std::shared_ptr<Widget> spw(new Widget);
```

Il est évident qu'une allocation de mémoire a lieu, mais, en réalité, ce code en effectue deux. Le conseil 19 explique que chaque `std::shared_ptr` pointe sur un bloc de contrôle qui contient, entre autres, le compteur de références associé à l'objet ciblé. La zone de mémoire réservée à ce bloc de contrôle est allouée par le constructeur de `std::shared_ptr`. Autrement dit, l'utilisation directe de `new` nécessite une première allocation de mémoire pour le `Widget` et une seconde pour le bloc de contrôle.

Supposons à présent que nous utilisions `std::make_shared` :

```
auto spw = std::make_shared<Widget>();
```

Dans ce cas, une seule allocation suffit. En effet, `std::make_shared` alloue une seule zone de mémoire qui contiendra l'objet `Widget` et le bloc de contrôle. Cette optimisation réduit la taille statique du programme, car le code comprend un seul appel à l'allocation de la mémoire, et elle augmente sa vitesse d'exécution, car l'allocation est effectuée une seule fois. Par ailleurs, en utilisant `std::make_shared`, certaines informations de gestion ne sont plus nécessaires dans le bloc de contrôle et l'empreinte mémoire globale du programme s'en trouve donc potentiellement diminuée.

Puisque ce constat d'efficacité de `std::make_shared` concerne également `std::allocate_shared`, cette fonction permet aussi d'améliorer les performances.

Nous avons donc toutes les raisons de préférer les fonctions `make` à une utilisation directe de `new`. Toutefois, malgré leurs avantages sur le plan de l'ingénierie logicielle, de la sécurité vis-à-vis des exceptions et de l'efficacité, ce conseil préconise de *préférer* les fonctions `make`, non de les utiliser systématiquement. Il existe en effet des situations où elles ne peuvent pas ou ne doivent pas être employées.

Par exemple, aucune des fonctions `make` ne permet de spécifier des supprimeurs personnalisés (voir les conseils 18 et 19), mais `std::unique_ptr` et `std::shared_ptr` offrent des constructeurs qui les prennent en charge. Supposons le supprimeur personnalisé suivant pour un `Widget` :

```
auto widgetDeleter = [](Widget* pw) { … };
```

Copyright © 2016 Dunod.
© Dunod – Toute reproduction non autorisée est un délit.

Avec new, il est très facile de créer un pointeur intelligent qui l'utilise :

```
std::unique_ptr<Widget, decltype(widgetDeleter)>
  upw(new Widget, widgetDeleter);

std::shared_ptr<Widget> spw(new Widget, widgetDeleter);
```

Avec une fonction make, cette possibilité n'existe pas.

Une seconde restriction des fonctions make provient d'un détail syntaxique dans leur implémentation. Le conseil 7 explique que lors de la création d'un objet dont le type surcharge des constructeurs avec et sans des paramètres std::initializer_list, la création avec des accolades choisit le constructeur avec std::initializer_list, tandis que la création avec des parenthèses se fonde sur le constructeur sans std::initializer_list. Les fonctions make retransmettent parfaitement leurs paramètres au constructeur de l'objet, mais emploient-elles pour cela des parenthèses ou des accolades ? Selon les types des arguments, la réponse à cette question fera une grande différence. Prenons par exemple les appels suivants :

```
auto upv = std::make_unique<std::vector<int>>(10, 20);

auto spv = std::make_shared<std::vector<int>>(10, 20);
```

Les pointeurs intelligents obtenus pointent-ils sur des std::vector de 10 éléments, chacun ayant la valeur 20, ou sur des std::vector de 2 éléments, l'un ayant la valeur 10, l'autre, la valeur 20 ? Ou, le résultat est-il indéterminé ?

Bonne nouvelle, le résultat n'est pas indéterminé : les deux appels créent des std::vector de 10 éléments, tous fixés à la valeur 20. Cela signifie que le code de retransmission parfaite dans les fonctions make utilise non pas des accolades mais des parenthèses. Mauvaise nouvelle : si nous voulons construire l'objet pointé en utilisant un initialiseur à accolades, nous devons appeler directement new. Pour employer une fonction make, il faudrait pouvoir retransmettre parfaitement un initialiseur à accolades, mais, comme l'explique le conseil 30, ce n'est pas possible avec ce type d'initialiseur. Le conseil 30 propose toutefois une alternative : exploiter la déduction de type auto pour créer un objet std::initializer_list à partir d'un initialiseur à accolades (voir le conseil 2), puis passer l'objet créé par auto au travers de la fonction make :

```
// Créer un std::initializer_list.
auto initList = { 10, 20 };

// Créer un std::vector en utilisant le constructeur de
// std::initializer_list.
auto spv = std::make_shared<std::vector<int>>(initList);
```

Pour std::unique_ptr, voilà les deux seuls scénarios (supprimeurs personnalisés et initialiseurs à accolades) dans lesquels ces fonctions make posent des difficultés. Pour std::shared_ptr et ces fonctions make, il existe deux cas supplémentaires, qui sont

Copyright © 2016 Dunod.

peut-être limites, mais certains développeurs prennent des risques et vous pourriez en faire partie.

Certaines classes définissent leur propre version de `operator new` et de `operator delete`. L'existence de ces fonctions sous-entend que les routines globales d'allocation et de désallocation de la mémoire ne sont pas adaptées aux objets de ce type. Le plus souvent, les routines spécifiques à une classe sont conçues uniquement pour allouer et désallouer des zones de mémoire dont la taille correspond précisément à celle des objets de la classe. Par exemple, les fonctions `operator new` et `operator delete` de la classe `Widget` sont faites uniquement pour allouer et désallouer des morceaux de mémoire dont la taille est égale à `sizeof(Widget)`. Ces routines s'accordent mal à la prise en charge de l'allocation (*via* `std::allocate_shared`) et de la désallocation (*via* des supprimeurs personnalisés) personnalisées de `std::shared_ptr`, car la quantité de mémoire demandée par `std::allocate_shared` ne correspond pas à la taille de l'objet alloué dynamiquement, mais à cette taille *plus* celle d'un bloc de contrôle. Par conséquent, utiliser des fonctions `make` pour créer des objets dont la classe dispose de versions spécifiques de `operator new` et de `operator delete` est souvent une mauvaise idée.

Les avantages en termes de taille et de rapidité de `std::make_shared` par rapport à une utilisation directe de `new` proviennent du placement du bloc de contrôle `std::shared_ptr` dans la même zone de mémoire que l'objet géré. Lorsque son compteur de références atteint zéro, l'objet est détruit (son destructeur est appelé). Cependant, la mémoire qu'il occupe ne peut pas être libérée tant que le bloc de contrôle n'a pas également été détruit, car elle contient ces deux éléments.

Le bloc de contrôle ne contient pas uniquement le compteur de références mais également d'autres informations de gestion. Le compteur de références permet le suivi du nombre de `std::shared_ptr` qui font référence au bloc de contrôle, mais celui-ci contient un second compteur qui dénombre les `std::weak_ptr` qui font référence au bloc de contrôle. Ce second compteur de références est appelé le *compteur faible*[1]. Lorsqu'un `std::weak_ptr` vérifie s'il est périmé (voir le conseil 19), il examine le compteur de références (non le compteur faible) dans le bloc de contrôle associé. Si ce compteur est à zéro (autrement dit si plus aucun `std::shared_ptr` ne pointe sur l'objet, qui a donc été détruit), cela signifie que le `std::weak_ptr` est périmé.

Tant que des `std::weak_ptr` font référence à un bloc de contrôle (c'est-à-dire que le compteur faible est supérieur à zéro), celui-ci doit continuer à exister. Et, tant qu'un bloc de contrôle existe, la mémoire qui le contient doit rester allouée. La zone de mémoire allouée par la fonction `std::shared_ptr` `make` ne peut donc pas être libérée tant que le dernier `std::shared_ptr` *et* le dernier `std::weak_ptr` qui y font référence n'ont pas été détruits.

1. Dans la pratique, la valeur du compteur faible n'est pas toujours égale au nombre de `std::weak_ptr` qui font référence au bloc de contrôle, car les développeurs de bibliothèques ont trouvé des manières de glisser des informations supplémentaires dans ce compteur de façon à obtenir du code plus efficace. Dans le cadre de cette section, nous ignorons cela et supposons que la valeur du compteur faible correspond au nombre de `std::weak_ptr` qui font référence au bloc de contrôle.

© Dunod – Toute reproduction non autorisée est un délit.
Copyright © 2016 Dunod.

Si la taille du type de l'objet est assez importante et si le temps entre la destruction du dernier `std::shared_ptr` et du dernier `std::weak_ptr` est relativement long, un décalage peut se produire entre le moment où l'objet est détruit et celui où la mémoire qu'il occupait est libérée :

```
class ReallyBigType { … };

auto pBigObj =                          // Créer un très grand
  std::make_shared<ReallyBigType>();    // objet par le biais
                                        // de std::make_shared.

…       // Créer des std::shared_ptr et des std::weak_ptr sur
        // l'objet volumineux et les utiliser pour le manipuler.

…       // Le dernier std::shared_ptr sur l'objet est détruit ici,
        // mais il reste des std::weak_ptr.

…       // Au cours de cette période, la mémoire initialement
        // occupée par l'objet volumineux reste allouée.

…       // Le dernier std::weak_ptr sur l'objet est détruit ici ;
        // la mémoire pour le bloc de contrôle et l'objet est libérée.
```

En utilisant directement `new`, la mémoire réservée à l'objet `ReallyBigType` peut être libérée dès que le dernier `std::shared_ptr` associé à l'objet est détruit :

```
class ReallyBigType { … };              // Comme précédemment.

std::shared_ptr<ReallyBigType> pBigObj(new ReallyBigType);
                                        // Créer un très grand
                                        // objet avec new.

…       // Comme précédemment, créer des std::shared_ptr et
        // des std::weak_ptr sur l'objet et les utiliser.

…       // Le dernier std::shared_ptr sur l'objet est détruit ici,
        // mais il reste des std::weak_ptr ;
        // la mémoire réservée à l'objet est désallouée.

…       // Au cours de cette période, seule la mémoire réservée
        // au bloc de contrôle reste allouée.

…       // Le dernier std::weak_ptr sur l'objet est détruit ici ;
        // la mémoire pour le bloc de contrôle est libérée.
```

Si nous nous trouvons dans une situation où l'utilisation de `std::make_shared` est impossible ou inappropriée, nous voudrons néanmoins éviter les problèmes de sécurité vis-à-vis des exceptions décrits précédemment. Pour cela, la meilleure solution consiste à passer immédiatement le résultat de l'utilisation directe de `new` au constructeur d'un pointeur intelligent dans *une instruction qui ne fait rien d'autre*. De cette manière, le compilateur ne générera pas du code qui pourrait lancer une exception entre

Copyright © 2016 Dunod.

l'utilisation de `new` et l'invocation du constructeur du pointeur intelligent en charge de l'objet alloué par `new`.

Prenons comme exemple une version légèrement modifiée de l'appel à la fonction `processWidget` qui n'était pas sûre vis-à-vis des exceptions. Cette fois-ci, nous spécifions un supprimeur personnalisé :

```
void processWidget(std::shared_ptr<Widget> spw,  // Comme précédemment.
                   int priority);

void cusDel(Widget *ptr);                        // Supprimeur
                                                 // personnalisé.
```

Voici l'appel non sûr vis-à-vis des exceptions :

```
processWidget(                                   // Comme précédemment,
  std::shared_ptr<Widget>(new Widget, cusDel),   // fuite de
  computePriority()                              // ressources
);                                               // potentielle !
```

Petit rappel : si `computePriority` est appelée après « `new Widget` » mais avant le constructeur de `std::shared_ptr` et si `computePriority` lance une exception, le `Widget` alloué dynamiquement est perdu.

Puisque l'utilisation d'un supprimeur personnalisé empêche celle de `std::make_shared`, la manière d'éviter le problème consiste à placer l'allocation du `Widget` et la construction du `std::shared_ptr` dans leur propre instruction, puis d'appeler `processWidget` avec le `std::shared_ptr` obtenu. Voici les bases de la technique, mais, comme nous le verrons plus loin, il est possible d'en améliorer les performances :

```
std::shared_ptr<Widget> spw(new Widget, cusDel);

processWidget(spw, computePriority());   // Correct, sans être optimal ;
                                         // voir ci-après.
```

Cela fonctionne, car un `std::shared_ptr` assume la propriété du pointeur brut passé à son constructeur, même si celui-ci lève une exception. Dans cet exemple, si le constructeur de `spw` lance une exception (par exemple en raison de l'impossibilité d'allouer dynamiquement de la mémoire pour un bloc de contrôle), l'invocation de `cusDel` sur le pointeur fourni par « `new Widget` » est garantie.

Le petit souci de performance vient du fait que, dans l'appel non sûr vis-à-vis des exceptions, nous passons une rvalue à `processWidget` :

```
processWidget(
  std::shared_ptr<Widget>(new Widget, cusDel),  // L'argument est une
                                                // rvalue.
  computePriority()
);
```

Copyright © 2016 Dunod.
© Dunod – Toute reproduction non autorisée est un délit.

alors que, dans l'appel sûr vis-à-vis des exceptions, nous passons une lvalue :

```
processWidget(spw, computePriority());              // L'argument est une
                                                    // lvalue.
```

Puisque le paramètre `std::shared_ptr` est passé par valeur à `processWidget`, la construction à partir d'une rvalue demande seulement un déplacement, tandis que celle à partir d'une lvalue impose une copie. Pour un `std::shared_ptr`, la différence peut être significative, car copier un `std::shared_ptr` implique une incrémentation atomique de son compteur de références, tandis que le déplacer n'en demande aucune manipulation. Pour que le code sûr vis-à-vis des exceptions arrive au même niveau de performance que le code non sûr vis-à-vis des exceptions, nous devons appliquer `std::move` à `spw` afin de le convertir en rvalue (voir le conseil 23) :

```
processWidget(std::move(spw),              // À la fois efficace et sûr
              computePriority());          // vis-à-vis des exceptions.
```

C'est intéressant et bon à savoir, mais cela reste souvent inapproprié car les raisons de ne pas utiliser une fonction `make` sont plutôt rares. Et, à moins qu'une telle raison existe vraiment, il faut utiliser une fonction `make`.

À retenir

- Au contraire de l'utilisation directe de `new`, les fonctions `make` éliminent la duplication du code source, améliorent la sécurité vis-à-vis des exceptions et, pour `std::make_shared` et `std::allocate_shared`, permettent d'obtenir un code plus concis et plus rapide.
- Les fonctions `make` ne conviennent pas lorsque des supprimeurs personnalisés ou des initialiseurs à accolades sont requis.
- Pour les `std::shared_ptr`, il existe d'autres cas où les fonctions `make` sont inappropriées : (1) classes avec une gestion personnalisée de la mémoire et (2) systèmes où l'occupation mémoire importe, où des objets sont très volumineux et où des `std::weak_ptr` vivent plus longtemps que les `std::shared_ptr` correspondants.

CONSEIL N° 22. AVEC L'IDIOME PIMPL, DÉFINIR DES FONCTIONS MEMBRES SPÉCIALES DANS LE FICHIER D'IMPLÉMENTATION

Si vous avez déjà eu à résoudre des problèmes de temps de construction excessif, l'*idiome Pimpl* (*pointer to implementation*) ne vous est certainement pas inconnu. Cette technique consiste à remplacer les données membres d'une classe par un pointeur sur une classe (ou une structure) d'implémentation, à placer dans celle-ci les données membres qui se trouvaient dans la classe initiale, et à accéder à ces données membres

Copyright © 2016 Dunod.

indirectement au travers du pointeur. Supposons, par exemple, que la classe `Widget` soit écrite de la manière suivante :

```
class Widget {                          // Dans l'en-tête "widget.h".
public:
  Widget();
  …
private:
  std::string name;
  std::vector<double> data;
  Gadget g1, g2, g3;                    // Gadget est un type défini
};                                      // par l'utilisateur.
```

Puisque les données membres de `Widget` ont les types `std::string`, `std::vector` et `Gadget`, les en-têtes de ces types doivent être présents pour que la classe `Widget` puisse être compilée. Autrement dit, le code qui utilise `Widget` doit inclure `<string>`, `<vector>` et `gadget.h`. Ces fichiers d'en-tête augmentent le temps de compilation des clients de `Widget` et les rendent dépendants de leur contenu. Si un fichier d'en-tête est modifié, les clients de `Widget` doivent être recompilés. Les en-têtes standard `<string>` et `<vector>` seront rarement modifiés, mais il est possible que `gadget.h` fasse l'objet de révisions fréquentes.

Pour appliquer l'idiome Pimpl en C++98, nous remplaçons les données membres de `Widget` par un pointeur brut sur une structure déclarée mais non définie :

```
class Widget {                          // Toujours dans l'en-tête "widget.h".
public:
  Widget();
  ~Widget();                            // Destructeur requis (voir ci-après).
  …

private:
  struct Impl;                          // Déclarer la structure d'implémentation
  Impl *pImpl;                          // et le pointeur associé.
};
```

Puisque les types `std::string`, `std::vector` et `Gadget` ne sont plus mentionnés dans `Widget`, le code qui utilise cette classe n'a plus besoin d'inclure les fichiers d'en-tête pour ces types. La compilation s'en trouve accélérée et, si ces en-têtes sont modifiés, les clients de `Widget` n'en sont pas affectés.

Un type qui a été déclaré sans avoir été défini est appelé *type incomplet*. `Widget::Impl` en est un exemple. Les utilisations d'un type incomplet sont peu nombreuses, mais l'une d'elles est de déclarer un pointeur sur ce type. L'idiome Pimpl se fonde sur cette possibilité.

La première partie de l'idiome Pimpl correspond à la déclaration d'une donnée membre qui est un pointeur sur un type incomplet. La seconde partie implique l'allocation dynamique et la désallocation de l'objet qui contient les données membres présentes auparavant dans la classe d'origine. Le code d'allocation et de désallocation

Copyright © 2016 Dunod.
© Dunod – Toute reproduction non autorisée est un délit.

va dans le fichier d'implémentation. Par exemple, pour `Widget`, il s'agit du fichier `widget.cpp` :

```
#include "widget.h"            // Dans le fichier d'implémentation
#include "gadget.h"            // "widget.cpp".
#include <string>
#include <vector>

struct Widget::Impl {          // Définition de Widget::Impl
  std::string name;            // avec les données membres
  std::vector<double> data;    // précédemment dans Widget.
  Gadget g1, g2, g3;
};

Widget::Widget()               // Allouer les données membres
: pImpl(new Impl)              // pour cet objet Widget.
{}

Widget::~Widget()              // Détruire les données membres
{ delete pImpl; }              // pour cet objet.
```

Nous montrons les directives `#include` afin que les choses soient claires : les dépendances globales avec les en-têtes pour `std::string`, `std::vector` et `Gadget` continuent à exister. Cependant, elles ne sont plus dans `widget.h` (qui est vu et utilisé par le code client de `Widget`) mais dans `widget.cpp` (qui est vu et utilisé uniquement par le code d'implémentation de `Widget`). Nous illustrons également le code qui alloue dynamiquement et désalloue l'objet `Impl`. Puisque nous devons désallouer cet objet lors de la destruction d'un `Widget`, le destructeur de `Widget` est indispensable.

Mais ce code C++98 date d'une époque révolue. Il se fonde sur des pointeurs bruts et des appels à `new` et à `delete`. Ce chapitre s'articule autour de l'idée que les pointeurs intelligents sont meilleurs que les pointeurs bruts, et si nous voulons une allocation dynamique d'un objet `Widget::Impl` dans le constructeur de `Widget`, avec sa destruction en même temps que le `Widget`, un `std::unique_ptr` (voir le conseil 18) va répondre parfaitement à nos besoins. En remplaçant le pointeur brut `pImpl` par un `std::unique_ptr`, nous obtenons le code d'en-tête suivant :

```
class Widget {                        // Dans "widget.h".
public:
  Widget();
  …

private:
  struct Impl;
  std::unique_ptr<Impl> pImpl;        // Utiliser un pointeur intelligent
};                                    // à la place du pointeur brut.
```

Voici le fichier d'implémentation correspondant :

Copyright © 2016 Dunod.

```
#include "widget.h"                  // Dans "widget.cpp".
#include "gadget.h"
#include <string>
#include <vector>

struct Widget::Impl {                // Comme précédemment.
  std::string name;
  std::vector<double> data;
  Gadget g1, g2, g3;
};

Widget::Widget()                     // Conformément au conseil 21,
: pImpl(std::make_unique<Impl>())    // créer un std::unique_ptr
{}                                   // avec std::make_unique.
```

Vous l'aurez remarqué, le destructeur de `Widget` a disparu. En effet, nous n'avons plus de code à y mettre. `std::unique_ptr` supprime automatiquement l'élément sur lequel il pointe lorsqu'il (le `std::unique_ptr`) est détruit. Nous n'avons donc plus rien à supprimer par nous-mêmes. Voilà l'un des attraits des pointeurs intelligents : ils nous évitent toute implication dans la libération manuelle des ressources.

Si la compilation de ce code ne pose pas problème, ce n'est pas le cas pour le code client le plus simple :

```
#include "widget.h"

Widget w;                            // Erreur !
```

Le message d'erreur généré dépend du compilateur, mais il mentionne généralement l'application de `sizeof` ou de `delete` à un type incomplet. Ces opérations font partie des opérations impossibles sur de tels types.

Cette faillite apparente de l'idiome Pimpl fondé sur les `std::unique_ptr` est alarmante car (1) `std::unique_ptr` est censé prendre en charge les types incomplets et (2) l'idiome Pimpl est l'un des cas d'utilisation types des `std::unique_ptr`. Heureusement, la correction du code ne pose aucune difficulté. Il suffit simplement de comprendre l'origine du problème.

Elle se trouve dans le code généré lors de la destruction de `w` (par exemple, lorsqu'il est hors de portée) et que son destructeur est invoqué. Dans la définition de la classe qui utilise `std::unique_ptr`, nous n'avons déclaré aucun destructeur car nous n'avons aucun code à y placer. En accord avec les règles habituelles des fonctions membres spéciales générées par le compilateur (voir le conseil 17), celui-ci génère un destructeur à notre place. Il y insère du code qui appelle le destructeur de la donnée membre `pImpl` de `Widget`. `pImpl` est un `std::unique_ptr<Widget::Impl>`, c'est-à-dire un `std::unique_ptr` qui utilise le supprimeur par défaut. Celui-ci est une fonction qui appelle `delete` sur le pointeur brut contenu dans le `std::unique_ptr`. Cependant, avant d'utiliser `delete`, les implémentations du supprimeur par défaut se servent généralement de `static_assert` de C++11 pour vérifier que le pointeur brut ne vise pas un type incomplet. Lorsque le compilateur génère le code de destruction

© Dunod – Toute reproduction non autorisée est un délit.
Copyright © 2016 Dunod.

du `Widget w`, il rencontre généralement un `static_assert` qui échoue et qui conduit donc au message d'erreur. Le message est associé à l'emplacement de la destruction de `w` car le destructeur de `Widget`, comme toute fonction membre spéciale générée par le compilateur, est implicitement `inline`. Le message fait souvent référence à la ligne où `w` est créé car c'est le code source qui crée explicitement l'objet qui conduit à sa destruction implicite ultérieure.

Pour corriger le problème, il suffit de s'assurer qu'à l'endroit où le code de destruction du `std::unique_ptr<Widget::Impl>` est généré, `Widget::Impl` est un type complet. Pour qu'un type soit complet, sa définition doit avoir été rencontrée. Puisque `Widget::Impl` est défini dans `widget.cpp`, nous devons faire en sorte que le compilateur voie le corps du destructeur de `Widget` (c'est-à-dire l'endroit où le compilateur générera du code pour détruire la donnée membre `std::unique_ptr`) uniquement à l'intérieur de `widget.cpp`, après la définition de `Widget::Impl`.

Pour cela, nous déclarons le destructeur de `Widget` dans `widget.h`, mais sans le définir dans ce fichier :

```
class Widget {                  // Comme précédemment, dans "widget.h".
public:
  Widget();
  ~Widget();                    // Déclaration uniquement.
  …

private:                        // Comme précédemment.
  struct Impl;
  std::unique_ptr<Impl> pImpl;
};
```

Sa définition arrive dans `widget.cpp`, après celle de `Widget::Impl` :

```
#include "widget.h"             // Comme précédemment, dans "widget.cpp".
#include "gadget.h"
#include <string>
#include <vector>

  struct Widget::Impl {         // Comme précédemment, définition de
  std::string name;             // Widget::Impl.
  std::vector<double> data;
  Gadget g1, g2, g3;
};

Widget::Widget()                // Comme précédemment.
: pImpl(std::make_unique<Impl>())
{}

Widget::~Widget()               // Définition de ~Widget.
{}
```

Cette solution fonctionne parfaitement et nécessite peu de saisie. Mais, si nous voulons souligner le fait que le destructeur généré par le compilateur effectue le travail approprié et que la seule raison de le déclarer était que sa définition soit générée dans

Copyright © 2016 Dunod.

le fichier d'implémentation de `Widget`, nous pouvons définir le corps du destructeur avec « `= default` » :

```
Widget::~Widget() = default;       // Même résultat que précédemment.
```

Les classes qui se fondent sur l'idiome Pimpl sont parfaitement adaptées à la prise en charge du déplacement car les opérations correspondantes générées par le compilateur réalisent exactement ce que nous souhaitons : effectuer un déplacement du `std::unique_ptr` sous-jacent. Mais, comme l'explique le conseil 17, la déclaration d'un destructeur dans `Widget` empêche le compilateur de générer les opérations de déplacement. Par conséquent, pour prendre en charge le déplacement, nous devons déclarer ces fonctions nous-mêmes. Étant donné que les versions générées par le compilateur auraient le comportement approprié, nous pourrions être tentés de les implémenter de la manière suivante :

```
class Widget {                                    // Toujours dans
public:                                           // "widget.h".
  Widget();
  ~Widget();

  Widget(Widget&& rhs) = default;                 // Bonne idée,
  Widget& operator=(Widget&& rhs) = default;      // mauvais code !

  …

private:                                          // Comme précédemment.
  struct Impl;
  std::unique_ptr<Impl> pImpl;
};
```

Cette approche conduit à un problème comparable à celui de la déclaration de la classe sans destructeur, essentiellement pour la même raison. L'opérateur d'affectation par déplacement généré par le compilateur doit détruire l'objet pointé par `pImpl` avant de le réaffecter, mais, dans le fichier d'en-tête de `Widget`, `pImpl` point sur un type incomplet. Le cas du constructeur de déplacement est différent. Le problème vient du fait que le compilateur produit généralement un code pour détruire `pImpl` dans le cas où une exception surviendrait à l'intérieur du constructeur de déplacement, mais la destruction de `pImpl` exige que son type soit complet.

Puisque le problème est identique au précédent, nous pouvons le corriger de la même manière. Nous plaçons la définition des opérations de déplacement dans le fichier d'implémentation :

```
class Widget {                          // Toujours dans "widget.h".
public:
  Widget();
  ~Widget();

  Widget(Widget&& rhs);                 // Déclarations
  Widget& operator=(Widget&& rhs);      // uniquement.
```

Copyright © 2016 Dunod.
© Dunod – Toute reproduction non autorisée est un délit.

```
  …

private:                                    // Comme précédemment.
  struct Impl;
  std::unique_ptr<Impl> pImpl;
};

#include <string>                           // Comme précédemment,
…                                           // dans "widget.cpp".

struct Widget::Impl { … };                  // Comme précédemment.

Widget::Widget()                            // Comme précédemment.
: pImpl(std::make_unique<Impl>())
{}

Widget::~Widget() = default;                // Comme précédemment.

Widget::Widget(Widget&& rhs) = default;             // Définitions.
Widget& Widget::operator=(Widget&& rhs) = default;
```

L'idiome Pimpl permet de diminuer les dépendances de compilation entre l'implémentation d'une classe et les clients de cette classe mais, conceptuellement, son utilisation ne change en rien ce qu'elle représente. La classe `Widget` d'origine contenait des données membres `std::string`, `std::vector` et `Gadget`, et, en supposant que, à l'instar des `std::string` et des `std::vector`, les `Gadget` peuvent être copiés, il serait sensé que `Widget` prenne en charge les opérations de copie. Nous devons écrire ces fonctions nous-mêmes car (1) le compilateur ne génère pas les opérations de copie pour les classes qui comprennent de types réservés aux déplacements, comme `std::unique_ptr`, et (2), même s'il le faisait, les fonctions générées copieraient uniquement le `std::unique_ptr` (effectueraient une *copie superficielle*), alors que nous voulons copier l'élément ciblé par le pointeur (réaliser une *copie profonde*).

Selon le rituel désormais familier, nous déclarons les fonctions dans le fichier d'en-tête et les mettons en œuvre dans le fichier d'implémentation :

```
class Widget {                  // Toujours dans "widget.h".
public:
  …                             // Autres fonctions, comme précédemment.


  Widget(const Widget& rhs);              // Déclarations
  Widget& operator=(const Widget& rhs);   // uniquement.

private:                        // Comme précédemment.
  struct Impl;
  std::unique_ptr<Impl> pImpl;
};


#include "widget.h"             // Comme précédemment, dans "widget.cpp".
…
```

Copyright © 2016 Dunod.

```
struct Widget::Impl { … };     // Comme précédemment.

Widget::~Widget() = default;   // Autres fonctions, comme précédemment.

Widget::Widget(const Widget& rhs)            // Constructeur de copie.
: pImpl(std::make_unique<Impl>(*rhs.pImpl))
{}

Widget& Widget::operator=(const Widget& rhs) // Opérateur
                                             // d'affectation par
                                             // copie.
{
  *pImpl = *rhs.pImpl;
  return *this;
}
```

L'implémentation des deux fonctions reste classique. Dans chacune, nous copions simplement les champs de la structure `Impl` depuis l'objet source (`rhs`) vers l'objet destination (`*this`). Au lieu de copier des champs un par un, nous profitons du fait que le compilateur va créer des opérations de copie pour `Impl` et que ces opérations copieront automatiquement chaque champ. Nous réalisons donc les opérations de copie de `Widget` en appelant les opérations de copie générées par le compilateur pour `Widget::Impl`. Dans le constructeur de copie, nous suivons le conseil 21, en choisissant d'utiliser `std::make_unique` plutôt que d'appeler directement `new`.

Pour la mise en œuvre de l'idiome Pimpl, nous conseillons le pointeur intelligent `std::unique_ptr` car le pointeur `pImpl` dans un objet (par exemple dans un `Widget`) est le propriétaire exclusif de l'objet d'implémentation correspondant (par exemple l'objet `Widget::Impl`). Il est toutefois intéressant de noter que si, à la place de `std::unique_ptr`, nous choisissions `std::shared_ptr` pour `pImpl`, le conseil ici donné ne s'appliquerait plus. Il n'y aurait plus besoin de déclarer un destructeur dans `Widget` et, sans ce destructeur déclaré par l'utilisateur, le compilateur générerait les opérations de déplacement appropriées. Autrement dit, en supposant le code suivant dans `widget.h` :

```
class Widget {                     // Dans "widget.h".
public:
  Widget();
  …                                // Aucune déclaration pour
                                   // le destructeur ou les opérations
                                   // de déplacement.

private:
  struct Impl;
  std::shared_ptr<Impl> pImpl;     // std::shared_ptr à la place
};                                 // de std::unique_ptr.
```

Copyright © 2016 Dunod.

© Dunod – Toute reproduction non autorisée est un délit.

et le code client suivant qui inclut `widget.h` :

```
Widget w1;

auto w2(std::move(w1));            // Construire w2 par déplacement.

w1 = std::move(w2);                // Affecter w1 par déplacement.
```

la compilation et l'exécution se passent comme nous le souhaitons : `w1` est construit par défaut, sa valeur est déplacée dans `w2`, elle est ensuite redéplacée dans `w1`, et, pour finir, `w1` et `w2` sont détruits (provoquant la destruction de l'objet `Widget::Impl` pointé).

L'origine de la différence de comportement liée à l'emploi de `std::unique_ptr` ou de `std::shared_ptr` pour `pImpl` vient de la prise en charge des supprimeurs personnalisés par chacun de ces pointeurs intelligents. Pour `std::unique_ptr`, le type du supprimeur fait partie du type du pointeur intelligent et le compilateur peut ainsi générer des structures de données plus petites et un code d'exécution plus rapide. En raison de cette efficacité accrue, les types pointés doivent être complets si des fonctions spéciales générées par le compilateur (par exemple des destructeurs ou des opérations de déplacement) sont employées. Pour `std::shared_ptr`, le type du supprimeur ne fait pas partie du type du pointeur intelligent. Cela impose des structures de données d'exécution plus volumineuses et du code un tantinet plus lent. En revanche, les types pointés n'ont pas besoin d'être complets pour utiliser les fonctions spéciales générées par le compilateur.

Dans le contexte de l'idiome Pimpl, les caractéristiques de `std::unique_ptr` et de `std::shared_ptr` n'exigent pas véritablement des concessions, car la relation entre des classes comme `Widget` et des classes comme `Widget::Impl` consiste en une propriété exclusive et `std::unique_ptr` est donc l'outil adapté. Néanmoins, il est bon de savoir que dans d'autres situations, lorsqu'une propriété partagée existe (et où `std::shared_ptr` est donc le choix de conception approprié), la définition des fonctions imposée par l'utilisation de `std::unique_ptr` n'est plus requise.

À retenir

- L'idiome Pimpl diminue les temps de construction en réduisant les dépendances de compilation entre les clients des classes et les implémentations de ces classes.
- Pour les pointeurs `pImpl` de type `std::unique_ptr`, il faut déclarer les fonctions membres spéciales dans l'en-tête de la classe, mais les mettre en œuvre dans le fichier d'implémentation. Il faut procéder ainsi même lorsque les implémentations par défaut des fonctions conviennent.
- Le conseil précédent s'applique à `std::unique_ptr`, non à `std::shared_ptr`.

Copyright © 2016 Dunod.

5

Références rvalue, sémantique du déplacement et transmission parfaite

Lorsqu'on les découvre pour la première fois, la sémantique du déplacement et la transmission parfaite semblent plutôt simples :

- La **sémantique du déplacement** permet au compilateur de remplacer des opérations de copie coûteuses par des opérations de déplacement plus légères. Tout comme les constructeurs de copie et les opérateurs d'affectation par copie nous donnent un contrôle sur le sens attribué à la copie des objets, les constructeurs de déplacement et les opérateurs d'affectation par déplacement nous offrent un contrôle sur la sémantique du déplacement. Celle-ci permet également de créer des types réservés au déplacement, comme `std::unique_ptr`, `std::future` et `std::thread`.
- La **transmission parfaite** permet d'écrire des templates de fonctions qui prennent des arguments quelconques et qui les relaient à d'autres fonctions en faisant en sorte que celles-ci reçoivent exactement les mêmes arguments que ceux qui ont été passés aux fonctions de retransmission.

Les références rvalue sont le ciment qui lie ces deux fonctionnalités plutôt hétéroclites. Elles représentent le mécanisme sous-jacent du langage qui rend possibles la sémantique du déplacement et la transmission parfaite.

Plus nous acquérons d'expérience avec ces fonctionnalités, plus nous réalisons que notre impression initiale relève de la métaphore de l'iceberg. Le monde de la sémantique du déplacement, de la transmission parfaite et des références rvalue est plus subtil qu'il ne paraît. Par exemple, `std::move` n'effectue aucun déplacement et la transmission parfaite est imparfaite. Les opérations de déplacement ne sont pas

Copyright © 2016 Dunod.
© Dunod – Toute reproduction non autorisée est un délit.

toujours moins coûteuses que la copie. Lorsque c'est le cas, leur prix est parfois plus élevé qu'attendu. Et elles ne sont pas toujours demandées dans un contexte où le déplacement est valide. La construction « *type*`&&` » ne représente pas toujours une référence rvalue.

Plus nous avançons dans le monde de ces fonctionnalités, plus il semble rester des choses à découvrir. Heureusement, ce monde a des limites et ce chapitre va nous y conduire. Une fois ce point atteint, cette partie de C++11 prendra tout son sens. Par exemple, les conventions d'utilisation de `std::move` et de `std::forward` seront connues. La nature ambiguë de « *type*`&&` » ne sera plus perturbante. Les différents comportements des opérations de déplacement seront compris. Toutes ces pièces vont se mettre en place. Nous reviendrons alors à notre point de départ, là où la sémantique de déplacement, la transmission parfaite et les références rvalue semblaient plutôt simples. Mais cette fois-ci, elles le seront réellement.

Dans les conseils de ce chapitre, il est particulièrement important de garder à l'esprit qu'un paramètre est toujours une lvalue, même si son type est une référence rvalue. Prenons l'instruction suivante :

```
void f(Widget&& w);
```

Le paramètre `w` est une lvalue, même si son type est une référence rvalue sur un `Widget`. (Si cela vous surprend, retournez à la présentation des lvalues et des rvalues qui débute à la section « Terminologie et conventions », page 2.)

CONSEIL N° 23. COMPRENDRE `STD::MOVE` ET `STD::FORWARD`

Pour aborder `std::move` et `std::forward`, il est plus commode d'expliquer ce que ces fonctions ne font pas. `std::move` ne déplace rien, et `std::forward` ne transmet rien. Au moment de l'exécution, elles ne font absolument rien. Elles ne génèrent aucun code exécutable. Pas un seul octet.

`std::move` et `std::forward` sont simplement des fonctions (en réalité des templates de fonctions) de conversion de type. `std::move` convertit systématiquement son argument en une rvalue, tandis que `std::forward` effectue cette conversion uniquement si une condition précise est remplie. Voilà tout. Cette explication soulève de nouvelles questions mais, fondamentalement, l'histoire s'arrête là.

Pour que les choses soient plus concrètes, voici un exemple d'implémentation de `std::move` en C++11. Elle n'est pas totalement conforme aux détails de la norme, mais elle en est très proche.

```
template<typename T>                          // Dans l'espace de noms std.
typename remove_reference<T>::type&&
move(T&& param)
{
  using ReturnType =                          // Déclaration d'alias ;
    typename remove_reference<T>::type&&;     // voir le conseil 9.
```

Copyright © 2016 Dunod.

```
    return static_cast<ReturnType>(param);
}
```

Deux parties du code sont soulignées. La première est le nom de la fonction, car la spécification du type de retour est plutôt complexe et nous ne voulons pas que vous vous y perdiez. La seconde est la conversion de type qui forme l'essence de la fonction. Vous le constatez, `std::move` prend une référence sur un objet (plus précisément une référence universelle ; voir le conseil 24) et renvoie une référence sur le même objet.

La présence de « `&&` » dans le type de retour de la fonction implique que `std::move` renvoie une référence rvalue, mais, comme l'explique le conseil 28, si `T` est une référence lvalue, `T&&` devient une référence lvalue. Pour éviter que cela ne se produise, le trait de type (voir le conseil 9) `std::remove_reference` est appliqué à `T`, garantissant que « `&&` » concerne un type qui n'est pas une référence. De cette manière, `std::move` retourne réellement une référence rvalue. Ce point est essentiel car les références rvalue renvoyées par des fonctions sont des rvalues. En résumé, `std::move` convertit son argument en une rvalue, et c'est tout.

En C++14, grâce à la déduction du type de retour d'une fonction (voir le conseil 3) et à l'alias de template `std::remove_reference_t` de la bibliothèque standard (voir le conseil 9), nous pouvons implémenter `std::move` de façon plus concise :

```
template<typename T>                          // C++14 ; toujours dans
decltype(auto) move(T&& param)                // l'espace de noms std.
{
  using ReturnType = remove_reference_t<T>&&;
  return static_cast<ReturnType>(param);
}
```

N'est-ce pas plus agréable à l'œil ?

Puisque `std::move` se contente de convertir son argument en une rvalue, il a été suggéré de changer son nom en quelque chose comme `rvalue_cast`. Mais peu importe, son nom est `std::move` et l'important est de ne pas oublier ce que cette fonction fait et ne fait pas. Elle réalise une conversion de type. Elle n'effectue pas un déplacement.

Les rvalues sont évidemment candidates au déplacement, et l'application de `std::move` à un objet indique au compilateur que celui-ci peut être déplacé. Voilà pourquoi `std::move` a pris le nom de l'opération qu'elle réalise : faciliter la désignation des objets qui peuvent être déplacés.

En vérité, les rvalues sont seulement *en général* des candidates au déplacement. Supposons que nous écrivions une classe qui représente des annotations. Son constructeur prend un paramètre `std::string` qui correspond à l'annotation et copie son contenu dans une donnée membre. Munis des informations données au conseil 41, nous déclarons un paramètre par valeur :

Copyright © 2016 Dunod.
© Dunod – Toute reproduction non autorisée est un délit.

```
class Annotation {
public:
  explicit Annotation(std::string text);   // Paramètre à copier,
  …                                        // donc passé par valeur
};                                         // selon le conseil 41.
```

Mais le constructeur de `Annotation` doit simplement lire la valeur de `text`, sans avoir besoin de la modifier. Conformément à la tradition immémoriale qui veut que `const` soit appliqué dès que c'est possible, nous modifions la déclaration et rendons `text const` :

```
class Annotation {
public:
  explicit Annotation(const std::string text)
  …
};
```

Pour éviter de payer le prix d'une opération de copie lors de la mémorisation de `text` dans la donnée membre, nous suivons le conseil 41 en appliquant `std::move` à `text` et produisons ainsi une rvalue :

```
class Annotation {
public:
  explicit Annotation(const std::string text)
  : value(std::move(text))    //"Déplacer" text dans value ; ce code
  { … }                       // n'a pas le comportement supposé !
  …

private:
  std::string value;
};
```

La compilation, l'édition de liens et l'exécution de ce code ne génèrent pas d'erreur. Le contenu de `text` est affecté à la donnée membre `value`. Le seul point qui ne soit pas parfaitement conforme à notre vision du fonctionnement de ce code est que le contenu de `text` n'est pas déplacé dans `value`, il y est *copié*. Certes, `text` est converti en rvalue par `std::move`, mais `text` est déclaré `const std::string`. Par conséquent, avant la conversion, `text` est une lvalue `const std::string` et la conversion produit une rvalue `const std::string`, mais son caractère `const` est préservé.

Examinons les conséquences éventuelles lorsque le compilateur doit déterminer le constructeur de `std::string` à appeler. Il existe deux possibilités :

```
class string {            // std::string est en réalité un
public:                   // typedef pour std::basic_string.
  …
  string(const string& rhs);    // Constructeur de copie.
  string(string&& rhs);         // Constructeur de déplacement.
  …
};
```

Copyright © 2016 Dunod.

Dans la liste d'initialisation du constructeur de `Annotation`, le résultat de `std::move(text)` est une rvalue de type `const std::string`. Cette rvalue ne peut pas être passée au constructeur de déplacement de `std::string`, car celui-ci prend une référence rvalue sur un `std::string` non `const`. En revanche, la rvalue peut être passée au constructeur de copie, car une référence lvalue sur un `const` peut être liée à une rvalue `const`. L'initialisation de membre peut donc invoquer le constructeur de *copie* dans `std::string`, même si `text` a été converti en rvalue ! Un tel comportement est essentiel pour la conservation des caractères `const`. Puisque le déplacement d'une valeur en dehors d'un objet modifie généralement cet objet, le langage ne doit pas autoriser le passage d'objets `const` à des fonctions (par exemple les constructeurs de déplacement) qui peuvent les modifier.

Nous pouvons tirer deux enseignements de cet exemple. Premièrement, les objets ne doivent pas être déclarés `const` s'ils doivent être impliqués dans des déplacements. Les demandes de déplacement sur des objets `const` sont transformées en opérations de copie. Deuxièmement, non seulement `std::move` ne déplace rien du tout, mais elle ne garantit pas que l'objet cible de la conversion soit éligible au déplacement. La seule chose certaine est que le résultat de l'application de `std::move` à un objet donne une rvalue.

Pour `std::forward`, le scénario est comparable à celui de `std::move`, mais, là où `std::move` convertit *sans condition* son argument en rvalue, `std::forward` y procède sous certaines conditions. `std::forward` est une conversion de type *conditionnelle*. Pour comprendre les conditions de la conversion, rappelons l'usage type de `std::forward`. Le cas le plus fréquent est un template de fonction avec en paramètre une référence universelle passée à une autre fonction :

```
void process(const Widget& lvalArg);      // Traiter les lvalues.
void process(Widget&& rvalArg);           // Traiter les rvalues.

template<typename T>                      // Template qui passe
void logAndProcess(T&& param)             // param à process.
{
  auto now =                              // Obtenir l'heure.
    std::chrono::system_clock::now();

  makeLogEntry("Appel de 'process' à", now);
  process(std::forward<T>(param));
}
```

Examinons deux appels à `logAndProcess`, l'un avec une lvalue, l'autre avec une rvalue :

```
Widget w;

logAndProcess(w);                 // Appel avec une lvalue.
logAndProcess(std::move(w));      // Appel avec une rvalue.
```

`logAndProcess` transmet le paramètre `param` à la fonction `process`. Celle-ci est surchargée pour les lvalues et les rvalues. Lorsque nous appelons `logAndProcess` avec

© Dunod – Toute reproduction non autorisée est un délit.
Copyright © 2016 Dunod.

une lvalue, nous nous attendons naturellement à ce que cette lvalue soit retransmise à `process` comme une lvalue, et que, lorsque `logAndProcess` est appelée avec une rvalue, la surcharge de `process` pour les rvalue soit invoquée.

Mais `param`, comme n'importe quel paramètre de fonction, est une lvalue. Chaque appel à `process` dans `logAndProcess` voudra donc invoquer la surcharge de `process` pour les lvalues. Afin d'éviter cela, nous avons besoin d'un mécanisme qui puisse convertir `param` en rvalue si et seulement si l'argument avec lequel `param` a été initialisé – l'argument passé à `logAndProcess` – était une rvalue. C'est précisément la raison d'être de `std::forward`. Voilà pourquoi `std::forward` est une conversion *conditionnelle* : la conversion en rvalue est effectuée uniquement si son argument a été initialisé avec une rvalue.

Vous vous demandez sans doute comment `std::forward` peut savoir que son argument a été initialisé avec une rvalue. Par exemple, dans le code précédent, comment `std::forward` peut-elle savoir si `param` a été initialisé à partir d'une lvalue ou d'une rvalue ? Voici la réponse courte : cette information est codée dans le paramètre de template `T` de `logAndProcess`. Ce paramètre est passé à `std::forward`, qui récupère l'information codée. Tous les détails de ce fonctionnement se trouvent dans le conseil 28.

Puisque `std::move` et `std::forward` se résument à des conversions de type, leur seule différence étant que `std::move` l'effectue systématiquement, tandis que `std::forward` l'effectue parfois, vous pourriez vous demander s'il ne serait pas possible de se passer de `std::move` et de se contenter de `std::forward`. D'un point de vue purement technique, la réponse est oui : `std::forward` peut être employée dans tous les cas. `std::move` n'est pas nécessaire. Bien entendu, aucune des fonctions n'est réellement *nécessaire* puisque nous pouvons écrire les conversions nous-mêmes, mais nous espérons que vous serez d'accord pour considérer cette approche plutôt maladroite.

L'intérêt de `std::move` réside dans son côté pratique, des potentialités d'erreur réduites et une meilleure clarté du code. Prenons une classe dans laquelle nous voulons suivre le nombre d'invocations du constructeur de déplacement. Nous avons simplement besoin d'un compteur `static` incrémenté lors de la construction par déplacement. En supposant que la seule donnée non statique dans la classe soit un `std::string`, voici la façon classique (c'est-à-dire en utilisant `std::move`) d'implémenter le constructeur de déplacement :

```
class Widget {
public:
  Widget(Widget&& rhs)
  : s(std::move(rhs.s))
  { ++moveCtorCalls; }

  ...

private:
```

Copyright © 2016 Dunod.

```
  static std::size_t moveCtorCalls;
  std::string s;
};
```

Voici la mise en œuvre du même comportement avec `std::forward` :

```
class Widget {
public:
  Widget(Widget&& rhs)                          // Implémentation,
  : s(std::forward<std::string>(rhs.s))         // non conventionnelle
  { ++moveCtorCalls; }                          // non souhaitable.

  ...

};
```

Notons tout d'abord que `std::move` n'exige qu'un seul argument de fonction (`rhs.s`), alors que `std::forward` a besoin d'un argument de fonction (`rhs.s`) et d'un argument de type de template (`std::string`). Remarquons ensuite que le type passé à `std::forward` ne doit pas être une référence, car il s'agit de la convention pour indiquer que l'argument passé est une rvalue (voir le conseil 28). En résumé, cela signifie que `std::move` demande moins de saisie que `std::forward` et que nous n'avons pas besoin de passer un argument de type qui encode le fait que l'argument passé est une rvalue. Cela élimine également la possibilité de passer un type incorrect (par exemple `std::string&`, qui conduirait à une construction non pas par déplacement mais par copie de la donnée membre `s`).

Plus important encore, l'utilisation de `std::move` signale une conversion inconditionnelle en rvalue, tandis que celle de `std::forward` indique une conversion en rvalue uniquement pour des références auxquelles des rvalues ont été liées. Ces deux actions sont très différentes. La première configure généralement un déplacement, tandis que la seconde passe – *transmet* – simplement un objet à une autre fonction en conservant son caractère initial de lvalue ou de rvalue. Puisque ces actions sont si différentes, il est préférable d'avoir deux fonctions séparées (et des noms de fonction différents) pour les distinguer.

À retenir

- `std::move` effectue une conversion inconditionnelle en rvalue. En soi, elle ne réalise aucun déplacement.
- `std::forward` convertit son argument en rvalue uniquement si celui-ci est lié à une rvalue.
- Ni `std::move` ni `std::forward` n'effectue une opération à l'exécution.

Copyright © 2016 Dunod.
© Dunod – Toute reproduction non autorisée est un délit.

CONSEIL N° 24. DISTINGUER LES RÉFÉRENCES UNIVERSELLES ET LES RÉFÉRENCES RVALUE

Quelqu'un a dit « la vérité vous rendra libres » mais, lorsque les conditions s'y prêtent, un mensonge bien choisi peut également être libérateur. Ce conseil 24 est un tel mensonge. Mais, puisque nous parlons de logiciel, évitons le mot « mensonge » et préférons plutôt dire que ce conseil représente une « abstraction ».

Pour déclarer une référence rvalue sur un type `T`, nous écrivons `T&&`. Il semble donc pertinent de supposer que la présence de « `T&&` » dans du code source dévoile une référence rvalue. Hélas, ce n'est pas aussi simple :

```
void f(Widget&& param);              // Une référence rvalue.

Widget&& var1 = Widget();            // Une référence rvalue.

auto&& var2 = var1;                  // Pas une référence rvalue.

template<typename T>
void f(std::vector<T>&& param);      // Une référence rvalue.

template<typename T>
void f(T&& param);                   // Pas une référence rvalue.
```

En réalité, « `T&&` » a deux significations différentes. L'une correspond évidemment à une référence rvalue. De telles références affichent le comportement attendu : elles se lient uniquement à des rvalues et leur principale raison d'être est d'identifier des objets qui peuvent être impliqués dans des déplacements.

La seconde interprétation pour « `T&&` » est *soit* une référence rvalue, *soit* une référence lvalue. Dans le code source, de telles références ressemblent à des références rvalue (c'est-à-dire « `T&&` »), mais elles ont le comportement de références lvalue (c'est-à-dire « `T&` »). En raison de leur double nature, elles peuvent être liées aussi bien à des rvalues (comme des références rvalue) qu'à des lvalues (comme des références lvalue). Par ailleurs, elles peuvent se lier à des objets `const` ou non, à des objets `volatile` ou non, et même à des objets `const` et `volatile`. Elles peuvent se lier à, virtuellement, *n'importe quoi*. Ces références à la souplesse sans précédent méritent leur propre nom. Nous les appelons *références universelles*[1].

Les références universelles se rencontrent dans deux contextes. Le plus répandu est celui des paramètres d'un template de fonction, comme dans cet extrait de l'exemple de code précédent :

```
template<typename T>
void f(T&& param);              // param est une référence universelle.
```

1. Le conseil 25 explique qu'il faut pratiquement toujours appliquer `std::forward` aux références universelles et, au moment où cet ouvrage était mis sous presse, certains membres de la communauté C++ ont commencé à désigner les références universelles par *forwarding references*.

Copyright © 2016 Dunod.

Les déclarations `auto` représentent le second contexte, notamment celle-ci tirée du code précédent :

```
auto&& var2 = var1;            // var2 est une référence universelle.
```

Le point commun entre ces contextes réside dans l'existence de la *déduction de type*. Dans le template `f`, le type de `param` est déduit et, dans la déclaration de `var2`, le type de cette variable est déduit. Faisons une comparaison avec les exemples suivants (également tirés du code précédent), pour lesquels la déduction de type est absente. Lorsque nous rencontrons « `T&&` » sans déduction de type, nous sommes face à une référence rvalue :

```
void f(Widget&& param);        // Aucune déduction de type ;
                               // param est une référence rvalue.

Widget&& var1 = Widget();      // Aucune déduction de type ;
                               // var1 est une référence rvalue.
```

En tant que références, les références universelles doivent être initialisées. L'initialiseur d'une référence universelle détermine si elle représente une référence rvalue ou une référence lvalue. Si l'initialiseur est une rvalue (respectivement une lvalue), la référence universelle correspond à une référence rvalue (respectivement une référence lvalue). Lorsque les références universelles sont des paramètres de fonction, l'initialiseur est fourni à l'endroit de l'appel :

```
template<typename T>
void f(T&& param);     // param est une référence universelle.

Widget w;
f(w);                  // lvalue passée à f ; param est de type
                       // Widget& (une référence lvalue).

f(std::move(w));       // rvalue passée à f ; param est de type
                       // Widget&& (une référence rvalue).
```

Pour qu'une référence soit universelle, la déduction de type est nécessaire, mais elle n'est pas suffisante. La *forme* de la déclaration de la référence doit également être correcte et les contraintes sont plutôt strictes. Elle doit être précisément « `T&&` ». Reprenons l'exemple suivant extrait du code montré précédemment :

```
template<typename T>
void f(std::vector<T>&& param);  // param est une référence rvalue.
```

À l'invocation de `f`, le type `T` est déduit (sauf si l'appelant le spécifie explicitement ; nous ne nous occuperons pas de ce cas limite). Mais la déclaration du type de `param` est non pas de la forme « `T&&` » mais de la forme « `std::vector<T>&&` ». Puisque cela écarte la possibilité que `param` soit une référence universelle, `param` est donc une référence rvalue, ce que confirmera le compilateur si nous tentons de passer une lvalue à `f` :

Copyright © 2016 Dunod.
© Dunod – Toute reproduction non autorisée est un délit.

```
std::vector<int> v;
f(v);                          // Erreur ! Une lvalue ne peut pas être
                               // liée à une référence rvalue.
```

Même la simple présence du qualificateur `const` suffit à éliminer la candidature d'une référence au statut de référence universelle.

```
template<typename T>
void f(const T&& param);       // param est une référence rvalue.
```

Si, dans un template, un paramètre de fonction est de type « `T&&` », nous pourrions penser qu'il est possible d'y voir une référence universelle. Mais ce n'est pas le cas. En effet, le fait d'être un template ne garantit pas la mise en place de la déduction de type. Examinons la fonction membre `push_back` dans `std::vector` :

```
template<class T, class Allocator = allocator<T>>  // Extrait de C++.
class vector {
public:
  void push_back(T&& x);
  ...
};
```

Le paramètre de `push_back` a bien la forme appropriée pour une référence universelle, mais, dans ce cas, la déduction de type n'a pas lieu. En effet, `push_back` ne peut pas exister sans une instanciation spécifique de `vector` dont elle doit faire partie, et le type de cette instanciation détermine pleinement la déclaration de `push_back`. Par exemple, supposons la déclaration suivante :

```
std::vector<Widget> v;
```

Dans ce cas, le template `std::vector` est instancié de la manière suivante :

```
class vector<Widget, allocator<Widget>> {
public:
  void push_back(Widget&& x);              // Référence rvalue.
  ...
};
```

Nous voyons à présent clairement que `push_back` ne déclenche aucune déduction de type. Cette fonction `push_back` pour `vector<T>` (il y en a deux, la fonction étant surchargée) déclare toujours un paramètre de type référence rvalue sur `T`.

À l'opposé, la fonction membre `emplace_back` de `std::vector`, conceptuellement similaire, emploie la déduction de type :

```
template<class T, class Allocator = allocator<T>>  // Toujours extrait
class vector {                                     // de C++.
public:
```

Copyright © 2016 Dunod.

```
  template <class... Args>
  void emplace_back(Args&&... args);
  ...
};
```

Dans ce cas, le paramètre de type `Args` est indépendant du paramètre de type `T` de `vector`, et `Args` doit donc être déduit chaque fois que `emplace_back` est appelée. (D'accord, `Args` est en réalité non pas un paramètre de type mais un ensemble de paramètres, mais, dans le cadre de cette discussion, nous pouvons le considérer comme un paramètre de type.)

Le fait que le paramètre de type de `emplace_back` se nomme `Args`, encore qu'il soit toujours une référence universelle, étaye notre commentaire précédent sur la *forme* d'une référence universelle, qui doit être « `T&&` ». Rien ne nous oblige à utiliser le nom de `T`. Par exemple, le template suivant prend une référence universelle, car la forme est correcte (« *type*`&&` ») et le type de `param` sera déduit (sauf, une fois encore le cas limite, si l'appelant le spécifie explicitement) :

```
template<typename MyTemplateType>          // param est une référence
void someFunc(MyTemplateType&& param);     // universelle.
```

Nous avons indiqué précédemment que les variables `auto` peuvent également être des références universelles. De façon plus précise, les variables déclarées avec le type `auto&&` sont des références universelles, car la déduction de type a lieu et elles ont la forme appropriée (« `T&&` »). Les références universelles `auto` ne sont pas aussi répandues que celles utilisées pour les paramètres des templates de fonctions, mais nous les rencontrons de temps à autre en C++11. Elles sont beaucoup plus fréquentes en C++14, car les expressions lambda de C++14 peuvent déclarer des paramètres `auto&&`. Par exemple, pour écrire une expression lambda qui enregistre le temps passé dans une invocation de fonction quelconque, nous pouvons procéder de la manière suivante :

```
auto timeFuncInvocation =
  [](auto&& func, auto&&... params)              // C++14.
  {
    Lancer le chronomètre.
    std::forward<decltype(func)>(func)(          // Invoquer func
      std::forward<decltype(params)>(params)...  // avec params.
      );
    Arrêter le chronomètre et enregistrer le temps écoulé.
  };
```

Si votre réaction au code « `std::forward<decltype(`*bla bla bla*`)>` » dans l'expression lambda est « Punaise mais c'est quoi ce truc ! », il est probable que vous n'ayez pas encore lu le conseil 33. Ne vous en occupez pas. Le point important dans ce conseil concerne les paramètres `auto&&` déclarés par l'expression lambda. `func` est une référence universelle qui peut être liée à n'importe quel objet, lvalue ou rvalue invocable. `args` correspond à zéro ou plusieurs références universelles (c'est-à-dire un ensemble de paramètres de référence universelle) qui peuvent être liées à un

Copyright © 2016 Dunod.
© Dunod – Toute reproduction non autorisée est un délit.

nombre quelconque d'objets de type arbitraire. Grâce aux références universelles `auto`, nous avons une fonction `timeFuncInvocation` qui peut chronométrer l'exécution de quasiment n'importe quelle fonction. (Pour plus d'informations sur la différence entre « toutes les fonctions » et « quasiment n'importe quelle fonction », consultez le conseil 30.)

N'oubliez pas que l'intégralité de ce conseil, sur les bases des références universelles, est un mensonge, une « abstraction ». La vérité sous-jacente se nomme *réduction de référence* (*reference collapsing*) et sera dévoilée au conseil 28. Cependant, la vérité n'enlève rien à l'utilité de l'abstraction. En faisant la différence entre les références rvalue et les références universelles, nous pouvons avoir une lecture plus précise du code source (« Est-ce que ce `T&&` se lie à des rvalues uniquement ou à n'importe quoi ? ») et nous pouvons éviter les ambiguïtés lors des échanges avec nos collègues (« J'utilise une référence universelle ici, non une référence rvalue... »). Cela nous permet également de donner un sens aux conseils 25 et 26, qui se fondent sur cette distinction. Accueillez donc l'abstraction, et savourez-la. Tout comme les lois du mouvement de Newton (qui sont techniquement incorrectes) sont généralement aussi utiles et plus faciles à appliquer que la théorie de relativité générale d'Einstein (« la vérité »), la notion de référence universelle est normalement préférable à l'analyse des détails de la réduction de référence.

À retenir

- Si un paramètre de template de fonction est de type `T&&` pour un type `T` déduit, ou si un objet est déclaré avec `auto&&`, le paramètre ou l'objet est une référence universelle.
- Si la forme de la déclaration du type n'est pas exactement *`type`*`&&`, ou si la déduction de type n'a pas lieu, *`type`*`&&` correspond à une référence rvalue.
- Les références universelles correspondent à des références rvalue si elles sont initialisées avec des rvalues, et à des références lvalue si l'initialisation se fait avec des lvalues.

CONSEIL N° 25. UTILISER STD::MOVE SUR DES RÉFÉRENCES RVALUE, STD::FORWARD SUR DES RÉFÉRENCES UNIVERSELLES

Les références rvalue se lient uniquement aux objets compatibles avec le déplacement. Si un paramètre est une référence rvalue, nous *savons* que l'objet auquel il est lié peut être déplacé :

```
class Widget {
  Widget(Widget&& rhs);          // rhs fait assurément référence à un
  …                              // objet candidat au déplacement.
};
```

Copyright © 2016 Dunod.

Nous voudrions donc passer de tels objets à d'autres fonctions de sorte que celles-ci puissent exploiter le fait qu'ils sont des rvalues. Pour cela, nous convertissons en rvalues les paramètres liés à de tels objets. Comme l'explique le conseil 23, c'est le rôle de `std::move`, qui a été créée dans ce but :

```
class Widget {
public:
  Widget(Widget&& rhs)              // rhs est une référence rvalue.
  : name(std::move(rhs.name)),
    p(std::move(rhs.p))
    { … }
  …

private:
  std::string name;
  std::shared_ptr<SomeDataStructure> p;
};
```

Les références universelles *peuvent*, quant à elles (voir le conseil 24), être liées à des objets susceptibles d'être déplacés. Elles doivent être converties en rvalues uniquement si elles ont été initialisées avec des rvalues. Le conseil 23 explique que c'est précisément le rôle de `std::forward` :

```
class Widget {
public:
  template<typename T>
  void setName(T&& newName)                 // newName est une
  { name = std::forward<T>(newName); }      // référence universelle.

  …
};
```

En résumé, les références rvalue doivent être *systématiquement converties* en rvalues (avec `std::move`) lorsqu'elles sont transmises à d'autres fonctions, car elles sont *toujours* liées à des rvalues, tandis que les références universelles doivent être *converties de façon conditionnelle* en rvalues (avec `std::forward`) lorsqu'elles sont relayées, car elles ne sont que *parfois* liées à des rvalues.

Le conseil 23 explique que l'utilisation de `std::forward` sur des références rvalue permet d'obtenir le comportement approprié, mais le code source est plus verbeux, sujet aux erreurs et littéral. Il est donc préférable d'éviter d'appliquer `std::forward` aux références rvalue. En revanche, il faut proscrire l'utilisation de `std::move` avec des références universelles, car cela peut conduire à une modification inattendue des lvalues (par exemple des variables locales) :

```
class Widget {
public:
  template<typename T>
  void setName(T&& newName)           // Référence universelle.
  { name = std::move(newName); }      // Le code compile, mais il
  …                                   // est très très mauvais !
```

© Dunod – Toute reproduction non autorisée est un délit.
Copyright © 2016 Dunod.

```
private:
  std::string name;
  std::shared_ptr<SomeDataStructure> p;
};

std::string getWidgetName();        // Fonction fabrique.

Widget w;

auto n = getWidgetName();           // n est une variable locale.

w.setName(n);                       // Déplace n dans w !

...                                 // La valeur de n est à présent inconnue.
```

Dans cet exemple, la variable locale `n` est passée à `w.setName`, que l'appelant est en droit de considérer comme une opération en lecture seule sur `n`. Mais, puisque `setName` utilise `std::move` en interne pour convertir systématiquement en rvalue la référence données en paramètre, la valeur de `n` est déplacée dans `w.name` et, suite à l'appel à `setName`, `n` revient avec une valeur indéfinie. De tels comportements peuvent conduire les développeurs au désespoir, voire à la violence.

Vous pourriez soutenir que `setName` n'aurait pas dû déclarer son paramètre en tant que référence universelle. Ces références ne peuvent pas être `const` (voir le conseil 24), mais il est certain que `setName` ne devrait pas modifier son paramètre. Vous pourriez souligner que si `setName` avait simplement été surchargée pour les lvalues `const` et pour les rvalues, le problème aurait pu être évité :

```
class Widget {
public:
  void setName(const std::string& newName)      // Affectation à partir
  { name = newName; }                           // d'une lvalue const.

  void setName(std::string&& newName)           // Affectation à partir
  { name = std::move(newName); }                // d'une rvalue.

  ...
};
```

Cette solution fonctionne, dans ce cas, mais elle a des inconvénients. Premièrement, le code est plus long à écrire et à maintenir (deux fonctions à la place d'un seul template). Deuxièmement, elle peut être moins efficace. Prenons, par exemple, cette utilisation de `setName` :

```
w.setName("Adela Novak");
```

Avec la version de `setName` qui prend une référence universelle, la chaîne de caractères littérale `"Adela Novak"` est passée à `setName`, où elle est transmise à l'opérateur d'affectation du `std::string` dans `w`. La donnée membre `name` de `w` est donc affectée directement à partir de la chaîne littérale, sans intervention d'objets

Copyright © 2016 Dunod.

`std::string` temporaires. En revanche, avec les versions surchargées de `setName`, un objet `std::string` temporaire est créé afin d'y lier le paramètre de `setName`. Ce `std::string` temporaire est ensuite déplacé dans la donnée membre de `w`. Un appel à `setName` entraîne donc l'exécution d'un constructeur de `std::string` (pour créer l'objet temporaire), d'un opérateur d'affectation par copie de `std::string` (pour déplacer `newName` dans `w.name`) et d'un destructeur de `std::string` (pour détruire l'objet temporaire). Cette suite d'exécutions est certainement plus onéreuse que la seule invocation de l'opérateur d'affectation de `std::string` avec un pointeur `const char*`. Le coût supplémentaire variera d'une implémentation à l'autre et son impact sera plus ou moins important selon les applications et les bibliothèques, mais il n'en reste pas moins que remplacer un template prenant une référence universelle par deux fonctions surchargées pour les références lvalue et rvalue augmentera le temps d'exécution dans certains cas. Si nous généralisons l'exemple en ayant une donnée membre de `Widget` de type arbitraire (au lieu de savoir qu'elle est un `std::string`), la baisse des performances peut croître énormément, car le déplacement de certains types est beaucoup plus coûteux que celui d'un `std::string` (voir le conseil 29).

Cependant, le problème le plus grave de la surcharge pour les lvalues et les rvalues n'est pas la quantité ou le caractère littéral du code source, pas plus que ses performances à l'exécution. Le souci le plus important vient du manque d'évolutivité de la conception. `Widget::setName` ne prenant qu'un seul paramètre, seules deux surcharges sont nécessaires. En revanche, pour les fonctions qui prennent plusieurs paramètres, chacun pouvant être une lvalue ou une rvalue, le nombre de surcharges croît de façon géométrique : n paramètres demandent 2^n surcharges. Mais il y a plus grave encore. Certaines fonctions, les fonctions templates pour être précis, prennent un nombre de paramètres *illimité*, chacun pouvant être une lvalue ou une rvalue. `std::make_shared` en est un parfait exemple, tout comme, depuis C++14, `std::make_unique` (voir le conseil 21). Examinons les déclarations de leurs surcharges les plus employées :

```
template<class T, class... Args>                    // Extrait de C++11.
shared_ptr<T> make_shared(Args&&... args);

template<class T, class... Args>                    // Extrait de C++14.
unique_ptr<T> make_unique(Args&&... args);
```

Avec de telles fonctions, il est impossible d'envisager des surcharges pour les lvalues et les rvalues : la seule solution consiste à utiliser des références universelles. Et, à l'intérieur de ces fonctions, nous pouvons vous l'assurer, `std::forward` est appliquée aux références universelles passées en paramètres lorsqu'elles sont transmises à d'autres fonctions. C'est exactement de cette manière que nous devons procéder.

En réalité, le plus souvent, à un moment donné, mais pas nécessairement dès le début. Dans certains cas, nous voudrons utiliser à plusieurs reprises dans une même fonction l'objet qui est lié à une référence rvalue ou à une référence universelle, et nous devrons nous assurer qu'il n'est pas déplacé tant que nous en avons besoin. Dans ce cas, nous appliquerons `std::move` (pour les références rvalue) ou `std::forward` (pour

Copyright © 2016 Dunod.
© Dunod – Toute reproduction non autorisée est un délit.

les références universelles) uniquement lors de la *dernière* utilisation de la référence. Par exemple :

```
template<typename T>                          // text est une référence
void setSignText(T&& text)                    // universelle.
{
  sign.setText(text);                         // Utiliser text, mais
                                              // sans le modifier.

  auto now =                                  // Obtenir l'heure.
    std::chrono::system_clock::now();
  signHistory.add(now,
                  std::forward<T>(text));     // Convertir sous condition
}                                             // text en rvalue.
```

Dans cet exemple, nous voulons être certains que la valeur de `text` n'est pas modifiée par `sign.setText`, car nous souhaitons employer cette valeur dans un appel à `signHistory.add`. `std::forward` est donc appliquée uniquement lors de la dernière utilisation de la référence universelle.

Pour `std::move`, nous pouvons tenir le même raisonnement (c'est-à-dire appliquer `std::move` à une référence rvalue lors de sa dernière utilisation), mais il est important de noter que, dans de rares cas, nous voudrons appeler `std::move_if_noexcept` à la place de `std::move`. Pour comprendre quand et pourquoi ces cas se présentent, consultez le conseil 14.

Si le retour de la fonction se fait *par valeur* et s'il concerne un objet lié à une référence rvalue ou une référence universelle, nous appliquerons `std::move` ou `std::forward` au moment du renvoi de la référence. Prenons l'exemple d'une fonction `operator+` qui effectue l'addition de deux matrices, celle de gauche étant une rvalue (sa zone de mémoire peut donc être réutilisée pour y placer la matrice résultante) :

```
Matrix                                        // Retour par valeur.
operator+(Matrix&& lhs, const Matrix& rhs)
{
  lhs += rhs;
  return std::move(lhs);                      // Déplacer lhs dans
}                                             // la valeur de retour.
```

En convertissant `lhs` en rvalue dans l'instruction `return` (avec `std::move`), `lhs` est déplacée à l'emplacement de la valeur de retour de la fonction. Supposons que nous omettions l'appel à `std::move` :

```
Matrix                                        // Comme précédemment.
operator+(Matrix&& lhs, const Matrix& rhs)
{
  lhs += rhs;
  return lhs;                                 // Copier lhs dans
}                                             // la valeur de retour.
```

Copyright © 2016 Dunod.

Dans ce cas, puisque `lhs` est une lvalue, le compilateur est obligé de la *copier* à l'emplacement de la valeur de retour. Si le type `Matrix` prend en charge la construction par déplacement, qui est plus efficace que la construction par copie, l'application de `std::move` dans l'instruction `return` permet d'obtenir un code plus performant.

Si `Matrix` ne prend pas en charge le déplacement, sa conversion en rvalue ne posera pas de problème, car la rvalue sera simplement copiée par le constructeur de copie de `Matrix` (voir le conseil 23). Si `Matrix` est ensuite modifiée pour prendre en charge le déplacement, `operator+` en bénéficiera automatiquement dès sa compilation. C'est pourquoi nous n'avons rien à perdre (et, dans certains cas, beaucoup à gagner) en appliquant `std::move` aux références rvalue qui sont renvoyées par des fonctions dont le retour se fait par valeur.

La situation est comparable pour les références universelles et `std::forward`. Examinons une fonction template `reduceAndCopy` qui prend un objet `Fraction` potentiellement non réduit, en fait la réduction et retourne une copie de la valeur réduite. Si l'objet initial est une rvalue, sa valeur doit être déplacée dans la valeur de retour (évitant ainsi le coût d'une copie), mais s'il s'agit d'une lvalue, une copie doit être effectuée :

```
template<typename T>
Fraction                          // Retour par valeur.
reduceAndCopy(T&& frac)           // Référence universelle en paramètre.
{
  frac.reduce();
  return std::forward<T>(frac);    // rvalue déplacée et lvalue copiée
}                                  // dans la valeur de retour.
```

Si l'appel à `std::forward` était absent, `frac` serait systématiquement copiée dans la valeur de retour de `reduceAndCopy`.

Certains programmeurs reprennent l'information précédente et tentent de l'étendre à des situations où elle ne s'applique pas. Voici leur raisonnement : « Si utiliser `std::move` sur une référence rvalue passée en paramètre et copiée dans la valeur de retour transforme la construction par copie en construction par déplacement, je peux obtenir la même optimisation pour les variables locales que je renvoie. » Autrement dit, ils pensent que si une fonction retourne une variable locale par valeur, comme dans l'exemple suivant :

```
Widget makeWidget()        // Version "par copie" de makeWidget.
{
  Widget w;                // Variable locale.

  …                        // Configurer w.

  return w;                // "Copier" w dans la valeur de retour.
}
```

ils peuvent l'« optimiser » en transformant la « copie » en déplacement :

Copyright © 2016 Dunod.
© Dunod – Toute reproduction non autorisée est un délit.

```
Widget makeWidget()        // Version par déplacement de makeWidget.
{
  Widget w;
  …
  return std::move(w);     // Déplacer w dans la valeur de retour
}                          // (à ne surtout pas faire !)
```

L'emploi des guillemets devrait vous indiquer que ce raisonnement présente quelques défauts. Mais qu'elle en est la raison ?

Il est erroné car le comité de normalisation a de l'avance sur ces programmeurs vis-à-vis de ce type d'optimisation. Il est depuis longtemps reconnu que la version « par copie » de `makeWidget` peut éviter la copie de la variable locale `w` en la construisant dans la zone de mémoire allouée à la valeur de retour de la fonction. Il s'agit de l'optimisation de la valeur de retour (RVO, *return value optimization*) et elle a été expressément adoptée par la norme C++ depuis qu'elle existe.

Exprimer ce bienfait par des mots demande un travail méticuleux, car nous voulons autoriser une telle *élision de copie* uniquement là où elle n'affectera pas le comportement observable du logiciel. En paraphrasant la prose légaliste (sans doute toxique) de la norme, cette approbation particulière stipule que le compilateur peut éluder la copie (ou le déplacement) d'un objet local[1] dans une fonction avec un retour par valeur si (1) le type de l'objet local est identique à celui retourné par la fonction et si (2) l'objet local est l'élément retourné. Avec ces éléments en tête, revenons à la version « par copie » de `makeWidget` :

```
Widget makeWidget()        // Version "par copie" de makeWidget.
{
  Widget w;
  …
  return w;                // "Copier" w dans la valeur de retour.
}
```

Dans ce cas, les deux conditions sont satisfaites et, vous pouvez nous croire sur parole, tout compilateur C++ sérieux mettra en œuvre la RVO de façon à éviter la copie de `w`. Autrement dit, la version « par copie » de `makeWidget` n'effectue aucune copie.

La version par déplacement de `makeWidget` a bien le comportement sous-entendu par son appellation (en supposant que `Widget` dispose d'un constructeur par déplacement) : elle déplace le contenu de `w` à l'emplacement de la valeur de retour de `makeWidget`. Mais pourquoi le compilateur n'utilise-t-il pas la RVO de façon à

1. Les objets locaux éligibles comprennent la plupart des variables locales (à l'instar de `w` dans `makeWidget`) ainsi que les objets temporaires créés pour les besoins de l'instruction `return`. Les paramètres d'une fonction ne sont pas concernés. Certaines personnes font une différence entre l'application de la RVO aux objets locaux nommés et non nommés (c'est-à-dire temporaires). Ils réservent le terme RVO aux objets non nommés et désignent son application aux objets nommés sous l'expression « optimisation de la valeur de retour nommée » (NRVO, *named return value optimization*).

Copyright © 2016 Dunod.

supprimer le déplacement, de nouveau en construisant `w` dans la zone de mémoire allouée à la valeur de retour de la fonction ? La réponse est simple : il ne le peut pas. La condition (2) stipule que la RVO peut être mise en œuvre uniquement si l'élément retourné est un objet local, ce qui n'est pas le cas dans la version par déplacement de `makeWidget`. Examinons de nouveau son instruction `return` :

```
return std::move(w);
```

L'élément retourné n'est pas l'objet local `w`, mais *une référence sur* `w` – le résultat de `std::move(w)`. Retourner une référence à un objet local ne remplit pas les conditions nécessaires à la RVO et le compilateur doit donc déplacer `w` dans la zone de mémoire réservée à la valeur de retour de la fonction. Les développeurs qui tentent d'aider leur compilateur dans ses optimisations en appliquant `std::move` à une variable locale retournée restreignent en réalité les possibilités d'optimisation !

Il ne faut pas oublier que la RVO est une optimisation. Le compilateur n'est pas obligé d'éluder les opérations de copie et de déplacement, même s'il en est capable. Mais peut-être sommes-nous paranoïaques et imaginons-nous que notre compilateur veut nous punir par des opérations de copie, uniquement parce qu'il en a la possibilité. Ou peut-être sommes-nous suffisamment informés pour reconnaître que, dans certains cas, la RVO est difficile à appliquer, par exemple lorsque différents chemins de contrôle dans une fonction renvoient des variables locales différentes. (Le compilateur doit générer le code de construction de la variable locale appropriée dans la mémoire allouée à la valeur de retour de la fonction, mais comment peut-il déterminer la variable locale appropriée ?) Dans ce cas, vous serez disposé à payer le prix d'un déplacement en contrepartie de la garantie de l'absence du coût d'une copie. Autrement dit, vous pensez qu'il est raisonnable d'appliquer `std::move` à un objet local retourné, simplement parce que vous savez que vous n'aurez jamais à payer pour une copie.

Dans ce cas, appliquer `std::move` à un objet local serait *encore* une mauvaise idée. La section de la norme qui approuve la RVO poursuit en expliquant que si les conditions de la RVO sont satisfaites, mais que le compilateur décide de ne pas éluder la copie, l'objet retourné *doit être traité comme une rvalue*. En effet, la norme exige que, si la RVO est permise, soit l'élision de copie a lieu, soit `std::move` est appliqué implicitement aux objets locaux retournés. Par conséquent, dans la version « par copie » de `makeWidget` :

```
Widget makeWidget()        // Comme précédemment.
{
  Widget w;
  …
  return w;
}
```

le compilateur doit éluder la copie de `w` ou doit considérer que la fonction est écrite de la manière suivante :

Copyright © 2016 Dunod.
© Dunod – Toute reproduction non autorisée est un délit.

```
Widget makeWidget()
{
  Widget w;
  …
  return std::move(w);       // Traiter w comme une rvalue, car
}                            // l'élision de copie est absente.
```

Le cas des paramètres de fonction passés par valeur est comparable. Ils ne sont pas éligibles à l'élision de copie en tant que valeur de retour de la fonction, mais le compilateur doit les traiter comme des rvalues s'ils sont renvoyés. Par conséquent, si notre code source ressemble à ceci :

```
Widget makeWidget(Widget w)       // Paramètre par valeur de même type
{                                 // que le retour de la fonction.
  …
  return w;
}
```

le compilateur doit considérer qu'il est écrit de la façon suivante :

```
Widget makeWidget(Widget w)
{
  …
  return std::move(w);            // Traiter w comme une rvalue.
}
```

Cela signifie que si nous utilisons `std::move` sur un objet local renvoyé par une fonction dont le retour se fait par valeur, nous ne pouvons pas aider le compilateur (s'il ne met pas en place l'élision de copie, il doit traiter l'objet local comme une rvalue), mais nous pouvons certainement le gêner (en empêchant la RVO). Il existe des cas où l'application de `std::move` à une variable locale peut se révéler raisonnable (c'est-à-dire lorsque nous la passons à une fonction et savons que nous ne l'utiliserons plus), mais pas dans celui d'une instruction `return` qui serait sinon éligible à la RVO ou qui retourne un paramètre par valeur.

À retenir

- Appliquer `std::move` aux références rvalue et `std::forward` aux références universelles au moment de leur dernière utilisation.
- Appliquer le même traitement aux références rvalue et aux références universelles qui sont renvoyées par des fonctions dont le retour se fait par valeur.
- Ne jamais appliquer `std::move` ou `std::forward` aux objets locaux qui seraient sinon éligibles à l'optimisation de la valeur de retour.

Copyright © 2016 Dunod.

CONSEIL N° 26. ÉVITER LA SURCHARGE SUR LES RÉFÉRENCES UNIVERSELLES

Supposons que nous ayons besoin d'une fonction qui prend un nom en paramètre, journalise la date et l'heure courantes, et ajoute le nom à une structure de données globale. Voici le code auquel nous pourrions arriver :

```
std::multiset<std::string> names;     // Structure de données globale.

void logAndAdd(const std::string& name)
{
  auto now =                          // Obtenir l'heure.
    std::chrono::system_clock::now();

  log(now, "logAndAdd");              // Créer l'entrée du journal.

  names.emplace(name);                // Ajouter name à la structure
}                                     // de données globale ; voir
                                      // le conseil 42 pour emplace.
```

Ce code ne semble pas déraisonnable, mais il n'est pas aussi efficace qu'il pourrait l'être. Examinons trois appels potentiels :

```
std::string petName("Darla");

logAndAdd(petName);                    // Passer un std::string
                                       // en lvalue.

logAndAdd(std::string("Persephone"));  // Passer un std::string
                                       // en rvalue.

logAndAdd("Patty Dog");                // Passer une chaîne littérale.
```

Dans le premier appel, le paramètre `name` de `logAndAdd` est lié à la variable `petName`. À la fin de la fonction `logAndAdd`, `name` est passé à `names.emplace`. Puisque `name` est une lvalue, il est copié dans `names`. Il est impossible d'éviter cette copie, car une lvalue (`petName`) a été transmise à `logAndAdd`.

Dans le deuxième appel, le paramètre `name` est lié à une rvalue (le `std::string` créé explicitement à partir de `"Persephone"`). Puisque `name` est lui-même une lvalue, il est copié dans `names`, mais nous reconnaissons que, en principe, sa valeur pourrait être déplacée dans `names`. Dans cet appel, nous payons une copie, mais nous pourrions faire en sorte d'obtenir uniquement un déplacement.

Dans le troisième appel, le paramètre `name` est à nouveau lié à une rvalue, mais il s'agit cette fois-ci d'un `std::string` temporaire créé implicitement à partir de `"Patty Dog"`. Comme dans le deuxième appel, `name` est copié dans `names`, mais, dans ce cas, l'argument initialement passé à `logAndAdd` est une chaîne littérale. Si cette chaîne était passée directement à `emplace`, il serait inutile de créer un `std::string` temporaire. À la place, `emplace` utiliserait la chaîne littérale pour créer l'objet `std::string` directement

Copyright © 2016 Dunod.
© Dunod – Toute reproduction non autorisée est un délit.

dans le `std::multiset`. Par conséquent, dans ce troisième appel, nous payons pour une copie d'un `std::string`, alors qu'il n'y a aucune véritable raison de payer ne serait-ce que pour un déplacement, et encore moins pour une copie.

Nous pouvons apporter une solution à l'inefficacité des deuxième et troisième appels, en modifiant `logAndAdd` afin qu'elle prenne en paramètre une référence universelle (voir le conseil 24) et, en accord avec le conseil 25, appeler `std::forward` sur cette référence en la passant à `emplace`. Le résultat parle de lui-même :

```
template<typename T>
void logAndAdd(T&& name)
{
  auto now = std::chrono::system_clock::now();
  log(now, "logAndAdd");
  names.emplace(std::forward<T>(name));
}

std::string petName("Darla");           // Comme précédemment.

logAndAdd(petName);                     // Comme précédemment, copier
                                        // une lvalue dans un multiset.

logAndAdd(std::string("Persephone"));   // Déplacer une rvalue au lieu
                                        // de la copier.

logAndAdd("Patty Dog");                 // Créer un std::string dans
                                        // un multiset au lieu de copier
                                        // un std::string temporaire.
```

Génial, efficacité optimale !

Si l'histoire s'arrêtait là, nous pourrions nous retirer fièrement. Cependant, nous avons omis de préciser que les clients n'ont pas toujours un accès direct aux noms nécessaires à `logAndAdd`. Certains ne disposent que d'un indice dont `logAndAdd` se sert pour rechercher le nom dans un tableau. Pour ces clients, nous surchargeons `logAndAdd` :

```
std::string nameFromIdx(int idx);       // Retourner le nom qui
                                        // correspond à idx.

void logAndAdd(int idx)                 // Nouvelle surcharge.
{
  auto now = std::chrono::system_clock::now();
  log(now, "logAndAdd");
  names.emplace(nameFromIdx(idx));
}
```

La résolution des appels aux deux surcharges se fait comme attendu :

```
std::string petName("Darla");           // Comme précédemment.
```

Copyright © 2016 Dunod.

```
logAndAdd(petName);                    // Comme précédemment, ces
logAndAdd(std::string("Persephone"));  // appels invoquent tous
logAndAdd("Patty Dog");                // la surcharge T&&.

logAndAdd(22);                         // Appeler la surcharge int.
```

En réalité, la résolution se passe correctement uniquement si nous n'en demandons pas trop. Supposons qu'un client utilise un `short` pour mémoriser l'indice et le passe à `logAndAdd` :

```
short nameIdx;
…                                      // Attribuer une valeur à nameIdx.

logAndAdd(nameIdx);                    // Erreur !
```

Le commentaire de la dernière ligne est un tantinet succinct. Expliquons ce qui se passe.

`logAndAdd` a deux surcharges. Celle qui prend en argument une référence universelle peut déduire que `T` est un `short` et constitue donc une correspondance exacte. La surcharge pour un paramètre `int` peut correspondre à l'argument `short` uniquement au travers d'une promotion. D'après les règles de résolution normale de la surcharge, une correspondance exacte a la priorité sur une correspondance par promotion. En conséquence, la surcharge pour les références universelles est invoquée.

Dans cette fonction surchargée, le paramètre `name` est lié au `short` passé en paramètre. `std::forward` est ensuite appliquée à `name` lors de l'invocation de la fonction membre `emplace` sur `names` (un `std::multiset<std::string>`). À son tour, elle le retransmet scrupuleusement au constructeur de `std::string`. Puisqu'il n'existe aucun constructeur de `std::string` qui prenne un `short`, l'appel au constructeur de `std::string` à l'intérieur de l'appel à `multiset::emplace` dans l'appel à `logAndAdd` échoue. Tout cela uniquement parce que la surcharge avec une référence universelle offre une meilleure correspondance pour un `short` que celle avec un `int`.

Les fonctions qui prennent en paramètres des références universelles sont les plus gourmandes de C++. Leur instanciation crée des correspondances exactes pour quasiment n'importe quel type d'argument. (Les quelques sortes d'arguments non concernés sont décrites au conseil 30.) C'est pourquoi il est préférable de ne jamais combiner surcharge et références universelles : une surcharge avec une référence universelle aspire beaucoup plus de types d'arguments que son développeur ne s'y attend généralement.

Une solution à ce comportement indésirable consiste à écrire un constructeur de transmission parfaite. Une petite modification de l'exemple de `logAndAdd` illustre le problème. Au lieu d'écrire une fonction qui prend soit un `std::string`, soit un indice utilisé pour rechercher un `std::string`, imaginons une classe `Person` dont les constructeurs remplissent le même objectif :

Copyright © 2016 Dunod.
© Dunod – Toute reproduction non autorisée est un délit.

```
class Person {
public:
  template<typename T>
  explicit Person(T&& n)           // Constructeur de transmission parfaite ;
  : name(std::forward<T>(n)) {}    // initialise la donnée membre.

  explicit Person(int idx)         // Constructeur avec un int.
  : name(nameFromIdx(idx)) {}
  …

private:
  std::string name;
};
```

Comme c'était le cas avec `logAndAdd`, le passage d'un type entier autre qu'un `int` (par exemple `std::size_t`, `short`, `long`, etc.) déclenchera l'appel de la surcharge du constructeur pour une référence universelle à la place de celle pour un `int`, et conduira à un échec de la compilation. Dans cet exemple, le problème est toutefois pire, car les surcharges dans `Person` sont plus nombreuses qu'on ne le voit. Le conseil 17 explique que, sous certaines conditions, C++ générera des constructeurs de copie et de déplacement, même si la classe comprend un template de constructeur dont l'instanciation permet d'obtenir la signature du constructeur de copie ou de déplacement. Si ces constructeurs sont générés pour `Person`, sa déclaration équivaut en réalité à la suivante :

```
class Person {
public:
  template<typename T>              // Constructeur de transmission parfaite.
  explicit Person(T&& n)
  : name(std::forward<T>(n)) {}

  explicit Person(int idx);         // Constructeur avec un int.

  Person(const Person& rhs);        // Constructeur de copie
                                    // (généré par le compilateur).

  Person(Person&& rhs);             // Constructeur de déplacement
  …                                 // (généré par le compilateur).

};
```

Ce comportement n'est évident que pour les programmeurs qui ont passé beaucoup de temps avec des compilateurs ou qui en ont développés, et qui ont oublié ce qu'était un être humain :

```
Person p("Nancy");

auto cloneOfP(p);            // Créer un nouveau Person à partir de p ;
                             // ce code ne compile pas !
```

Copyright © 2016 Dunod.

Dans cet exemple, nous essayons de créer un objet `Person` à partir d'un autre `Person`, ce qui semble évidemment un cas de construction par copie. (`p` est une lvalue et nous pouvons donc oublier toute idée de « copie » accomplie par une opération de déplacement.) Mais ce code n'invoque pas le constructeur de copie. Il va appeler le constructeur de transmission parfaite, qui va tenter d'initialiser la donnée membre `std::string` de `Person` avec un objet `Person` (`p`). Puisque `std::string` n'a pas de constructeur qui prend un `Person`, le compilateur abandonne et nous gratifie de messages d'erreur éventuellement longs et incompréhensibles.

Vous vous demandez sans doute pourquoi le constructeur de transmission parfaite est appelé à la place du constructeur de copie, alors que l'initialisation du `Person` se fait à partir d'un autre `Person`. Si l'initialisation est bien celle-là, le compilateur se doit nénamoins de respecter les règles de C++ et, dans ce cas, il s'agit de celles de la résolution des appels aux fonctions surchargées.

Voici le raisonnement suivi par le compilateur. Puisque `cloneOfP` est initialisé avec une lvalue non `const` (`p`), le template de constructeur peut être instancié de façon à prendre une lvalue non `const` de type `Person`. Suite à cette instanciation, la classe `Person` ressemble à la suivante :

```
class Person {
public:
  explicit Person(Person& n)              // Instancié à partir du
  : name(std::forward<Person&>(n)) {}     // template de transmission
                                          // parfaite.

  explicit Person(int idx);               // Comme précédemment.

  Person(const Person& rhs);              // Constructeur de copie
  ...                                     // (généré par le compilateur).

};
```

Dans l'instruction

```
auto cloneOfP(p);
```

`p` peut être passé au constructeur de copie ou au template instancié. L'appel du constructeur de copie nécessiterait l'ajout de `const` à `p` afin de correspondre au type du paramètre de ce constructeur, mais l'appel au template instancié ne requiert aucun ajout. La surcharge générée à partir du template donne donc une meilleure correspondance et le compilateur fait ce pourquoi il a été conçu : générer un appel à la fonction qui présente la meilleure correspondance. Par conséquent, la « copie » de lvalue non `const` de type `Person` est prise en charge non pas par le constructeur de copie mais par le constructeur de transmission parfaite.

Si nous modifions légèrement l'exemple de sorte que l'objet à copier soit `const`, nous entendons un autre son de cloche :

Copyright © 2016 Dunod.
© Dunod – Toute reproduction non autorisée est un délit.

```
const Person cp("Nancy");     // L'objet est à présent const.

auto cloneOfP(cp);            // Appeler le constructeur de copie !
```

Puisque l'objet à copier est à présent `const`, nous obtenons une correspondance exacte avec le paramètre du constructeur de copie. Le template de constructeur peut également être instancié pour obtenir la même signature :

```
class Person {
public:
  explicit Person(const Person& n);      // Instancié à partir du
                                         // template.

  Person(const Person& rhs);             // Constructeur de copie
                                         // (généré par le compilateur).
  ...
};
```

Mais cette instanciation ne compte pas, car l'une des règles de la résolution de la surcharge en C++ stipule que si une instanciation de template et une fonction non template (c'est-à-dire une fonction normale) donnent toutes deux la même correspondance pour un appel de fonction, la fonction normale doit avoir la priorité. Le constructeur de copie (une fonction normale) surpasse donc le template instancié pour obtenir la même signature.

(Si vous vous demandez pourquoi un compilateur génère un constructeur de copie lorsqu'il a la possibilité d'obtenir la signature de celui-ci en instanciant un template de constructeur, consultez le conseil 17.)

L'interaction entre les constructeurs de transmission parfaite et les opérations de copie et de déplacement générées par le compilateur apporte des soucis supplémentaires lorsque l'héritage est de la partie. En particulier, les implémentations classiques des opérations de copie et de déplacement dans une classe dérivée affichent un comportement assez surprenant. Examinons la situation suivante :

```
class SpecialPerson: public Person {
public:
  SpecialPerson(const SpecialPerson& rhs)  // Constructeur de copie ;
  : Person(rhs)                            // appelle le constructeur de
  { ... }                                  // transmission de la classe
                                           // de base!

  SpecialPerson(SpecialPerson&& rhs)     // Constructeur de déplacement ;
  : Person(std::move(rhs))               // appelle le constructeur de
  { ... }                                // transmission de la classe
                                         // de base!
};
```

Les commentaires révèlent que les constructeurs de copie et de déplacement de la classe dérivée n'invoquent pas leur homologue de la classe de base, mais le constructeur de transmission parfaite de celle-ci ! Pour en comprendre la raison, remarquons que les

Copyright © 2016 Dunod.

fonctions de la classe dérivée utilisent des arguments de type `SpecialPerson` à passer à leur classe de base, puis subissent les conséquences de l'instanciation de template et de la résolution de la surcharge pour les constructeurs de la classe `Person`. Pour finir, le code ne compile pas car `std::string` n'offre pas de constructeur qui prend un `SpecialPerson`.

Vous devriez à présent être convaincu que la surcharge sur des paramètres qui sont des références universelles doit être évitée autant que possible. D'accord, mais comment pouvons-nous procéder si nous avons besoin d'une fonction qui transmet la plupart des types d'arguments et doit en traiter certains de façon particulière ? Puisque les réponses à cette question sont nombreuses, nous allons y consacrer le conseil 27.

À retenir

- La surcharge sur des références universelles conduit presque toujours à l'appel de cette surcharge dans des situations plus nombreuses qu'imaginé.
- Les constructeurs de transmission parfaite soulèvent des difficultés, car ils donnent généralement de meilleures correspondances que les constructeurs de copie pour des lvalues non `const`, et ils peuvent empêcher les appels aux constructeurs de copie et de déplacement d'une classe de base depuis une classe dérivée.

CONSEIL N° 27. SE FAMILIARISER AVEC LES ALTERNATIVES À LA SURCHARGE SUR LES RÉFÉRENCES UNIVERSELLES

Le conseil 26 explique que la surcharge sur les références universelles peut conduire à divers problèmes, tant au niveau des fonctions membres (en particulier les constructeurs) que des fonctions indépendantes. Toutefois, elle présente également des exemples où cette surcharge se révèle utile. Si seulement elle pouvait se comporter comme nous l'aimerions ! Ce conseil décrit comment arriver au comportement souhaité, que ce soit par une conception qui évite les surcharges sur les références universelles ou par une utilisation qui contraint les types des arguments auxquels elles peuvent correspondre.

Les explications se fondent sur les exemples donnés au conseil 26. Si cela vous semble nécessaire, n'hésitez pas à le (re)lire avant de poursuivre.

Renoncer à la surcharge

Le premier exemple du conseil 26, `logAndAdd`, illustre parfaitement les nombreuses fonctions qui permettent d'éviter les inconvénients de la surcharge sur les références universelles en donnant simplement des noms différents à ce qui serait des surcharges. Ainsi, les deux surcharges de `logAndAdd` pourraient être renommées en `logAndAddName` et `logAndAddNameIdx`. Malheureusement, cette solution ne convient pas dans le

Copyright © 2016 Dunod.
© Dunod – Toute reproduction non autorisée est un délit.

deuxième exemple étudié, le constructeur de `Person`, car les noms des constructeurs sont fixés par le langage. De toute manière, qui voudrait abandonner la surcharge ?

Passer par `const T&`

Une autre approche consiste à revenir à du C++98 et à remplacer le passage par une référence universelle par un passage par une référence lvalue sur un `const`. Il s'agit en réalité de la première alternative envisagée dans le conseil 26. Son inconvénient réside dans une conception qui manque d'efficacité. Conscients des problèmes liés à l'interaction entre les références universelles et la surcharge, nous pourrions préférer garder les choses simples et perdre un peu en efficacité.

Passer par valeur

Une méthode qui permet souvent de composer avec les performances sans augmenter la complexité consiste, contre toute attente, à remplacer le passage par référence des paramètres par un passage par valeur. Elle suit le conseil 41, où nous préconisons de passer des objets par valeur lorsque nous savons qu'ils seront copiés. Puisque tous les détails de ce fonctionnement et de l'efficacité obtenue se trouvent dans le conseil 41, nous nous contenterons ici de montrer son utilisation avec `Person` :

```
class Person {
public:
  explicit Person(std::string n)  // Remplace le constructeur T&& ;
  : name(std::move(n)) {}         // voir le conseil 41 pour l'emploi
                                  // de std::move.

  explicit Person(int idx)        // Comme précédemment.
  : name(nameFromIdx(idx)) {}
  …

private:
  std::string name;
};
```

Puisque `std::string` n'offre aucun constructeur qui prend uniquement un entier, tous les arguments `int` et équivalents à `int` destinés à un constructeur de `Person` (par exemple `std::size_t`, `short`, `long`) arrivent à la surcharge sur `int`. De manière comparable, tous les arguments de type `std::string` (et tout ce qui permet de créer un `std::string`, par exemple des littéraux comme `"Ruth"`) sont transmis au constructeur qui attend un `std::string`. Aucune surprise pour le code appelant. Vous pourriez souligner que certaines personnes pourraient être surprises que la représentation d'un pointeur nul par `0` ou `NULL` invoque la surcharge sur `int`, mais, dans ce cas, nous leur conseillons de lire attentivement le conseil 8 jusqu'à ce qu'elles oublient l'idée d'utiliser `0` ou `NULL` comme pointeur nul.

Copyright © 2016 Dunod.

Utiliser le *tag dispatching*

Avec le passage par référence lvalue sur `const` ou le passage par valeur, la transmission parfaite n'est pas prise en charge. Si la raison d'utiliser une référence universelle vient de la transmission parfaite, nous n'avons pas le choix. Pourtant, nous ne souhaitons pas abandonner la surcharge. Dans ce cas, si nous ne voulons pas renoncer à la surcharge ni aux références universelles, comment pouvons-nous éviter la surcharge sur les références universelles ?

En réalité, ce n'est pas difficile. La résolution des appels aux fonctions surchargées passe par l'analyse de tous les paramètres de toutes les surcharges et de tous les arguments dans les appels, puis par le choix de la fonction qui donne la meilleure correspondance globale (en prenant en compte toutes les combinaisons de paramètres et d'arguments). En général, un paramètre de type référence universelle permet d'obtenir une correspondance exacte quel que soit l'argument passé, mais, si la référence universelle fait partie d'une liste de paramètres qui ne sont pas tous des références universelles, il suffit d'une correspondance moins bonne sur ces paramètres pour qu'une surcharge ne soit plus retenue. Voilà les bases du *tag dispatching*. Un exemple facilitera la compréhension de la description qui va suivre.

Nous allons appliquer le *tag dispatching* à l'exemple `logAndAdd` donné au conseil 26. Voici le code correspondant :

```
std::multiset<std::string> names;      // Structure de données globale.

template<typename T>                   // Créer l'entrée du journal et
void logAndAdd(T&& name)               // ajouter name à la structure
{                                      // de données.
  auto now = std::chrono::system_clock::now();
  log(now, "logAndAdd");
  names.emplace(std::forward<T>(name));
}
```

En soi, cette fonction affiche le comportement souhaité, mais, si nous introduisons la surcharge sur `int` afin de rechercher des objets par un indice, nous retombons dans les problèmes décrits au conseil 26. L'objectif de ce conseil est de les éviter. Au lieu d'ajouter la surcharge, nous allons revoir `logAndAdd` de façon à mettre en place une délégation vers deux autres fonctions, l'une pour les valeurs entières, l'autre pour tous les autres types. `logAndAdd` acceptera tous les types d'arguments, qu'il s'agisse d'entiers ou non.

Les deux fonctions qui réalisent le véritable travail se nommeront `logAndAddImpl`. Autrement dit, nous allons utiliser la surcharge. Puisque l'une des fonctions prendra une référence universelle, nous aurons la combinaison de la surcharge et des références universelles. Cependant, chaque fonction va également prendre un second paramètre, qui précisera si l'argument passé est un entier. C'est grâce à ce second paramètre que nous éviterons les difficultés décrites au conseil 26. En effet, nous ferons en sorte que le second paramètre soit le critère de sélection de la surcharge.

Fin du blablabla, voici le code de la nouvelle version presque correcte de `logAndAdd` :

Copyright © 2016 Dunod.
© Dunod – Toute reproduction non autorisée est un délit.

```
template<typename T>
void logAndAdd(T&& name)
{
  logAndAddImpl(std::forward<T>(name),
                std::is_integral<T>());   // Pas tout à fait correct.
}
```

Cette fonction transmet son paramètre à `logAndAddImpl`, mais elle lui passe également un argument indiquant si le type du paramètre (`T`) correspond à un entier. Tout au moins, c'est ce que nous supposons. Lorsque l'argument entier est une rvalue, le comportement est correct. Mais, comme l'explique le conseil 28, si un argument lvalue est passé à la référence universelle `name`, le type déduit pour `T` sera une référence lvalue. Autrement dit, si une lvalue de type `int` est passé à `logAndAdd`, le type déduit pour `T` sera `int&`. Il ne s'agit pas d'un type entier, car les références ne sont pas des entiers. Cela signifie que `std::is_integral<T>` retournera faux pour tout argument qui est une lvalue, même si celui-ci représente une valeur entière.

Identifier le problème équivaut à le résoudre, car la bibliothèque standard de C++ dispose d'un trait de type (voir le conseil 9), `std::remove_reference`, qui effectue l'opération suggérée par son nom et dont nous avons besoin : retirer tout qualificatif de référence à un type. Voici donc la bonne manière d'écrire `logAndAdd` :

```
template<typename T>
void logAndAdd(T&& name)
{
  logAndAddImpl(
    std::forward<T>(name),
    std::is_integral<typename std::remove_reference<T>::type>()
  );
}
```

L'affaire est réglée. (C++14 permet de gagner au niveau de la saisie en remplaçant le code en exergue par `std::remove_reference_t<T>` ; voir le conseil 9.)

Nous pouvons à présent nous focaliser sur la fonction invoquée, `logAndAddImpl`. Elle présente deux surcharges, la première correspondant uniquement aux types non entiers (c'est-à-dire les types pour lesquels `std::is_integral<typename std::remove_reference<T>::type>` est faux) :

```
template<typename T>                                  // Argument
void logAndAddImpl(T&& name, std::false_type)         // non entier :
{                                                     // l'ajouter à
  auto now = std::chrono::system_clock::now();        // la structure
  log(now, "logAndAdd");                              // de données
  names.emplace(std::forward<T>(name));               // globale.
}
```

Ce code est simple, dès lors que la mécanique derrière le paramètre en exergue est comprise. Conceptuellement, `logAndAdd` passe à `logAndAddImpl` un booléen qui indique si un type entier a été transmis à `logAndAdd`. Cependant, `true` et `false` sont des

Copyright © 2016 Dunod.

valeurs *à l'exécution*, alors que nous avons besoin de la résolution de la surcharge – un phénomène qui se produit *à la compilation* – pour choisir la surcharge de `logAndAddImpl` appropriée. Autrement dit, nous avons besoin d'un *type* qui correspond à `true` et d'un autre pour `false`. Ce besoin est suffisamment fréquent pour que la bibliothèque standard apporte le nécessaire sous les noms `std::true_type` et `std::false_type`. L'argument passé à `logAndAddImpl` par `logAndAdd` est un objet dont le type dérive de `std::true_type` lorsque `T` est un entier, et de `std::false_type` lorsque `T` n'est pas un entier. En résumé, cette surcharge de `logAndAddImpl` sera sélectionnée pour l'appel effectué dans `logAndAdd` uniquement si `T` n'est pas un entier.

La seconde surcharge s'occupe du cas opposé : lorsque `T` est un type entier. `logAndAddImpl` recherche alors le nom qui correspond à l'indice passé et le transmet simplement à `logAndAdd` :

```
std::string nameFromIdx(int idx);            // Comme dans le conseil 26.

void logAndAddImpl(int idx, std::true_type)  // Agument entier :
{                                            // rechercher le nom
  logAndAdd(nameFromIdx(idx));               // et le passer à
}                                            // logAndAdd.
```

Puisque la version de `logAndAddImpl` qui prend un indice recherche le nom correspondant et transmet celui-ci à `logAndAdd` (dans laquelle il est passé à l'autre surcharge de `logAndAddImpl` avec `std::forward`), nous évitons l'ajout du code de journalisation dans les deux variantes de `logAndAddImpl`.

Dans cette conception, les types `std::true_type` et `std::false_type` représentent des « étiquettes » (*tag*) dont le seul but est d'orienter la résolution de la surcharge dans la direction souhaitée. Vous remarquerez que nous n'avons même pas nommé ces paramètres. Ils n'ont aucun rôle à l'exécution et, en réalité, nous espérons que le compilateur déterminera que les paramètres étiquettes ne sont pas utilisés et qu'il les retirera de l'image exécutable du programme. (Certains compilateurs le font, tout au moins parfois.) L'appel aux fonctions surchargées dans `logAndAdd` « distribue » (*dispatch*) le travail à la surcharge appropriée grâce à la création de l'objet étiquette adéquat. Voilà l'origine du nom donné à cette conception : *tag dispatching*. Il s'agit d'un élément standard de la métaprogrammation par template et plus vous examinerez le code des bibliothèques C++ modernes, plus vous le rencontrerez.

Dans notre cas, l'important n'est pas tant le fonctionnement du *tag dispatching* mais sa capacité à nous laisser combiner les références universelles et la surcharge sans rencontrer les problèmes décrits au conseil 26. La fonction de distribution – `logAndAdd` – prend un paramètre non contraint de type référence universelle, mais elle n'est pas surchargée. Les fonctions d'implémentation – `logAndAddImpl` – sont surchargées et prennent un paramètre de type référence universelle. Cependant, la résolution des appels de ces fonctions ne dépend pas uniquement de ce paramètre mais également du paramètre étiquette, dont les valeurs permettent de déterminer une correspondance valide avec une seule surcharge. En résumé, la surcharge invoquée est sélectionnée par cette étiquette. Le fait que le paramètre de type référence universelle donnera toujours une correspondance exacte pour son argument ne compte pas.

Copyright © 2016 Dunod.
© Dunod – Toute reproduction non autorisée est un délit.

Contraindre des templates qui prennent des références universelles

Le *tag dispatching* se fonde sur l'existence d'une unique (non surchargée) fonction dans l'API cliente qui distribue le travail à effectuer à des fonctions d'implémentation. La création d'une fonction de distribution non surchargée se révèle en général facile, mais le second cas problématique du conseil 26, celui du constructeur de transmission parfaite pour la classe `Person`, fait exception. Puisque le compilateur peut décider lui-même de générer des constructeurs de copie et de déplacement, il ne nous suffit pas d'écrire un seul constructeur et d'y employer la distribution. En effet, certains appels aux constructeurs pourront se trouver dans des fonctions produites par le compilateur qui contournent le mécanisme de distribution.

En réalité, le véritable problème ne vient pas du fait que les fonctions générées par le compilateur contournent parfois le système de distribution par étiquette, mais qu'elle ne le contourne pas *à chaque fois*. Virtuellement, nous voulons toujours que le constructeur de copie d'une classe traite les demandes de copie des lvalues de ce type, mais, le conseil 26 le montre, en offrant un constructeur qui prend une référence universelle, celui-ci (et non le constructeur de copie) est appelé lors de la copie de lvalue non `const`. Ce conseil explique également que si une classe de base déclare un constructeur de transmission parfaite, celui-ci est généralement appelé lorsque des classes dérivées implémentent leurs constructeurs de copie et de déplacement de façon classique, même si le comportement correct voudrait que les constructeurs de copie et de déplacement de la classe de base soient invoqués.

Dans de telles situations, lorsqu'une fonction surchargée prenant une référence universelle est plus gourmande que nous le voudrions, mais pas suffisamment pour jouer le rôle d'une seule fonction de distribution, la solution du *tag dispatching* ne convient pas. Nous avons besoin d'une technique différente, qui nous permette de réduire les conditions d'emploi du template de fonction dont fait partie la référence universelle. La solution se nomme `std::enable_if`.

`std::enable_if` permet d'obliger le compilateur à considérer qu'un template précis n'existe pas. De tels templates sont dits *inactifs*. Par défaut, tous les templates sont *actifs*, mais un template qui utilise `std::enable_if` ne l'est que si la condition indiquée à `std::enable_if` est satisfaite. Dans notre cas, nous souhaitons activer le constructeur de transmission parfaite de `Person` uniquement si le type passé n'est pas `Person`. Dans le cas contraire, nous voulons le désactiver (autrement dit, qu'il soit ignoré par le compilateur), car cela conduit au traitement de l'appel par le constructeur de copie ou de déplacement de la classe, précisément ce que nous souhaitons lorsqu'un objet `Person` est initialisé à partir d'un autre objet `Person`.

La manière d'exprimer cette idée ne pose pas de difficulté particulière, mais la syntaxe peut être rebutante, notamment si vous ne l'avez jamais rencontrée auparavant. Nous allons donc vous l'expliquer progressivement. Du code passe-partout entoure la condition indiquée dans `std::enable_if` et nous allons commencer par ce point. Voici la déclaration du constructeur de transmission parfaite de `Person`, où n'est montré que l'indispensable pour utiliser `std::enable_if`. Seule la déclaration de ce constructeur est donnée, car l'utilisation de `std::enable_if` n'a aucune conséquence sur l'implémentation de la fonction. Elle reste identique à celle du conseil 26.

Copyright © 2016 Dunod.

```
class Person {
public:
  template<typename T,
           typename = typename std::enable_if<condition>::type>
  explicit Person(T&& n);

  ...

};
```

Pour comprendre précisément ce qui se passe dans le code mis en exergue, nous avons le regret de vous conseiller de consulter d'autres sources, car les explications détaillées sont longues et nous n'avons pas assez de place dans cet ouvrage. (Faites une recherche sur « SFINAE » et sur `std::enable_if`, car le fonctionnement de `std::enable_if` repose sur la technologie SFINAE.) Nous préférons nous focaliser sur l'expression de la condition qui détermine l'activation du constructeur.

Notre condition est que `T` ne soit pas un `Person`, autrement dit que le template du constructeur ne soit activé que si `T` est un type autre que `Person`. Grâce au trait de type `std::is_same` qui permet de savoir si deux types sont identiques, nous pouvons imaginer que la condition appropriée est `!std::is_same<Person, T>::value`. (Notez-le « `!` » au début de l'expression. Nous voulons que `Person` et `T` ne soient *pas* identiques.) Nous approchons de la bonne réponse, mais celle-ci n'est pas tout à fait correcte car, comme explique le conseil 28, le type déduit pour une référence universelle initialisée à partir d'une lvalue est toujours une référence lvalue. Autrement dit, pour un code semblable au suivant :

```
Person p("Nancy");

auto cloneOfP(p);          // Initialiser à partir d'une lvalue.
```

le type déduit pour `T` dans le constructeur universel sera `Person&`. Puisque les types `Person` et `Person&` sont différents, le résultat de `std::is_same<Person, Person&>::value` sera égal à faux.

En réfléchissant plus précisément à ce que nous entendons par activation du template de constructeur de `Person` uniquement si `T` n'est pas un `Person`, nous réalisons que lorsque nous examinons `T`, nous voulons ignorer deux aspects :

- **S'il s'agit d'une référence.** Dans notre contexte, les types `Person`, `Person&` et `Person&&` doivent être considérés comme identiques à `Person`.
- **S'il est `const` ou `volatile`.** De notre point de vue, un `const Person`, un `volatile Person` et un `const volatile Person` sont identiques à `Person`.

Nous devons donc retirer de `T` les références et spécificateurs `const` et `volatile` avant de vérifier si son type est identique à `Person`. Une fois encore, la bibliothèque standard répond à notre besoin sous forme d'un trait de type : `std::decay`. `std::decay<T>::type` équivaut à `T`, à l'exception du retrait des références et des qualificatifs `const` ou `volatile`. (Nous ne disons pas toute la vérité car, comme son

© Dunod – Toute reproduction non autorisée est un délit.
Copyright © 2016 Dunod.

nom le suggère, `std::decay` convertit également les types tableau et fonction en pointeurs [voir le conseil 1] mais, dans le contexte de cette discussion, `std::decay` se comporte comme nous le décrivons.) La condition qui détermine l'activation de notre constructeur devient donc :

```
!std::is_same<Person, typename std::decay<T>::type>::value
```

Autrement dit, `Person` n'est pas identique au type `T`, sans tenir compte des références ni des qualificatifs `const` ou `volatile`. (Comme l'explique le conseil 9, le mot clé « `typename` » qui précède `std::decay` est indispensable, car le type `std::decay<T>::type` dépend du paramètre de template `T`.)

En insérant cette condition dans l'instruction `std::enable_if` précédente et en mettant le code en forme de façon à mieux distinguer les différents éléments, nous obtenons la déclaration suivante pour le constructeur de transmission parfaite de `Person` :

```
class Person {
public:
  template<
    typename T,
    typename = typename std::enable_if<
                  !std::is_same<Person,
                                typename std::decay<T>::type
                               >::value
               >::type
  >
  explicit Person(T&& n);

  …

};
```

Vous n'avez sans doute jamais rencontré un tel code auparavant, alors réjouissez-vous de votre bonne fortune. Nous avons gardé cette conception pour la fin car nous avions une bonne raison. En effet, si l'une des autres approches permet d'éviter la combinaison des références universelles et de la surcharge (c'est quasiment toujours le cas), il faut l'adopter. Cependant, dès que la syntaxe fonctionnelle et la prolifération des crochets obliques ne gênent plus, cette dernière solution n'est pas si mauvaise, d'autant qu'elle apporte le comportement recherché. Avec la déclaration précédente, la construction d'un `Person` à partir d'un autre `Person` – lvalue ou rvalue, `const` ou non, `volatile` ou non – ne se fera jamais en invoquant le constructeur qui prend une référence universelle.

Et voilà, nous avons réussi !

En réalité, pas totalement. Un point mentionné au conseil 26 se balade encore et nous devons le fixer.

Supposons qu'une classe dérivée de `Person` implémente les opérations de copie et de déplacement de manière conventionnelle :

Copyright © 2016 Dunod.

```
class SpecialPerson: public Person {
public:
  SpecialPerson(const SpecialPerson& rhs)  // Constructeur de copie ;
  : Person(rhs)                            // appelle le constructeur de
  { … }                                    // transmission de la classe
                                           // de base !

  SpecialPerson(SpecialPerson&& rhs)     // Constructeur de déplacement ;
  : Person(std::move(rhs))               // appelle le constructeur de
  { … }                                  // transmission de la classe
                                         // de base !

  …
};
```

Il s'agit du code que nous avons déjà montré au conseil 26, avec les mêmes commentaires, qui, hélas, restent d'actualité. Lorsque nous copions ou déplaçons un objet `SpecialPerson`, nous supposons que la copie ou le déplacement des éléments de sa classe de base se fera avec les constructeurs de copie et de déplacement de cette classe. Cependant, dans ces fonctions, nous passons des objets `SpecialPerson` aux constructeurs de la classe de base et, puisque `SpecialPerson` n'est pas identique à `Person` (pas même après l'application de `std::decay`), le constructeur pour les références universelles déclaré dans la classe de base est activé et instancié afin de donner une correspondance exacte pour un argument `SpecialPerson`. Cette correspondance exacte surpasse les conversions de classe dérivée vers la classe de base qui seraient nécessaires pour lier les objets `SpecialPerson` aux paramètres `Person` des constructeurs de copie et de déplacement de `Person`. Par conséquent, avec le code actuel, la copie et le déplacement d'objets `SpecialPerson` confient au constructeur de transmission parfaite de `Person` la copie ou le déplacement des éléments de la classe de base ! Nous voilà de retour face aux problèmes décrits au conseil 26.

La classe dérivée se contente de suivre les règles normales d'implémentation des constructeurs de copie et de déplacement dans une classe dérivée. La correction du problème doit donc se faire dans la classe de base et, plus précisément, au niveau de la condition d'activation du constructeur pour les références universelles de `Person`. Nous comprenons à présent que nous ne voulons pas activer ce constructeur pour n'importe quel type d'argument autre que `Person`, mais pour n'importe quel type d'argument autre que `Person` ou *dérivé de `Person`*. Fichu héritage !

Vous ne devriez pas être surpris d'apprendre que, parmi les traits de type standard, il en existe un qui permet de déterminer si un type dérive d'un autre : `std::is_base_of`. `std::is_base_of<T1, T2>::value` est vrai si `T2` hérite de `T1`. Puisque l'on considère qu'un type dérive de lui-même, `std::is_base_of<T, T>::value` est vrai. Cela va se révéler commode, car nous voulons revoir la condition sur le constructeur de transmission parfaite de `Person` afin qu'il ne soit activé que si le type `T`, après avoir retiré les références et les qualificatifs `const` et `volatile`, n'est ni `Person` ni une classe dérivée de `Person`. En remplaçant `std::is_same` par `std::is_base_of`, nous arrivons à nos fins :

Copyright © 2016 Dunod.
© Dunod – Toute reproduction non autorisée est un délit.

```
class Person {
public:
  template<
    typename T,
    typename = typename std::enable_if<
                 !std::is_base_of<Person,
                                  typename std::decay<T>::type
                                 >::value
               >::type
  >
  explicit Person(T&& n);

  …

};
```

Avec C++11, nous avons terminé le code. Si nous utilisons C++14, ce code fonctionne encore, mais nous pouvons exploiter les alias de template pour nous débarrasser de « `typename` » et de « `::type` » :

```
class Person {                                        // C++14.
public:
  template<
    typename T,
    typename = std::enable_if_t<                      // Moins de code ici.
                 !std::is_base_of<Person,
                                  std::decay_t<T>     // Et ici.
                                 >::value
               >                                      // Et là.
  >
  explicit Person(T&& n);

  …

};
```

D'accord, nous avons menti. Ce n'est pas tout à fait terminé, mais nous sommes vraiment très proches de la fin.

Nous avons vu comment utiliser `std::enable_if` de façon à désactiver le constructeur pour les références universelles de `Person` lorsque le type de l'argument doit être pris en charge par les constructeurs de copie et de déplacement de la classe, mais nous n'avons pas encore vu comment l'appliquer pour différencier les arguments entiers et non entiers. Il s'agissait après tout de notre objectif initial, ce problème d'ambiguïté sur le constructeur n'étant apparu qu'en cours de route.

Tout ce que nous avons à faire est (1) d'ajouter une surcharge de construction d'un `Person` pour traiter les arguments entiers et (2) d'augmenter les contraintes sur le template de constructeur afin de le désactiver pour de tels arguments :

```
class Person {
public:
  template<
```

Copyright © 2016 Dunod.

```
    typename T,
    typename = std::enable_if_t<
      !std::is_base_of<Person, std::decay_t<T>>::value
      &&
      !std::is_integral<std::remove_reference_t<T>>::value
    >
  >
  explicit Person(T&& n)          // Constructeur pour std::string et
  : name(std::forward<T>(n))      // les arguments convertibles en
  { … }                           // std::string.

  explicit Person(int idx)        // Constructeur pour les arguments
  : name(nameFromIdx(idx))        // entiers.
  { … }

  …                               // Constructeurs de copie et
                                  // de déplacement, etc.

private:
  std::string name;
};
```

Et voilà ! La beauté du code n'apparaîtra peut-être qu'à l'amateur de métaprogrammation de template, mais il n'en reste pas moins que cette approche permet d'accomplir le travail et de belle manière. Puisqu'elle se fonde sur la transmission parfaite, nous obtenons une efficacité maximale, et, puisqu'elle maîtrise la combinaison des références universelles et de la surcharge au lieu de la bannir, elle peut être appliquée dans des situations (par exemple pour les constructeurs) où la surcharge est inévitable.

Avantages et inconvénients

Les trois premières techniques décrites dans ce conseil – abandonner la surcharge, passer par `const T&` et passer par valeur – spécifient un type pour chaque paramètre des fonctions à appeler. Les deux dernières techniques – *tag dispatching* et contraindre l'éligibilité d'un template – exploitent la transmission parfaite et ne spécifient donc pas les types des paramètres. Ce choix fondamental, spécifier ou non un type, a des conséquences.

En règle générale, la transmission parfaite est plus efficace car elle évite la création d'objets temporaires dans le seul but de se conformer au type de la déclaration d'un paramètre. Dans le cas du constructeur de `Person`, elle permet de transmettre une chaîne de caractères littérale, comme `"Nancy"`, au constructeur du `std::string` qui se trouve à l'intérieur de `Person`. En revanche, les techniques qui n'utilisent pas la transmission parfaite doivent créer un objet `std::string` temporaire à partir de la chaîne littérale afin de respecter la spécification de paramètre du constructeur de `Person`.

La transmission parfaite a toutefois des inconvénients. Tout d'abord, certains arguments ne peuvent pas être transmis parfaitement, même s'ils peuvent être passés à

Copyright © 2016 Dunod.
© Dunod – Toute reproduction non autorisée est un délit.

des fonctions qui prennent des types spécifiques. Le conseil 30 détaille ces cas où la transmission parfaite échoue.

Ensuite, les messages d'erreur produits lorsque le client passe des arguments invalides ne sont pas toujours très compréhensibles. Supposons, par exemple, qu'un client qui crée un objet `Person` passe une chaîne de caractères littérale constituée de `char16_t` (un type apporté par C++11 pour représenter des caractères sur 16 bits) à la place de `char` (dont est constitué un `std::string`) :

```
Person p(u"Konrad Zuse");   // "Konrad Zuse" est composé de
                            // caractères de type const char16_t.
```

Avec les trois premières approches étudiées dans ce conseil, le compilateur verra que les constructeurs disponibles prennent un `int` ou un `std::string`, et générera donc un message d'erreur plus ou moins clair indiquant que la conversion depuis un `const char16_t[12]` vers un `int` ou un `std::string` est impossible.

En revanche, avec l'approche fondée sur la transmission parfaite, la liaison du tableau de `const char16_t` avec le paramètre du constructeur ne pose aucune difficulté. Il est ensuite transmis au constructeur de la donnée membre `std::string` de `Person`, et c'est à ce moment-là que l'incompatibilité entre la donnée passée par l'appelant (un tableau de `const char16_t`) et ce qui est attendu (tout type accepté par le constructeur de `std::string`) est découverte. Le message d'erreur résultant risque d'être quelque peu impressionnant. Avec l'un des compilateurs que nous utilisons, il occupe plus de 160 lignes.

Dans cet exemple, la référence universelle est transmise une seule fois (depuis le constructeur de `Person` au constructeur de `std::string`), mais plus le système est complexe, plus une référence universelle traversera de multiples niveaux d'appels de fonctions avant d'arriver finalement au point où la validité des types des arguments est déterminée. Avec un nombre plus élevé de transmissions de la référence universelle, le message d'erreur risque fort d'être assez déroutant. De nombreux développeurs estiment que ce seul problème suffit à réserver les paramètres de type référence universelle aux interfaces qui exigent de bonnes performances.

Dans le cas de `Person`, nous savons que le paramètre de type référence universelle de la fonction de transmission sert à l'initialisation d'un `std::string`. Nous pouvons donc utiliser `static_assert` pour vérifier qu'il peut jouer ce rôle. Le trait de type `std::is_constructible` effectue un test à la compilation pour déterminer si un objet d'un type peut être construit à partir d'un objet (ou d'un ensemble d'objets) de type (ou d'un ensemble de types) différent. Il est donc facile d'écrire l'assertion :

```
class Person {
public:
  template<                                   // Comme précédemment.
    typename T,
    typename = std::enable_if_t<
      !std::is_base_of<Person, std::decay_t<T>>::value
      &&
      !std::is_integral<std::remove_reference_t<T>>::value
```

Copyright © 2016 Dunod.

```
    >
  >
  explicit Person(T&& n)
  : name(std::forward<T>(n))
  {
    // Confirmer qu'un std::string peut être créé à partir d'un objet T.
    static_assert(
      std::is_constructible<std::string, T>::value,
      "Le paramètre n ne peut pas servir à construire un std::string"
   );

    ...                // Les actions du constructeur se font ici.

  }

  ...                  // Suite de la classe Person (comme précédemment).

};
```

Le message d'erreur indiqué est affiché si le code client tente de créer un `Person` à partir d'un type qui ne peut pas servir à construire un `std::string`. Malheureusement, dans cet exemple, `static_assert` se trouve dans le corps du constructeur, tandis que le code de transmission, qui fait partie de la liste d'initialisation du membre, le précède. Avec les compilateurs que nous utilisons, le joli message compréhensible généré par `static_assert` apparaît uniquement *après* les messages d'erreur habituels (après plus de 160 lignes).

À retenir

- Les alternatives à la combinaison des références universelles et de la surcharge sont l'emploi de noms de fonctions différents, le passage des paramètres sous forme de références lvalue sur des `const`, le passage de paramètres par valeur et la mise en œuvre du *tag dispatching*.
- Contraindre les templates avec `std::enable_if` permet d'utiliser conjointement les références universelles et la surcharge, mais cela limite les conditions sous lesquelles le compilateur peut utiliser les surcharges sur les références universelles.
- Les paramètres de type références universelles apportent souvent des avantages au niveau de l'efficacité, mais des inconvénients sur le plan de la facilité d'utilisation.

CONSEIL N° 28. COMPRENDRE LA RÉDUCTION DE RÉFÉRENCE

Le conseil 23 fait la remarque suivante : lorsqu'un argument est passé à une fonction template, le type déduit pour le paramètre du template encode le fait que l'argument est une lvalue ou une rvalue. Elle oublie cependant de mentionner que cela ne se produit que si l'argument sert à l'initialisation d'un paramètre qui est une référence

Copyright © 2016 Dunod.
© Dunod – Toute reproduction non autorisée est un délit.

universelle. Il y a une bonne raison à cette omission : la présentation des références universelles n'arrive qu'au conseil 24. Prenons le template suivant et voyons ce que signifient ces observations sur les références universelles et l'encodage de l'information lvalue/rvalue :

```
template<typename T>
void func(T&& param);
```

Le type déduit pour le paramètre de template `T` indiquera si l'argument passé à `param` est une lvalue ou une rvalue.

Le mécanisme d'encodage est simple. Lorsqu'une lvalue est passée en argument, le type déduit pour `T` est une référence lvalue. Lorsqu'une rvalue est transmise, `T` devient une non-référence. (Notez l'asymétrie : les lvalues sont encodées comme des références lvalue, tandis que les rvalues sont encodées comme des *non-références*.) Examinons le code suivant :

```
Widget widgetFactory();     // Fonction qui retourne une rvalue.

Widget w;                   // Une variable (une lvalue).

func(w);                    // Appeler func avec une lvalue ; T est
                            // déduit de type Widget&.

func(widgetFactory());      // Appeler func avec une rvalue ; T est
                            // déduit de type Widget.
```

Un `Widget` est passé dans les deux appels à `func`. Pourtant, puisque l'un des `Widget` est une lvalue et l'autre une rvalue, les types déduits pour le paramètre de template `T` sont différents. Nous le verrons plus loin, cela détermine si des références universelles deviennent des références rvalue ou des références lvalue, et sert de mécanisme sous-jacent au fonctionnement de `std::forward`.

Avant d'étudier en détail `std::forward` et les références universelles, nous devons préciser que les références sur les références sont illégales en C++. Si nous tentons d'en déclarer une, le compilateur nous réprimande :

```
int x;
...
auto& & rx = x;   // Erreur ! Une référence à une référence
                  // est interdite.
```

Voyons ce qui arrive lorsqu'une lvalue est passée à un template de fonction qui prend une référence universelle :

```
template<typename T>
void func(T&& param);    // Comme précédemment.

func(w);                 // Invoquer func avec une lvalue ;
                         // le type déduit pour T est Widget&.
```

Copyright © 2016 Dunod.

Si nous utilisons le type déduit pour `T` (c'est-à-dire `Widget&`) de façon à instancier le template, nous obtenons l'instruction suivante :

```
void func(Widget& && param);
```

Une référence à une référence ! Pourquoi le compilateur ne proteste-t-il pas ? Nous vous avons appris au conseil 24 que, en raison de l'initialisation de la référence universelle `param` avec une lvalue, le type de `param` est supposé être une référence lvalue, mais comment le compilateur fait-il pour partir du type déduit pour `T` et le remplacer dans le template par le suivant, afin d'obtenir cette signature finale pour la fonction ?

```
void func(Widget& param);
```

La réponse tient dans la *réduction de référence* (*reference collapsing*). C'est exact, il nous est interdit de déclarer des références à des références, mais le compilateur a le droit d'en générer dans des contextes spécifiques, notamment l'instanciation d'un template. Lorsque le compilateur génère des références à des références, la réduction de référence détermine ce qu'il advient ensuite.

Puisqu'il existe deux sortes de référence (lvalue et rvalue), quatre combinaisons référence-référence sont possibles : lvalue sur lvalue, lvalue sur rvalue, rvalue sur lvalue et rvalue sur rvalue. Si une référence à une référence se trouve dans un contexte qui les autorise (par exemple au cours de l'instanciation d'un template), les références sont *réduites* en une seule référence conformément à la règle suivante :

> Si l'une des références est une référence lvalue, le résultat est une référence lvalue. Sinon, c'est-à-dire si les deux références sont des références rvalue, le résultat est une référence rvalue.

Dans l'exemple précédent, la substitution du type déduit `Widget&` dans le template de `func` génère une référence rvalue à une référence lvalue. La règle de réduction des références nous permet donc de savoir que le résultat est une référence lvalue.

La réduction de référence est essentielle au fonctionnement de `std::forward`. Comme l'explique le conseil 25, `std::forward` est appliquée à des paramètres de type références universelles. Voici donc l'utilisation classique :

```
template<typename T>
void f(T&& fParam)
{
  ...                                    // Faire quelque chose.

  someFunc(std::forward<T>(fParam));     // Transmettre fParam à
}                                        // someFunc.
```

Puisque `fParam` est une référence universelle, nous savons que le paramètre de type `T` indiquera si l'argument passé à `f` (c'est-à-dire l'expression utilisée pour initialiser

Copyright © 2016 Dunod.
© Dunod – Toute reproduction non autorisée est un délit.

fParam) est une lvalue ou une rvalue. Le rôle de std::forward est de convertir fParam (une lvalue) en une rvalue si et seulement si T encode le fait que l'argument passé à f est une rvalue, autrement dit si T n'est pas un type référence.

Voici une implémentation possible pour std::forward :

```
template<typename T>                                // Dans l'espace
T&& forward(typename                                // de noms std.
              remove_reference<T>::type& param)
{
  return static_cast<T&&>(param);
}
```

Elle n'est pas vraiment conforme à la norme (quelques détails d'interface ont été omis), mais ces différences n'ont pas d'intérêt pour comprendre le comportement de std::forward.

Supposons que l'argument passé à f soit une lvalue de type Widget. Le type déduit pour T sera Widget& et l'appel à std::forward va prendre la forme std::forward<Widget&>. En insérant Widget& dans l'implémentation de std::forward, nous obtenons le code suivant :

```
Widget& && forward(typename
                     remove_reference<Widget&>::type& param)
{ return static_cast<Widget& &&>(param); }
```

Puisque le trait de type std::remove_reference<Widget&>::type donne Widget (voir le conseil 9), std::forward devient :

```
Widget& && forward(Widget& param)
{ return static_cast<Widget& &&>(param); }
```

La réduction de référence est également appliquée au type de retour et à la conversion de type, ce qui donne la version finale suivante de std::forward pour l'appel étudié :

```
Widget& forward(Widget& param)                  // Toujours dans l'espace
{ return static_cast<Widget&>(param);  }        // de noms std.
```

Vous le constatez, lorsqu'un argument lvalue est passé à la fonction template f, std::forward est instancié de façon à prendre et à renvoyer une référence lvalue. La conversion de type qui se trouve dans std::forward n'a pas d'incidence car param est déjà de type Widget& (le convertir en Widget& n'a aucun effet). Un argument lvalue passé à std::forward va donc retourner une référence lvalue. Puisque, par définition, les références lvalue sont des lvalues, passer une lvalue à std::forward provoque donc le retour d'une lvalue, exactement comme attendu.

Supposons à présent que l'argument passé à f soit une rvalue de type Widget. Dans ce cas, le type déduit pour le paramètre de type T de f sera simplement Widget. L'appel à

Copyright © 2016 Dunod.

`std::forward` dans `f` correspondra donc à `std::forward<Widget>`. Si nous remplaçons `T` par `Widget` dans l'implémentation de `std::forward`, nous obtenons le code suivant :

```
Widget&& forward(typename
                 remove_reference<Widget>::type& param)
{ return static_cast<Widget&&>(param); }
```

L'application de `std::remove_reference` à `Widget`, qui n'est pas type référence, donne le type de départ (`Widget`). `std::forward` devient donc :

```
Widget&& forward(Widget& param)
{ return static_cast<Widget&&>(param); }
```

Puisque ce code ne comprend aucune référence à une référence, la réduction de référence n'entre pas en scène et il correspond à la version finale de `std::forward` instanciée pour l'appel.

Les références rvalue renvoyées par des fonctions étant des rvalues, `std::forward` va, dans ce cas, transformer le paramètre `fParam` (une lvalue) de `f` en une rvalue. Un argument rvalue passé à `f` sera finalement transmis à `someFunc` comme une rvalue, exactement comme attendu.

En C++14, `std::remove_reference_t` permet d'implémenter `std::forward` de façon plus concise :

```
template<typename T>                          // C++14 ; toujours dans
T&& forward(remove_reference_t<T>& param)     // l'espace de noms std.
{
  return static_cast<T&&>(param);
}
```

La réduction de référence se produit dans quatre contextes. Le premier, et le plus courant, concerne l'instanciation de template. La génération de type pour les variables `auto` représente le deuxième. Le fonctionnement est essentiellement identique à celui déroulé pour les templates, car la déduction de type pour les variables `auto` est fondamentalement identique à celle mise en place pour les template (voir le conseil 2). Reprenons cet exemple donné précédemment :

```
template<typename T>
void func(T&& param);

Widget widgetFactory();      // Fonction qui retourne une rvalue.

Widget w;                    // Une variable (une lvalue).

func(w);                     // Appeler func avec une lvalue ; T est
                             // déduit de type Widget&.

func(widgetFactory());       // Appeler func avec une rvalue ; T est
                             // déduit de type Widget.
```

Copyright © 2016 Dunod.
© Dunod – Toute reproduction non autorisée est un délit.

Voyons ce qui se passe avec une variable `auto`. La déclaration

```
auto&& w1 = w;
```

initialise `w1` à partir d'une lvalue, conduisant au type déduit `Widget&` pour `auto`. En remplaçant `auto` par `Widget&` dans la déclaration de `w1`, nous obtenons un code qui comprend une référence à une référence :

```
Widget& && w1 = w;
```

Suite à la réduction de référence, nous arrivons à :

```
Widget& w1 = w;
```

Par conséquent, `w1` est une référence lvalue.

La déclaration

```
auto&& w2 = widgetFactory();
```

initialise `w2` à partir d'une rvalue, conduisant au type déduit `Widget` pour `auto` ; ce type déduit n'est pas une référence. En remplaçant `auto` par `Widget`, nous obtenons :

```
Widget&& w2 = widgetFactory();
```

Puisque cette ligne ne contient pas de référence à une référence, le traitement est terminé et `w2` est une référence rvalue.

Nous pouvons à présent vraiment comprendre les références universelles introduites au conseil 24. Une référence universelle n'est pas un nouveau genre de référence mais une référence rvalue dans un contexte où deux conditions sont satisfaites :

- **La déduction de type distingue les lvalues et les rvalues**. Les lvalues de type `T` ont pour type déduit `T&`, tandis que les rvalues de type `T` ont pour type déduit `T`.
- **La réduction de référence a lieu**.

Le concept de référence universelle est utile car il nous évite d'avoir à reconnaître l'existence de contextes de réduction des références, à déduire mentalement des types différents pour les lvalues et les rvalues, et à appliquer la règle de réduction de référence après avoir mentalement remplacé les types déduits dans les contextes concernés.

Nous avons mentionné quatre contextes, mais n'en avons présentés que deux : instanciation de template et génération de type `auto`. Le troisième correspond à la génération et à l'utilisation de `typedef` et des déclarations d'alias (voir le conseil 9). Si, au cours de la création ou de l'évaluation d'un `typedef`, des références à des références surgissent, la réduction de référence intervient pour les éliminer. Supposons, par exemple, que nous ayons une classe template `Widget` qui comprend un `typedef` pour un type de référence rvalue :

Copyright © 2016 Dunod.

```
template<typename T>
class Widget {
public:
  typedef T&& RvalueRefToT;
  ...
};
```

Supposons également que nous instanciions `Widget` avec un type de référence lvalue :

```
Widget<int&> w;
```

En remplaçant `T` par `int&` dans le template `Widget`, nous obtenons le `typedef` suivant :

```
typedef int& && RvalueRefToT;
```

La réduction de référence arrive à :

```
typedef int& RvalueRefToT;
```

Nous en concluons que le nom choisi pour le `typedef` n'est peut-être pas aussi adapté que nous l'espérions : *`RvalueRefToT`* est un `typedef` pour une *référence lvalue* quand `Widget` est instancié avec un type qui correspond à une référence lvalue.

Le dernier contexte d'application de la réduction de référence correspond aux utilisations de `decltype`. Si une référence à une référence apparaît au cours de l'analyse d'un type qui implique `decltype`, la réduction de référence va se charger de l'éliminer. (Pour de plus amples informations sur `decltype`, consulter le conseil 3.)

À retenir

- La réduction de référence intervient dans quatre contextes : instanciation de template, génération de type `auto`, création et utilisation de `typedef` et de déclarations d'alias, et utilisation de `decltype`.
- Lorsque le compilateur génère une référence à une référence dans un contexte de réduction de référence, le résultat est une seule référence. Si l'une des références d'origine est une référence lvalue, le résultat est une référence lvalue. Sinon, il s'agit d'une référence rvalue.
- Les références universelles sont des références rvalue dans les contextes où la déduction de type fait une différence entre les lvalues et les rvalues, et où la réduction de référence se produit.

Copyright © 2016 Dunod.
© Dunod – Toute reproduction non autorisée est un délit.

CONSEIL N° 29. SUPPOSER QUE LES OPÉRATIONS DE DÉPLACEMENT SONT ABSENTES, ONÉREUSES ET INUTILISÉES

La sémantique de déplacement est peut-être bien la principale nouveauté de C++11. Vous entendrez probablement certaines personnes dire que « Déplacer des conteneurs est à présent aussi efficace que copier des pointeurs ! » ou que « La copie d'objets temporaires est à présent tellement efficace que s'efforcer de l'éviter dans le code équivaut à une optimisation prématurée ! ». Ces analyses sont faciles à comprendre. La sémantique de déplacement est vraiment une fonctionnalité importante. Non seulement elle permet au compilateur de remplacer des opérations de copie onéreuses par des déplacements plus rapides, mais elle lui *impose* de procéder ainsi (lorsque les conditions d'application sont satisfaites). Prenez votre base de code C++98, soumettez-la à un compilateur C++11 accompagné de la bibliothèque standard correspondante, et, pouf !, le logiciel s'exécute plus rapidement.

La sémantique de déplacement permet réellement d'éviter des copies, ce qui l'a élevée au rang de légende. Cependant, les légendes découlent généralement d'une exagération. Dans ce conseil, notre objectif est de donner à vos attentes un certain réalisme.

Observons tout d'abord que de nombreux types ne sont pas compatibles avec la sémantique de déplacement. L'intégralité de la bibliothèque standard de C++98 a été revue pour C++11 de façon à ajouter des opérations de déplacement aux types pour lesquels le déplacement pouvait être plus rapide que la copie. Les composants de la bibliothèque ont également été adaptés pour exploiter ces opérations. Toutefois, il est probable que la base de code manipulée n'ait pas été totalement refondue pour tirer parti de C++11. Pour les types de nos applications (ou des bibliothèques que nous utilisons) qui n'ont pas été remaniés pour C++11, la prise en charge du déplacement par le compilateur n'apportera pas grand-chose. Il est vrai que C++11 est prêt à générer des opérations de déplacement pour les classes qui n'en disposent pas, mais cela ne concerne que les classes qui ne déclarent aucune opération de copie, opération de déplacement, ni destructeur (voir le conseil 17). Lorsqu'une donnée membre ou une classe de base a un type pour lequel le déplacement a été désactivé (par exemple en supprimant les opérations de déplacement ; voir le conseil 11), la génération des opérations de déplacement par le compilateur est également désactivée. Pour les types sans prise en charge explicite du déplacement et non éligibles aux opérations de déplacement générées par le compilateur, il n'y a aucune raison d'attendre de C++11 une amélioration des performances par rapport à C++98.

Même les types qui prennent explicitement en charge le déplacement pourraient ne pas apporter autant de bénéfices qu'espéré. Par exemple, tous les conteneurs de la bibliothèque standard de C++11 disposent des opérations de déplacement, mais il serait erroné de croire que le déplacement de n'importe quel conteneur se fait à faible coût. Pour certains d'entre eux, il est tout bonnement impossible de déplacer leur contenu sans payer un prix élevé. Pour d'autres, les opérations de déplacement

Copyright © 2016 Dunod.

bon marché offertes posent des conditions que les éléments du conteneur ne peuvent pas remplir.

Examinons le nouveau conteneur `std::array` de C++11. Fondamentalement, il équivaut à un tableau intégré doté d'une interface STL. Il est donc différent des autres conteneurs standard, qui stockent leur contenu sur le tas. Conceptuellement, les objets créés à partir de ces types de conteneurs comprennent, sous forme d'une donnée membre, uniquement un pointeur sur la zone de mémoire dans le tas qui sert à stocker leur contenu. (La réalité est plus complexe, mais cela n'a pas d'importance pour notre analyse.) Grâce à ce pointeur, il est possible de déplacer l'intégralité du contenu du conteneur en un temps constant : il suffit de copier le pointeur sur le contenu du conteneur depuis le conteneur source vers la cible et de fixer le pointeur initial à nul (figure 5.1) :

```
std::vector<Widget> vw1;

// Ajouter des données à vw1.

...

// Déplacer vw1 dans vw2. L'exécution
// se fait en un temps constant.
// Seuls des pointeurs dans vw1
// et dans vw2 sont modifiés.
auto vw2 = std::move(vw1);
```

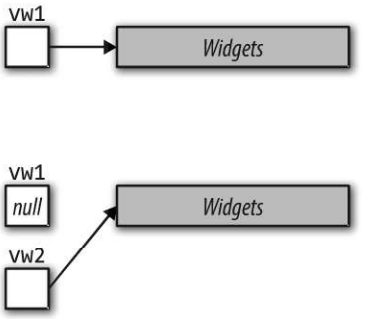


Figure 5.1 – Déplacement d'un conteneur par copie de pointeur.

Les objets `std::array` n'ont pas un tel pointeur, car les données contenues dans un `std::array` sont stockées directement dans l'objet `std::array` (figure 5.2) :

```
std::array<Widget, 10000> aw1;

// Ajouter des données à aw1.

...

// Déplacer aw1 dans aw2.
// L'exécution se fait en un temps
// linéaire.
// Tous les éléments de aw1
// sont déplacés dans aw2.
auto aw2 = std::move(aw1);
```

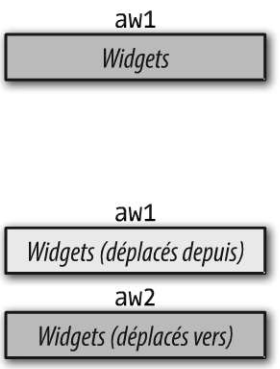


Figure 5.2 – Déplacement d'un conteneur par copie de ses éléments.

Les éléments de `aw1` sont *déplacés* dans `aw2`. En supposant que `Widget` soit un type pour lequel le déplacement est plus rapide que la copie, déplacer un `std::array` de `Widget` sera plus rapide que copier le même `std::array`. Par conséquent, `std::array` prend certainement en charge le déplacement. Cela dit, le déplacement et la copie d'un `std::array` sont des opérations dont la complexité en temps est linéaire, car chaque élément du conteneur doit être copié ou déplacé. On est loin du « déplacement d'un conteneur à présent aussi efficace que l'affectation de deux pointeurs », comme peuvent le clamer certains.

Copyright © 2016 Dunod.
© Dunod – Toute reproduction non autorisée est un délit.

Dans le cas de `std::string`, les déplacements se font en temps constant et les copies en temps linéaire. Il semblerait donc que le déplacement soit plus rapide que la copie, mais ce n'est pas nécessairement le cas. De nombreuses implémentations des chaînes de caractères mettent en œuvre l'*optimisation des petites chaînes* (SSO, *small string optimization*), dans laquelle les chaînes de caractères « courtes » (par exemple celles dont la capacité ne dépasse pas 15 caractères) sont mémorisées dans un tampon à l'intérieur de l'objet `std::string` ; aucun espace de stockage alloué sur le tas n'est utilisé. Dans une implémentation fondée sur la SSO, le déplacement de petites chaînes de caractères n'est pas plus rapide que leur copie, car l'astuce de la copie d'un seul pointeur qui avantage généralement les déplacements par rapport aux copies ne s'applique plus.

La SSO repose sur le constat que les chaînes courtes sont très répandues dans la plupart des applications. En stockant ces chaînes de caractères dans un tampon interne, l'allocation dynamique d'une zone de mémoire n'est plus requise. Des gains de performance en découlent généralement. Mais les déplacements ne sont plus, dans ce cas, plus rapides que les copies. Toutefois, nous pouvons raisonner dans le sens inverse et considérer que les copies ne sont plus plus lentes que les déplacements.

Même lorsque les types prennent en charge des opérations de déplacement rapides, il existe des situations où des copies sont effectuées alors que l'on imaginait assurément des déplacements. Le conseil 14 explique que certaines opérations des conteneurs de la bibliothèque standard apportent une garantie élevée sur la sécurité vis-à-vis des exceptions afin qu'un ancien code C++98 qui en dépend ne soit pas remis en question lors du passage à C++11. Les opérations de copie sous-jacentes sont alors remplacées par des opérations de déplacement uniquement si celles-ci sont assurées de ne pas lever d'exception. En conséquence, même si un type offre des opérations de déplacement plus efficaces que les opérations de copie correspondantes et même si, à un endroit particulier du code, une opération de déplacement serait normalement appropriée (par exemple si l'objet source est une rvalue), le compilateur peut être obligé d'invoquer une opération de copie car l'opération de déplacement correspondante n'est pas déclarée `noexcept`.

Voici plusieurs scénarios dans lesquels la sémantique de déplacement de C++11 ne fait aucun bien :

- **Aucune opération de déplacement :** l'objet à partir duquel se fait le déplacement ne propose aucune opération de déplacement. La demande de déplacement se transforme donc en demande de copie.
- **Le déplacement n'est pas plus rapide :** l'objet à partir duquel se fait le déplacement offre des opérations de déplacement dont l'efficacité n'est pas supérieure à celle de ses opérations de copie.
- **Le déplacement n'est pas utilisable :** le contexte dans lequel le déplacement devrait avoir lieu nécessite une opération de déplacement qui ne lève pas d'exception, mais cette opération n'est pas déclarée `noexcept`.

Il existe également un autre scénario dans lequel la sémantique de déplacement n'apporte aucune amélioration :

Copyright © 2016 Dunod.

- **L'objet source est une lvalue :** hormis de très rares exceptions (voir le conseil 25), seules des rvalues peuvent être employées comme source d'une opération de déplacement.

Mais l'intitulé de ce conseil indique qu'il faut *supposer* que les opérations de déplacement sont absentes, onéreuses et inutilisées. C'est généralement le cas dans le développement d'un code générique, par exemple l'écriture de templates, car nous ne pouvons pas connaître tous les types manipulés. Dans de telles circonstances, nous devons être aussi prudents vis-à-vis de la copie des objets que nous l'étions en C++98, avant que la sémantique de déplacement n'existe. Cela concerne également le code « non stabilisé », c'est-à-dire celui dans lequel les caractéristiques des types utilisés font l'objet de modifications relativement fréquentes.

Toutefois, nous connaissons très souvent les types utilisés dans notre code et nous pouvons supposer que leurs caractéristiques ne changent pas (par exemple que leurs opérations de déplacement ne sont pas onéreuses). Lorsque c'est le cas, oublions les suppositions et examinons simplement les détails de la prise en charge du déplacement pour les types employés. Si leurs opérations de déplacement sont efficaces et si nous utilisons les objets dans des contextes où ces opérations seront invoquées, nous pouvons compter sur la sémantique de déplacement pour remplacer les opérations de copie par les opérations de déplacement équivalentes moins onéreuses.

À retenir

- Supposer que les opérations de déplacement sont absentes, onéreuses et inutilisées.
- Avec le code qui utilise des types connus ou qui prend en charge la sémantique de déplacement, ne faire aucune supposition.

CONSEIL N° 30. SE FAMILIARISER AVEC LES CAS D'ÉCHEC DE LA TRANSMISSION PARFAITE

La transmission parfaite est probablement l'une des fonctionnalités de C++11 les plus annoncées. En y regardant de plus près, nous découvrons que, derrière la notion de *parfait*, il y a un idéal et une réalité. La transmission parfaite de C++11 est excellente, mais elle n'atteint la perfection que si nous sommes prêts à oublier un ou deux petits détails. Ce conseil a pour objectif de vous familiariser avec ces détails.

Avant d'explorer ces détails, examinons ce que signifie « transmission parfaite ». La « transmission » intervient lorsqu'une fonction passe, *transmet*, ses paramètres à une autre. L'objectif est que la seconde fonction (la destination de la transmission) reçoive les objets reçus par la première fonction (la source de la transmission). Les paramètres passés par valeur sont donc exclus, car ils sont des *copies* des arguments passés par le code appelant. La fonction destinataire doit pouvoir manipuler les objets d'origine. Les paramètres de type pointeur sont également écartés, car nous ne voulons

Copyright © 2016 Dunod.
© Dunod – Toute reproduction non autorisée est un délit.

pas imposer le passage de pointeurs au code appelant. Autrement dit, dans le contexte général de la transmission, les paramètres seront des références.

La *transmission parfaite* signifie que nous ne voulons pas nous contenter de transmettre des objets mais également leurs caractéristiques essentielles : leur type, s'ils sont des lvalues ou des rvalues, et s'ils sont `const` ou `volatile`. Puisque nous avons expliqué que les paramètres sont des références, cela implique que nous utiliserons des références universelles (voir le conseil 24). En effet, seuls les paramètres de type références universelles permettent de savoir si les arguments passés sont des lvalues ou des rvalues.

Supposons que nous ayons une fonction `f` et que nous souhaitions écrire une fonction (en réalité un template de fonction) qui transmette ses paramètres à `f`. Voici le code de base correspondant :

```
template<typename T>
void fwd(T&& param)                 // Accepter n'importe quel argument.
{
  f(std::forward<T>(param));        // Le transmettre à f.
}
```

Les fonctions de transmission sont, par nature, génériques. Par exemple, le template `fwd` accepte n'importe quel type d'argument et retransmet celui qu'il reçoit, quel qu'il soit. La généricité de ces fonctions peut être logiquement étendue en les écrivant comme des templates variadiques, c'est-à-dire acceptant un nombre quelconque d'arguments. Voici la version variadique de `fwd` :

```
template<typename... Ts>
void fwd(Ts&&... params)            // Accepter n'importe quels arguments.
{
  f(std::forward<Ts>(params)...);   // Les transmettre à f.
}
```

C'est la forme que nous rencontrerons, entre autres, dans les fonctions de placement (voir le conseil 42) et dans les fonctions fabriques de pointeurs intelligents, `std::make_shared` et `std::make_unique` (voir le conseil 21).

Avec notre fonction cible `f` et notre fonction de transmission `fwd`, la transmission parfaite *échoue* si l'appel de `f` avec un argument spécifique effectue une certaine opération mais que l'appel de `fwd` avec le même argument en fait une autre :

```
f( expression );            // Si cet appel effectue une opération et
fwd( expression );          // que celui-ci en réalise une autre, fwd ne
                            // transmet pas parfaitement expression à f.
```

Plusieurs sortes d'arguments conduisent à un tel échec. Puisqu'il est important de les connaître et de savoir comment les contourner, passons en revue les sortes d'arguments incompatibles avec la transmission parfaite.

Copyright © 2016 Dunod.

Initialiseurs à accolades

Supposons que `f` soit déclarée de la manière suivante :

```
void f(const std::vector<int>& v);
```

La compilation d'un appel de `f` avec un initialiseur à accolades ne pose pas de problème :

```
f({ 1, 2, 3 });        // Parfait, "{1, 2, 3}" est implicitement
                       // converti en std::vector<int>.
```

En revanche, si nous passons à `fwd` le même initialiseur à accolades, le code ne compile pas :

```
fwd({ 1, 2, 3 });      // Erreur ! La compilation échoue.
```

En effet, l'utilisation d'un initialiseur à accolades représente l'un des cas d'échec de la transmission parfaite.

Tous ces cas d'échec ont une cause commune. Dans un appel direct à `f` (comme dans `f({ 1, 2, 3 })`), le compilateur voit les arguments passés au point d'appel, ainsi que les types des paramètres déclarés par `f`. Il compare les arguments indiqués dans l'appel aux déclarations des paramètres afin de déterminer leur compatibilité. Si nécessaire, il effectue des conversions implicites pour que l'appel puisse se faire. Dans l'exemple précédent, il génère un objet `std::vector<int>` temporaire à partir de `{ 1, 2, 3 }` de sorte que le paramètre `v` de `f` dispose d'un objet `std::vector<int>` auquel il puisse être lié.

Lors d'un appel indirect à `f` au travers du template de fonction de transmission `fwd`, le compilateur ne compare plus les arguments passés depuis le point d'appel dans `fwd` aux déclarations de paramètres dans `f`. À la place, il *déduit* les types des arguments transmis à `fwd` et les compare aux déclarations de paramètres de `f`. La transmission parfaite échoue dans les deux situations suivantes :

- **Le compilateur ne parvient pas à déduire un type** pour un ou plusieurs des paramètres de `fwd`. La compilation du code échoue alors.
- **Le compilateur déduit le « mauvais » type** pour un ou plusieurs des paramètres de `fwd`. Dans ce contexte, « mauvais » peut signifier que l'instanciation de `fwd` avec les types déduits ne compilera pas, mais également que l'appel à `f` avec les types déduits de `fwd` aura un comportement différent d'un appel direct à `f` avec les arguments qui ont été passés à `fwd`. Ce comportement différent peut se produire lors d'une surcharge du nom de la fonction `f`. En raison d'une déduction de type « incorrect », la surcharge de `f` appelée dans `fwd` pourrait être différente de celle qui serait invoquée avec un appel direct de `f`.

Dans l'appel « `fwd({ 1, 2, 3 })` » précédent, le problème vient du fait que le passage d'un initialiseur à accolades à un paramètre de template de fonction qui

Copyright © 2016 Dunod.
© Dunod – Toute reproduction non autorisée est un délit.

n'est pas déclaré `std::initializer_list` constitue, comme le définit la norme, « un contexte non déduit ». Autrement dit, dans l'appel à `fwd`, le compilateur n'a pas le droit de déduire un type pour l'expression `{ 1, 2, 3 }` car le paramètre de `fwd` n'est pas déclaré de type `std::initializer_list`. Puisqu'il est dans l'incapacité de déduire un type pour le paramètre de `fwd`, le compilateur n'a d'autre choix que de rejeter l'appel.

Il est intéressant de noter que, comme l'explique le conseil 2, la déduction de type fonctionne parfaitement pour les variables `auto` initialisées avec un initialiseur à accolades. Puisque ces variables sont considérées comme des objets `std::initializer_list`, nous disposons d'une solution simple pour les cas où le type déduit dans la fonction de transmission doit être `std::initializer_list`. Il suffit de déclarer une variable locale avec `auto` et de passer celle-ci à la fonction de transmission :

```
auto il = { 1, 2, 3 };      // Le type déduit pour il
                            // est std::initializer_list<int>.

fwd(il);                    // Transmission parfaite de il à f.
```

`0` ou `NULL` en tant que pointeurs nuls

Le conseil 8 explique que si nous essayons de passer à un template un pointeur nul représenté par `0` ou `NULL`, la déduction de type va de travers et conduit non pas à un type pointeur mais à un type entier (en général un `int`) pour l'argument passé à la place. Par conséquent, ni `0` ni `NULL` ne permet une transmission parfaite en tant que pointeur nul. Mais le problème est facile à corriger, puisqu'il suffit de passer non pas `0` ou `NULL`, mais `nullptr`. Les détails se trouvent dans le conseil 8.

Données membres `static const` intégrales uniquement déclarées

De façon générale, il est inutile de définir les données membres `static const` intégrales dans les classes, leur déclaration suffit. En effet, le compilateur effectue une *propagation de const* sur les valeurs de ces membres, ce qui permet d'éviter de leur réserver de la mémoire. Prenons par exemple le code suivant :

```
class Widget {
public:
  static const std::size_t MinVals = 28;   // Déclarer MinVals.
  …
};
…                                           // Aucune définition
                                            // pour MinVals.

std::vector<int> widgetData;
widgetData.reserve(Widget::MinVals);        // Utiliser MinVals.
```

Dans cet exemple, nous utilisons `Widget::MinVals` (ci-après simplement noté `MinVals`) pour préciser la capacité initiale de `widgetData`, même s'il n'existe aucune

Copyright © 2016 Dunod.

définition de `MinVals`. Le compilateur résout le problème de définition absente (comme il y est obligé) en mettant la valeur 28 partout où `MinVals` est utilisé. Le fait qu'aucune zone de mémoire n'est réservée à la valeur de `MinVals` ne pose pas de problème. Pour que l'adresse de `MinVals` puisse être prise (par exemple si le programmeur crée un pointeur sur `MinVals`), il faut qu'un espace de stockage soit réservé à `MinVals` (afin que le pointeur puisse pointer sur quelque chose). Dans ce cas, le code précédent compile, mais l'édition de liens échoue car `MinVals` n'a pas été défini.

Avec ces informations en tête, imaginons que `f` (la fonction à laquelle `fwd` transmet son argument) soit déclarée de la manière suivante :

```
void f(std::size_t val);
```

L'appel de `f` avec `MinVals` ne pose pas de problème, car le compilateur se contente de remplacer `MinVals` par sa valeur :

```
f(Widget::MinVals);          // Parfait, traité comme "f(28)".
```

Hélas, l'appel à `f` *via* `fwd` ne se passe pas aussi bien :

```
fwd(Widget::MinVals);    // Erreur ! L'édition de liens doit échouer.
```

La compilation de ce code réussit, mais son édition de liens doit échouer. Si cet exemple vous fait penser à ce qui se passe lorsque nous essayons de prendre l'adresse de `MinVals`, vous avez raison, car le problème sous-jacent est identique.

Certes, nous ne prenons pas l'adresse de `MinVals` dans le code source, mais le paramètre de `fwd` est une référence universelle et les références, dans le code généré par le compilateur, sont généralement traitées comme des pointeurs. Dans le code binaire qui correspond au programme (et au niveau matériel), les pointeurs et les références sont essentiellement la même chose. À ce niveau, les références peuvent être vues comme des pointeurs qui sont déréférencés automatiquement. Dans ces conditions, passer `MinVals` par référence équivaut à passer cette variable au travers d'un pointeur, qui a donc besoin d'une zone de mémoire sur laquelle pointer. Par conséquent, pour passer des données membres `static const` intégrales par référence, nous devons généralement les définir et cette contrainte peut conduire à l'invalidité du code s'il utilise la transmission parfaite.

Peut-être avez-vous remarqué les termes employés dans la discussion précédente. L'édition de liens « doit échouer ». Les références sont « généralement » considérées comme des pointeurs. Pour passer des données membres `static const` intégrales par référence, elles doivent « généralement » être définies. C'est comme si nous savions des choses que nous voulions cacher…

C'est bien le cas. Selon la norme, passer `MinVals` par référence exige que cette variable soit définie. Mais toutes les implémentations n'imposent pas cette contrainte. Par conséquent, en fonction de votre compilateur et de votre éditeur de liens, il est possible que vous puissiez transmettre parfaitement des données membres `static`

Copyright © 2016 Dunod.
© Dunod – Toute reproduction non autorisée est un délit.

`const` intégrales sans les avoir définies. Dans ce cas, félicitations, mais il y a de bonnes raisons de penser qu'un tel code n'est pas portable. Pour qu'il le devienne, il suffit de définir la donnée membre `static const` intégrale en question. Par exemple, pour `MinVals` :

```
const std::size_t Widget::MinVals;      // Dans le fichier .cpp de Widget.
```

Notez que la définition ne reprend pas l'initialiseur (28, dans le cas de `MinVals`), mais ne vous focalisez pas sur ce détail. En effet, si vous donnez l'initialiseur aux deux endroits, le compilateur vous préviendra et vous pourrez corriger votre code.

Noms de fonctions surchargées et noms de templates

Supposons que le comportement de notre fonction `f` (celle à laquelle nous transmettons des arguments *via* `fwd`) puisse être personnalisé en lui passant une fonction qui prend en charge une partie du travail. En imaginant que les paramètres et la valeur de retour de cette fonction soient des `int`, voici une déclaration de `f` :

```
void f(int (*pf)(int));          // pf = "processing function".
```

`f` pourrait également être déclarée avec une syntaxe plus simple, sans le pointeur. Cette déclaration, que voici, aurait le même sens que la précédente :

```
void f(int pf(int));             // Déclarer la même fonction f.
```

Quelle que soit la déclaration retenue, supposons à présent que nous ayons une fonction surchargée, `processVal` :

```
int processVal(int value);
int processVal(int value, int priority);
```

Nous pouvons passer `processVal` à `f` :

```
f(processVal);                   // Parfait.
```

Mais cela est un peu surprenant, car `f` attend un pointeur sur une fonction, alors que `processVal` n'est pas un pointeur sur une fonction, ni même une fonction, mais le nom de deux fonctions différentes. Cependant, le compilateur est capable de savoir quelle `processVal` utiliser : celle qui correspond au type du paramètre de `f`. Il choisit donc la version de `processVal` qui prend un seul `int` et passe l'adresse de cette fonction à `f`.

Cela fonctionne car la déclaration de `f` permet au compilateur de déterminer la version appropriée de `processVal`. En revanche, puisque `fwd` est un template de fonction, il ne dispose d'aucune information sur les types requis et il lui est impossible de déterminer la surcharge à employer :

Copyright © 2016 Dunod.

```
fwd(processVal);            // Erreur ! Quelle processVal ?
```

En soi, `processVal` n'a pas de type. Sans type il ne peut y avoir de déduction de type. Et sans déduction de type, nous avons un autre cas d'échec de la transmission parfaite.

Nous rencontrons le même problème si nous tentons d'utiliser un template de fonction à la place (ou en plus) d'un nom de fonction surchargée. Un template de fonction représente non pas une mais *plusieurs* fonctions :

```
template<typename T>
T workOnVal(T param)        // Template pour traiter les valeurs.
{ ... }

fwd(workOnVal);             // Erreur ! Quelle instanciation de
                            // workOnVal ?
```

Pour qu'une fonction de transmission parfaite comme `fwd` accepte un nom de fonction surchargée ou un nom de template, nous devons préciser manuellement la surcharge ou l'instanciation vers laquelle doit se faire la transmission. Par exemple, nous pouvons créer un pointeur de fonction du même type que le paramètre de `f`, initialiser ce pointeur avec `processVal` ou `workOnVal` (ce qui sélectionne la bonne version de `processVal` ou génère l'instanciation appropriée de `workOnVal`) et passer le pointeur à `fwd` :

```
using ProcessFuncType =                         // Faire un typedef ;
  int (*)(int);                                 // voir le conseil 9.

ProcessFuncType processValPtr = processVal;     // Spécifier la signature
                                                // requise pour processVal.

fwd(processValPtr);                             // Parfait.

fwd(static_cast<ProcessFuncType>(workOnVal));   // Également parfait.
```

Bien entendu, cette solution impose que nous connaissions le type du pointeur de fonction vers lequel `fwd` effectue sa transmission. On peut raisonnablement supposer qu'une fonction de transmission parfaite fournira cette information. En effet, ces fonctions sont conçues pour accepter *n'importe quoi* et si aucune documentation ne nous indique quoi passer, comment pourrions-nous le savoir ?

Copyright © 2016 Dunod.
© Dunod – Toute reproduction non autorisée est un délit.

Champs de bits

Le dernier cas d'échec de la transmission parfaite concerne l'utilisation d'un champ de bits en argument d'une fonction. Pour comprendre ce que cela signifie dans la pratique, examinons la représentation suivante d'un en-tête IPv4[1] :

```
struct IPv4Header {
  std::uint32_t version:4,
                IHL:4,
                DSCP:6,
                ECN:2,
                totalLength:16;
  ...
};
```

Si notre fonction `f` (l'éternelle cible de notre fonction de transmission `fwd`) est déclarée avec un paramètre de type `std::size_t`, son appel avec, par exemple, le champ `totalLength` d'un objet `IPv4Header` ne va causer aucun tracas au compilateur :

```
void f(std::size_t sz);        // Fonction à appeler.

IPv4Header h;
...
f(h.totalLength);              // Parfait.
```

En revanche, si nous essayons de transmettre `h.totalLength` à `f` par l'intermédiaire de `fwd`, le résultat est assez différent :

```
fwd(h.totalLength);            // Erreur !
```

Le problème vient du fait que le paramètre de `fwd` est une référence et que `h.totalLength` est un champ de bits non `const`. On pourrait penser que tout va bien, mais la norme C++ interdit cette combinaison de façon inhabituellement claire : « Une référence non `const` ne doit pas être liée à un champ de bits. » Il existe une très bonne raison à cette interdiction. En effet, les champs de bits peuvent être constitués de parties quelconques des mots machine (par exemple les bits 3 à 5 d'un `int` sur 32 bits), mais il n'existe aucune manière d'adresser directement de tels éléments. Nous avons mentionné précédemment que, au niveau matériel, les références et les pointeurs sont équivalents. Puisqu'il n'existe aucune manière de créer un pointeur sur des bits quelconques (C++ stipule que le `char` est le plus petit élément sur lequel nous pouvons pointer), il n'existe aucune solution pour lier une référence à des bits quelconques.

Il est très facile de contourner l'impossibilité de transmission parfaite d'un champ de bits, mais il faut tout d'abord savoir qu'une fonction qui accepte un champ de bits

1. Nous supposons que ces champs de bits commencent par le bit le moins significatif et se terminent par le bit le plus significatif. C++ n'offre pas cette garantie, mais le compilateur propose souvent un mécanisme qui permet aux programmeurs de fixer l'agencement d'un champ de bits.

Copyright © 2016 Dunod.

en argument reçoit une *copie* de la valeur du champ de bits. En effet, aucune fonction ne peut lier une référence à un champ de bits, pas plus qu'elle ne peut accepter des pointeurs sur des champs de bits puisque ceux-ci n'existent pas. Les champs de bits ne peuvent être passés qu'à des paramètres par valeur et, plus intéressant, à des références sur `const`. Dans le cas des paramètres par valeur, la fonction appelée reçoit évidemment une copie de la valeur d'un champ de bits. Dans le cas d'un paramètre de type référence sur `const`, la norme impose que la référence soit liée à une *copie* de la valeur du champ de bits stockée dans un objet de type intégral standard (par exemple un `int`). Les références sur `const` ne sont pas liées à des champs de bits mais à des objets « normaux » dans lesquels les valeurs des champs de bits ont été copiées.

Pour passer un champ de bits à une fonction de transmission parfaite, il suffit d'exploiter le fait que la fonction cible recevra toujours une copie de la valeur de ce champ. Nous pouvons effectuer la copie nous-mêmes et la passer à la fonction de transmission. Dans le cas de notre exemple avec `IPv4Header`, voici le code correspondant :

```
// Copier la valeur du champ de bits ; voir le conseil 6
// pour des infos sur l'initialisation.
auto length = static_cast<std::uint16_t>(h.totalLength);

fwd(length);                          // Transmettre la copie.
```

En résumé

Dans la plupart des cas, la transmission parfaite se passe exactement comme prévu. Il est inutile de s'en préoccuper. En revanche, lorsqu'elle échoue, c'est-à-dire lorsque du code semble-t-il convenable ne compile pas ou, pire, compile mais n'affiche pas le comportement attendu, il est important de connaître ses imperfections. Et il est tout aussi important de savoir comment les contourner. En général, la solution n'a rien de compliqué.

À retenir

- La transmission parfaite échoue lorsque la déduction de type de template échoue ou lorsque le type déduit est erroné.
- Les sortes d'arguments qui conduisent à l'échec de la transmission parfaite sont : les initialiseurs à accolades, les pointeurs nuls représentés par `0` ou `NULL`, les données membres `const static` intégrales uniquement déclarées, les noms de templates et de fonctions surchargées, et les champs de bits.

Copyright © 2016 Dunod.
© Dunod – Toute reproduction non autorisée est un délit.

6

Expressions lambda

En programmation C++, les *expressions lambda* changent la donne. Cela vous surprend sans doute, car elles n'apportent au langage aucune nouvelle puissance d'expression. Tout ce qui est possible avec une expression lambda l'est en procédant manuellement, avec un travail de saisie plus important. Mais les expressions lambda sont tellement commodes pour créer des objets fonctions que l'impact sur le développement quotidien en C++ est énorme. Sans les expressions lambda, les algorithmes « `_if` » de la STL (par exemple `std::find_if`, `std::remove_if`, `std::count_if`, etc.) sont souvent utilisés avec des prédicats très simples. En revanche, lorsque les expressions lambda sont disponibles, leurs emplois avec des conditions complexes fleurissent. Il en va de même avec les algorithmes personnalisables par des fonctions de comparaison (par exemple `std::sort`, `std::nth_element`, `std::lower_bound`, etc.). En dehors de la STL, les expressions lambda permettent de créer aisément des supprimeurs personnalisés pour `std::unique_ptr` et `std::shared_ptr` (voir les conseils 18 et 19). Elles facilitent également la spécification des prédicats pour les variables de condition dans l'API des threads (voir le conseil 39). Sorties de la bibliothèque standard, les expressions lambda simplifient la spécification à la volée de fonctions de rappel, de fonctions d'adaptation de l'interface et de fonctions contextuelles pour les appels exceptionnels. Grâce aux expressions lambda, C++ est un langage de programmation encore plus agréable.

Le jargon associé aux expressions lambda peut se révéler quelque peu perturbant. Voici quelques rappels :

- Une *expression lambda* n'est rien d'autre qu'une expression. Elle fait partie du code source. Dans le code suivant,

```
std::find_if(container.begin(),container.end(),
             [](int val) { return 0 < val && val < 10; });
```

  l'expression mise en exergue est l'expression lambda.

© Dunod – Toute reproduction non autorisée est un délit.
Copyright © 2016 Dunod.

- Une *fermeture* est l'objet d'exécution créé par une expression lambda. En fonction du mode de capture, les fermetures détiennent des références sur les données capturées, ou des copies de celles-ci. Dans l'appel précédent à `std::find_if`, la fermeture correspond à l'objet passé en troisième argument à `std::find_if` au moment de l'exécution.
- Une *classe de fermeture* est une classe à partir de laquelle une fermeture est instanciée. Pour chaque expression lambda, le compilateur génère une classe de fermeture unique. Les instructions de l'expression lambda deviennent des instructions exécutables dans les fonctions membres de la classe de fermeture.

En général, une expression lambda sert à créer une fermeture utilisée uniquement en argument d'une fonction. C'est le cas dans l'appel précédent à `std::find_if`. Cependant, les fermetures peuvent souvent être copiées et nous pouvons donc avoir plusieurs fermetures dont le type correspond à une seule expression lambda. En voici un exemple :

```
{
  int x;                                  // x est une variable locale.
  …

  auto c1 =                               // c1 est une copie de la
    [x](int y) { return x * y > 55; };    // fermeture produite par
                                          // l'expression lambda.

  auto c2 = c1;                           // c2 est une copie de c1.

  auto c3 = c2;                           // c3 est une copie de c2.

  …
}
```

`c1`, `c2` et `c3` sont des copies de la fermeture générée par l'expression lambda.

De façon informelle, nous pouvons parfaitement rendre un peu floue la séparation entre les expressions lambda, les fermetures et les classes de fermetures. Mais, dans les conseils qui suivent, il sera souvent important de distinguer ce qui existe au moment de la compilation (expressions lambda et classes de fermetures) et ce qui existe au moment de l'exécution (fermetures), ainsi que les relations entre chaque concept.

CONSEIL N° 31. ÉVITER LES MODES DE CAPTURE PAR DÉFAUT

Il existe deux modes de capture par défaut en C++11 : par référence et par valeur. La capture par référence par défaut peut conduire à des références dans le vide. La capture par valeur par défaut nous attire car nous pensons être immunisés contre ce problème (ce n'est pas le cas) et nous donne le faux sentiment que nos fermetures sont indépendantes (ce n'est pas forcément le cas).

Copyright © 2016 Dunod.

Voilà en substance la synthèse de ce conseil. Mais si votre penchant pour la technique ne se satisfait pas de ce résumé et exige des informations complémentaires, commençons par examiner les dangers de la capture par référence par défaut.

Avec une capture par référence, la fermeture contient une référence à une variable locale ou à un paramètre, dont la disponibilité correspond à la portée dans laquelle l'expression lambda est définie. Si la durée de vie de la fermeture créée à partir de cette expression lambda dépasse celle de la variable locale ou du paramètre, la référence présente dans la fermeture va pendouiller. Par exemple, supposons que nous disposions d'un conteneur de fonctions de filtrage, chacune prenant un `int` et retournant un `bool` qui indique si la valeur passée convient au filtre :

```
using FilterContainer =                    // Voir le conseil 9 pour
  std::vector<std::function<bool(int)>>;   // "using", le conseil 2
                                           // pour std::function.

FilterContainer filters;                   // Fonctions de filtrage.
```

Nous pouvons ajouter un filtre pour les multiples de 5 :

```
filters.emplace_back(                      // Voir le conseil 42
  [](int value) { return value % 5 == 0; } // pour des infos sur
);                                         // emplace_back.
```

En réalité, le calcul du diviseur doit se faire au moment de l'exécution et nous ne pouvons donc pas figer simplement la valeur 5 dans l'expression lambda. L'ajout du filtre se fait alors de la manière suivante :

```
void addDivisorFilter()
{
  auto calc1 = computeSomeValue1();
  auto calc2 = computeSomeValue2();

  auto divisor = computeDivisor(calc1, calc2);

  filters.emplace_back(                                  // Danger !
    [&](int value) { return value % divisor == 0; }     // La référence à
  );                                                    // divisor va
}                                                       // pendouiller !
```

Ce code cache un problème qui n'attend que de se révéler. L'expression lambda fait référence à la variable locale `divisor`, mais celle-ci cesse d'exister dès que la fonction `addDivisorFilter` est terminée. Cela se produit immédiatement après l'exécution de `filters.emplace_back` et la fonction ajoutée à `filters` est essentiellement tuée dans l'œuf.

Le même problème existerait si la capture par référence de `divisor` était explicite :

Copyright © 2016 Dunod.
© Dunod – Toute reproduction non autorisée est un délit.

```
filters.emplace_back(
  [&divisor](int value)                // Danger ! La référence
  { return value % divisor == 0; }     // à divisor va encore
);                                     // pendouiller !
```

Toutefois, avec une capture explicite, il est plus facile de constater que la viabilité de l'expression lambda dépend de la durée de vie de `divisor`. Par ailleurs, la saisie du nom « divisor » nous incite à vérifier que `divisor` a une durée de vie au moins aussi longue que les fermetures de l'expression lambda. Cet aide-mémoire est plus précis que l'avertissement « vérifiez que rien ne pointe dans le vide » communiqué par « `[&]` ».

Si nous savons qu'une fermeture sera employée immédiatement (par exemple en étant passée à un algorithme STL) et qu'elle ne sera pas copiée, les références qu'elle contient ne vivront pas plus longtemps que les variables locales et les paramètres de l'environnement dans lequel son expression lambda est créée. Dans ce cas, nous pourrions estimer que le risque de référence dans le vide n'existe pas et qu'il n'y a aucune raison d'éviter le mode de capture par référence par défaut. Par exemple, notre expression lambda de filtrage peut être utilisée uniquement en argument de l'algorithme `std::all_of` de C++11, qui indique si tous les éléments de la plage spécifiée satisfont à une condition :

```
template<typename C>
void workWithContainer(const C& container)
{
  auto calc1 = computeSomeValue1();              // Comme précédemment.
  auto calc2 = computeSomeValue2();              // Comme précédemment.

  auto divisor = computeDivisor(calc1, calc2);   // Comme précédemment.

  using ContElemT = typename C::value_type;      // Type des éléments
                                                 // dans le conteneur.

  using std::begin;                              // Pour la généricité ;
  using std::end;                                // voir le conseil 13.

  if (std::all_of(                               // Si toutes les valeurs
        begin(container), end(container),        // dans container sont
        [&](const ContElemT& value)              // des multiples du
        { return value % divisor == 0; })        // diviseur...
      ) {
    …                                            // C'est le cas...
  } else {
    …                                            // Au moins une ne
  }                                              // l'est pas...
}
```

Oui ce code est sûr, mais sa sécurité est quelque peu précaire. S'il s'avérait que l'expression lambda pouvait être utile dans d'autres contextes (par exemple en tant que fonction ajoutée au conteneur `filters`) et qu'elle était copiée-collée dans un contexte où sa fermeture survivrait à `divisor`, nous reviendrions au problème de

Copyright © 2016 Dunod.

référence dans le vide et rien dans la clause de capture ne nous rappellerait qu'une analyse de la durée de vie de `divisor` est nécessaire.

Les bonnes pratiques conseillent donc de recenser explicitement les variables locales et les paramètres dont dépend une expression lambda.

En C++14, nous pouvons employer `auto` dans les spécifications des paramètres des expressions lambda et ainsi simplifier le code précédent. Le typedef `ContElemT` est retiré et la condition du `if` est revue :

```
if (std::all_of(begin(container), end(container),
                [&](const auto& value)                // C++14.
                { return value % divisor == 0; }))
```

Pour résoudre notre problème lié à `divisor`, une solution pourrait être d'opter pour le mode de capture par valeur par défaut. Dans ce cas, l'ajout de l'expression lambda à `filters` se ferait de la manière suivante :

```
filters.emplace_back(                                 // À présent,
  [=](int value) { return value % divisor == 0; }     // divisor ne
);                                                    // peut pas
                                                      // pendouiller.
```

Si cela convient dans cet exemple, la capture par valeur par défaut n'est pas, de façon générale, le remède anti-pendouillement que nous pouvons imaginer. Le problème vient du fait que nous capturons un pointeur par valeur, le copions dans les fermetures issues de l'expression lambda, mais n'empêchons pas le code extérieur à l'expression lambda d'invoquer `delete` sur le pointeur et donc de faire que notre copie pendouille.

Vous pourriez prétendre que cela ne se produira jamais car, après avoir lu le chapitre 4, vous ne jurez plus que par les pointeurs intelligents et que seuls les mauvais programmeurs C++98 emploient des pointeurs bruts et `delete`. Sans doute mais peu importe car, en réalité, vous utilisez des pointeurs bruts, qui peuvent être passés à `delete` pour votre compte. C'est simplement qu'avec votre style moderne de programmation en C++, le code source ne le dévoile guère.

Supposons que les `Widget` puissent ajouter des entrées au conteneur de filtres :

```
class Widget {
public:
  …                                   // Constructeurs, etc.
  void addFilter() const;             // Ajouter une entrée aux filtres.

private:
  int divisor;                        // Utilisé dans le filter du Widget.
};
```

Voici comment nous pouvons définir `Widget::addFilter` :

Copyright © 2016 Dunod.
© Dunod – Toute reproduction non autorisée est un délit.

```
void Widget::addFilter() const
{
  filters.emplace_back(
    [=](int value) { return value % divisor == 0; }
  );
}
```

Pour le non-initié béat, ce code semble parfaitement sûr. L'expression lambda dépend bien de `divisor`, mais le mode de capture par valeur par défaut fait en sorte que `divisor` est copié dans tous les fermetures issues de cette expression lambda.

Faux, archi faux, horriblement faux, définitivement faux.

Les captures s'appliquent uniquement aux variables locales non `static` (y compris les paramètres) visibles dans la portée dans laquelle l'expression lambda est créée. Dans le corps de `Widget::addFilter`, `divisor` est non pas une variable locale mais une donnée membre de la classe `Widget` qui ne peut donc pas être capturée. Si nous retirons le mode de capture par défaut, le code ne compile toujours pas :

```
void Widget::addFilter() const
{
  filters.emplace_back(                                  // Erreur !
    [](int value) { return value % divisor == 0; }  // divisor n'est
  );                                                     // pas disponible.
}
```

Par ailleurs, si nous tentons de capturer explicitement `divisor` (que ce soit par valeur ou par référence), le code ne compile pas car `divisor` n'est pas une variable locale ni un paramètre :

```
void Widget::addFilter() const
{
  filters.emplace_back(
    [divisor](int value)                // Erreur ! Aucune variable locale
    { return value % divisor == 0; }    // divisor à capturer.
  );
}
```

Si la clause de capture par valeur par défaut ne capture pas `divisor`, et malgré l'absence de la clause de capture par valeur par défaut, pourquoi le code ne compile-t-il pas ?

L'explication tient dans l'utilisation implicite d'un pointeur brut : `this`. Toute fonction membre non `static` possède un pointeur `this`, utilisé chaque fois que nous mentionnons une donnée membre de la classe. Par exemple, dans n'importe quelle fonction membre de `Widget`, le compilateur remplace les utilisations de `divisor` par `this->divisor`. Examinons la version de `Widget::addFilter` avec une capture par valeur par défaut :

Copyright © 2016 Dunod.

```
void Widget::addFilter() const
{
  filters.emplace_back(
    [=](int value) { return value % divisor == 0; }
  );
}
```

Dans ce cas, la capture concerne non pas `divisor` mais le pointeur `this` de `Widget`. Le compilateur considère que le code est écrit ainsi :

```
void Widget::addFilter() const
{
  auto currentObjectPtr = this;

  filters.emplace_back(
    [currentObjectPtr](int value)
    { return value % currentObjectPtr->divisor == 0; }
  );
}
```

Si nous comprenons cela, nous comprenons que la viabilité des fermetures issues de cette expression lambda est liée à la durée de vie du `Widget` dont elles contiennent une copie du pointeur `this`. Examinons le code suivant qui, en accord avec le chapitre 4, utilise uniquement des pointeurs intelligents :

```
using FilterContainer =                    // Comme précédemment.
  std::vector<std::function<bool(int)>>;

FilterContainer filters;                   // Comme précédemment.

void doSomeWork()
{
  auto pw =                          // Créer un Widget ; voir
    std::make_unique<Widget>();      // le conseil 21 pour
                                     // std::make_unique.

  pw->addFilter();                   // Ajouter un filtre qui
                                     // utilise Widget::divisor.
  ...
}                                    // Détruire le Widget ; filters
                                     // contient un pointeur dans le vide !
```

Lors d'un appel à `doSomeWork`, nous créons un filtre qui dépend de l'objet `Widget` produit par `std::make_unique`, c'est-à-dire un filtre qui contient une copie d'un pointeur sur ce `Widget` ; il s'agit du pointeur `this` du `Widget`. Ce filtre est ajouté à `filters`, mais à la terminaison de `doSomeWork`, le `Widget` est détruit par le `std::unique_ptr` qui gère son cycle de vie (voir le conseil 18). À partir de ce moment-là, `filters` contient une entrée avec un pointeur dans le vide.

Pour résoudre, ce problème précis, il suffit d'effectuer une copie locale de la donnée membre à capturer et de capturer cette copie à la place :

Copyright © 2016 Dunod.
© Dunod – Toute reproduction non autorisée est un délit.

```
void Widget::addFilter() const
{
  auto divisorCopy = divisor;                    // Copier la donnée membre.

  filters.emplace_back(
    [divisorCopy](int value)                     // Capturer la copie.
    { return value % divisorCopy == 0; }         // Utiliser la copie.
  );
}
```

Pour être honnête, la capture par valeur par défaut fonctionne également avec cette approche :

```
void Widget::addFilter() const
{
  auto divisorCopy = divisor;                    // Copier la donnée membre.

  filters.emplace_back(
    [=](int value)                               // Capturer la copie.
    { return value % divisorCopy == 0; }         // Utiliser la copie.
  );
}
```

Mais pourquoi tenter le sort ? Avec un mode de capture par défaut, nous risquons de capturer `this` par mégarde, alors que nous pensions capturer `divisor`.

En C++14, pour capturer une donnée membre, une meilleure solution consiste à employer une capture généralisée (voir le conseil 32) :

```
void Widget::addFilter() const
{
  filters.emplace_back(                  // C++14 :
    [divisor = divisor](int value)       // Copier divisor dans la fermeture.
    { return value % divisor == 0; }     // Utiliser la copie.
  );
}
```

Puisque le mode de capture par défaut n'existe pas dans une capture généralisée, même en C++14, ce conseil 31 reste valable : il faut éviter les modes de capture par défaut.

Les captures par valeur par défaut ont un autre inconvénient : elles peuvent suggérer que les fermetures correspondantes sont indépendantes et non touchées par les modifications des données effectuées à l'extérieur des fermetures. En général, c'est faux, car des expressions lambda peuvent dépendre non seulement de variables locales et de paramètres (qui peuvent être capturés), mais également d'objets ayant une *durée de stockage statique*. De tels objets sont définis dans une portée globale ou un espace de noms, ou sont déclarés à l'intérieur de classes, de fonctions ou de fichiers. Ils peuvent être employés dans des expressions lambda, mais ils ne peuvent pas être capturés. Pourtant, la spécification d'un mode de capture par valeur par défaut peut donner l'impression qu'ils le sont. Étudions une version revue de la fonction `addDivisorFilter` précédente :

Copyright © 2016 Dunod.

```
void addDivisorFilter()
{
  static auto calc1 = computeSomeValue1();       // À présent static.
  static auto calc2 = computeSomeValue2();       // À présent static.

  static auto divisor =                          // À présent static.
    computeDivisor(calc1, calc2);

  filters.emplace_back(
    [=](int value)                     // Rien n'est capturé !
    { return value % divisor == 0; }   // Référence au static précédent.
  );

  ++divisor;                           // Modifier divisor.
}
```

Nous pourrions pardonner au lecteur occasionnel d'un tel code qui voit « `[=]` » et qui en déduit que l'expression lambda effectue une copie de tous les objets qu'elle utilise et qu'elle est donc indépendante. Mais c'est faux. Puisque cette expression lambda n'utilise aucune variable locale non statique, rien n'est capturé. À la place, le code de l'expression lambda fait référence à la variable `static divisor`. Lorsque, au terme de chaque invocation de `addDivisorFilter`, `divisor` est incrémenté, toutes les expressions lambda qui ont été ajoutées à `filters` *via* cette fonction auront un nouveau comportement (qui correspond à la nouvelle valeur de `divisor`). D'un point de vue pratique, cette expression lambda capture `divisor` par référence, en totale contradiction avec ce que la clause de capture par valeur par défaut semble impliquer. En nous tenons loin des clauses de capture par valeur par défaut, nous évitons tout risque de lecture erronée de notre code.

À retenir

- La capture par référence par défaut peut conduire à des références dans le vide.
- La capture par valeur par défaut peut conduire à des pointeurs dans le vide (en particulier `this`) et fait croire, à tort, que les expressions lambda sont indépendantes.

CONSEIL N° 32. UTILISER DES CAPTURES GÉNÉRALISÉES POUR DÉPLACER DES OBJETS DANS DES FERMETURES

Dans certains cas, la capture par valeur ou par référence ne convient pas. Si nous avons un objet réservé au déplacement (par exemple un `std::unique_ptr` ou un `std::future`) que nous souhaitons utiliser dans une fermeture, C++11 n'offre aucune solution. Si le coût de la copie d'un objet est élevé alors que celui de son déplacement est faible (par exemple la plupart des conteneurs de la bibliothèque standard), et si nous souhaitons utiliser cet objet dans une fermeture, il vaut mieux le déplacer que le copier. Mais C++11 n'apporte toujours aucune réponse à ce besoin.

Copyright © 2016 Dunod.
© Dunod – Toute reproduction non autorisée est un délit.

En C++14, la situation est différente. Cette version du langage prend directement en charge le déplacement des objets dans des fermetures. Si vous utilisez un compilateur C++14, réjouissez-vous et poursuivez votre lecture. Sinon, vous pouvez quand même vous réjouir et poursuivre votre lecture car des solutions en C++11 existent pour s'approcher de la capture par déplacement.

L'absence de la capture par déplacement a été reconnue comme une lacune même au moment de l'adoption de C++11. Le remède simple aurait été de l'ajouter dans C++14, mais le comité de normalisation a choisi une voie différente. Il a introduit un nouveau mécanisme de capture tellement souple que la capture par déplacement ne représente que l'une de ses possibilités. Cette nouvelle capacité est appelée *capture généralisée* (*init capture*). Elle offre essentiellement les mêmes possibilités que les différentes formes de capture de C++11, et plus encore. Toutefois, la seule exception concerne l'utilisation d'un mode de capture par défaut. Mais, cela n'a pas d'importance car le conseil 31 explique qu'il est préférable de les éviter. (Pour des situations équivalentes, la syntaxe de la capture généralisée est plus verbeuse que celle des captures C++11. Par conséquent, si une capture C++11 convient à l'opération, il ne faut pas hésiter à l'utiliser.)

Grâce à une capture généralisée, nous pouvons spécifier :

1. **le nom d'une donnée membre** dans la classe de fermeture générée à partir de l'expression lambda, et
2. **une expression** d'initialisation de cette donnée membre.

Voici comment employer une capture généralisée pour déplacer un `std::unique_ptr` dans une fermeture :

```
class Widget {                          // Un type utile.
public:
  …

  bool isValidated() const;
  bool isProcessed() const;
  bool isArchived() const;

private:
  …
};


auto pw = std::make_unique<Widget>();   // Créer un Widget ; voir le
                                        // conseil 21 pour des infos
                                        // sur std::make_unique.

…                                       // Configurer *pw.

auto func = [pw = std::move(pw)]              // Initialiser une donnée
            { return pw->isValidated()        // membre dans la fermeture
                     && pw->isArchived(); };  // avec std::move(pw).
```

Copyright © 2016 Dunod.

Le code mis en exergue correspond à la capture généralisée. Nous trouvons, à gauche du signe « = », le nom de la donnée membre dans la classe de fermeture spécifiée et, à sa droite, l'expression d'initialisation. Il faut savoir que les portées à gauche et à droite du signe « = » ne sont pas identiques. La portée à gauche correspond à celle de la classe de fermeture, tandis que la portée à droite correspond à celle de la définition de l'expression lambda. Dans l'exemple précédent, le nom `pw` à gauche du signe « = » fait référence à une donnée membre de la classe de fermeture, tandis que le nom `pw` à sa droite fait référence à l'objet déclaré avant l'expression lambda, c'est-à-dire la variable initialisée par l'appel à `std::make_unique`. Par conséquent, « `pw = std::move(pw)` » signifie « créer une donnée membre `pw` dans la fermeture et l'initialiser avec le résultat de l'invocation de `std::move` sur la variable locale `pw` ».

Le code présent dans le corps de l'expression lambda se trouve dans la portée de la classe de fermeture. Les utilisations de `pw` font donc référence à la donnée membre de la classe de fermeture.

Dans cet exemple, le commentaire « configurer `*pw` » indique que, après la création du `Widget` par `std::make_unique` et avant la capture du résultat de `std::unique_ptr` sur ce `Widget` par l'expression lambda, le `Widget` est modifié d'une façon ou d'une autre. Si cette configuration n'est pas nécessaire, autrement dit si le `Widget` produit par `std::make_unique` est dans un état qui autorise sa capture par l'expression lambda, la variable locale `pw` est inutile. En effet, il est alors possible d'initialiser directement la donnée membre de la classe de fermeture par `std::make_unique` :

```
auto func = [pw = std::make_unique<Widget>()]  // Initialiser une donnée
            { return pw->isValidated()         // membre dans la fermeture
                     && pw->isArchived(); };   // avec le résultat de
                                               // l'appel à make_unique.
```

Cela devrait vous convaincre que la notion de « capture » dans C++14 a été largement étendue car, en C++11, il est impossible de capturer le résultat d'une expression. Vous comprenez également pourquoi cette forme de capture se nomme *capture généralisée*.

Mais si notre compilateur ne prend pas en charge la capture généralisée de C++14, quelles solutions s'offrent à nous ? Comment pouvons-nous arriver à une capture de déplacement dans un langage qui n'en dispose pas ?

Il faut se rappeler qu'une expression lambda n'est qu'une manière de provoquer la génération d'une classe et la création d'un objet de cette classe. Tout ce qu'il est possible de faire avec une expression lambda, nous pouvons le faire manuellement. Voici comment écrire en C++11 l'exemple de code C++14 vu précédemment :

```
class IsValAndArch {                        // "est validé et
public:                                     // archivé".
  using DataType = std::unique_ptr<Widget>;

  explicit IsValAndArch(DataType&& ptr)     // Le conseil 25 explique
  : pw(std::move(ptr)) {}                   // l'utilisation de std::move.
```

Copyright © 2016 Dunod.
© Dunod – Toute reproduction non autorisée est un délit.

```
  bool operator()() const
  { return pw->isValidated() && pw->isArchived(); }

private:
  DataType pw;
};

auto func = IsValAndArch(std::make_unique<Widget>());
```

Le code est effectivement plus long qu'avec une expression lambda, mais cela ne change rien au fait que si nous voulons une classe C++11 qui prend en charge le déplacement-initialisation de ses données membres, le seul obstacle qui nous sépare de notre objectif est le temps passé à sa saisie.

Si nous préférons conserver les expressions lambda (en raison de leur commodité), la capture par déplacement peut être obtenue en C++11 de la façon suivante :

1. **déplacer l'objet à capturer dans un objet fonction généré avec** `std::bind`, et
2. **donner à l'expression lambda une référence à l'objet capturé.**

Si vous avez l'habitude de `std::bind`, le code vous semblera plutôt évident. Dans le cas contraire, il vous faudra un peu plus de temps pour le comprendre, mais cela en vaut la peine.

Supposons que nous voulions créer un `std::vector` local, y stocker un ensemble de valeurs, puis le déplacer dans une fermeture. Voici comment procéder en C++14 :

```
std::vector<double> data;              // Objet à déplacer dans
                                       // une fermeture.

…                                      // Remplir data.

auto func = [data = std::move(data)]   // Capture généralisée de C++14.
            { /* Utilisations de data. */ };
```

Les parties importantes du code sont mises en exergue : le type de l'objet à déplacer (`std::vector<double>`), le nom de cet objet (`data`) et l'expression d'initialisation pour la capture généralisée (`std::move(data)`). Voici la version C++11 équivalente, dans laquelle les mêmes parties essentielles sont repérées :

```
std::vector<double> data;                    // Comme précédemment.

…                                            // Comme précédemment.

auto func =
  std::bind(                                 // Simulation de la capture
    [](const std::vector<double>& data)      // généralisée en C++11.
    { /* Utilisations de data. */ },
    std::move(data)
  );
```

À l'instar des expressions lambda, `std::bind` génère des objets fonctions (foncteurs). Nous appelons les objets fonctions renvoyés par `std::bind` des *objets liaisons*.

Copyright © 2016 Dunod.

Le premier argument de `std::bind` est un objet invocable. Les arguments suivants représentent les valeurs qui seront passées à cet objet.

Un objet liaison contient des copies de tous les arguments transmis à `std::bind`. Pour chaque argument lvalue, l'objet correspondant dans l'objet liaison est construit par copie. Pour chaque rvalue, il est construit par déplacement. Dans notre exemple, puisque le second argument est une rvalue (le résultat de `std::move` ; voir le conseil 23), `data` est construit par déplacement dans l'objet liaison. Cette construction par déplacement constitue le cœur de la simulation de la capture par déplacement. En effet, nous contournons l'impossibilité de déplacer une rvalue dans une fermeture en C++11 par le déplacement d'une rvalue dans un objet liaison.

Lorsqu'un objet liaison est appelé (autrement dit son opérateur d'appel de fonction est invoqué), les arguments qu'il stocke sont passés à l'objet invocable indiqué à `std::bind`. Dans cet exemple, cela signifie que, lorsque `func` (l'objet liaison) est appelé, la copie de `data` construite par déplacement dans `func` est passée en argument à la fonction lambda donnée à `std::bind`.

Cette expression lambda est identique à celle utilisée en C++14, à l'exception du paramètre `data` que nous avons ajouté pour correspondre à notre objet faussement capturé par déplacement. Ce paramètre est une référence lvalue à la copie de `data` dans l'objet liaison. (Il n'est pas une référence rvalue, car, même si l'expression d'initialisation de la copie de `data`, « `std::move(data)` », est une rvalue, la copie de `data` est elle-même une lvalue.) Les manipulations de `data` à l'intérieur de l'expression lambda vont donc se faire sur la copie de `data` construite par déplacement dans l'objet liaison.

Par défaut, la fonction membre `operator()` de la classe de fermeture générée à partir d'une expression lambda est `const`. Par conséquent, toutes les données membres de la fermeture sont `const` à l'intérieur du corps de l'expression lambda. En revanche, la copie de `data` construite par déplacement dans l'objet liaison n'est pas `const`. Pour empêcher que cette copie de `data` ne soit modifiée dans l'expression lambda, le paramètre de celle-ci est déclaré comme une référence sur `const`. Si l'expression lambda était déclarée `mutable`, la fonction `operator()` dans sa classe de fermeture ne serait pas déclarée `const` et nous pourrions omettre `const` dans la déclaration du paramètre de l'expression lambda :

```
auto func =
  std::bind(                                  // Simulation en C++11 de la
    [](std::vector<double>& data) mutable     // capture généralisée pour
    { /* Utilisations de data. */ },          // des expressions lambda
    std::move(data)                           // mutables
  );
```

Puisqu'un objet liaison stocke des copies de tous les arguments transmis à `std::bind`, l'objet liaison de notre exemple contient une copie de la fermeture produite par l'expression lambda donnée dans son premier argument. La durée de vie de la fermeture est donc identique à celle de l'objet liaison. Ce point est important car il signifie que, tant que la fermeture existe, l'objet liaison qui contient l'objet faussement capturé par déplacement existe également.

Copyright © 2016 Dunod.
© Dunod – Toute reproduction non autorisée est un délit.

Si c'est la première fois que vous êtes confronté à `std::bind`, n'hésitez pas à consulter votre documentation C++11 préférée pour que tous les détails des explications précédentes se mettent en place. Dans tous les cas, les points fondamentaux suivants doivent être clairs :

- Il n'est pas possible de construire par déplacement un objet dans une fermeture C++11, mais il est possible de construire par déplacement un objet dans un objet liaison C++11.
- La simulation de la capture par déplacement en C++11 consiste à construire par déplacement un objet dans un objet liaison, puis à passer par référence l'objet construit par déplacement à l'expression lambda.
- Puisque la durée de vie de l'objet liaison est identique à celle de la fermeture, il est possible de considérer que les objets présents dans l'objet liaison se trouvent dans la fermeture.

Prenons un second exemple d'utilisation de `std::bind` pour simuler la capture par déplacement. Voici le code C++14 qui nous a servi précédemment à créer un `std::unique_ptr` dans une fermeture :

```
auto func = [pw = std::make_unique<Widget>()]       // Comme précédemment,
            { return pw->isValidated()              // créer pw dans une
                     && pw->isArchived(); };        // fermeture.
```

Voici son équivalent en C++11 :

```
auto func = std::bind(
              [](const std::unique_ptr<Widget>& pw)
              { return pw->isValidated()
                       && pw->isArchived(); },
              std::make_unique<Widget>()
            );
```

Vous êtes peut-être étonné que nous montrions comment utiliser `std::bind` pour contourner les limites des expressions lambda en C++11 alors que le conseil 34 préconise l'utilisation des expressions lambda à la place de `std::bind`. Toutefois, ce conseil explique que, en C++11, il existe certains cas où `std::bind` se révèle utile, et nous venons d'en décrire un. (Les fonctionnalités de C++14, comme la capture généralisée et les paramètres `auto` font disparaître ces cas.)

À retenir

- Utiliser la capture généralisée de C++14 pour déplacer des objets dans des fermetures.
- En C++11, simuler la capture généralisée à l'aide de classes écrites à la main ou avec `std::bind`.

Copyright © 2016 Dunod.

CONSEIL N° 33. UTILISER DECLTYPE SUR DES PARAMÈTRES AUTO&& POUR LES PASSER À STD::FORWARD

Les *expressions lambda génériques*, celles qui utilisent `auto` dans leurs spécifications des paramètres, font partie des fonctionnalités les plus attrayantes de C++14. Leur mise en œuvre est simple : la fonction `operator()` dans la classe de fermeture de l'expression lambda est un template. Prenons par exemple l'expression lambda suivante :

```
auto f = [](auto x){ return func(normalize(x)); };
```

Voici l'opérateur d'appel de fonction de la classe de fermeture :

```
class SomeCompilerGeneratedClassName {
public:
  template<typename T>                    // Voir le conseil 3 pour
  auto operator()(T x) const              // le type de retour auto.
  { return func(normalize(x)); }

  ...                                     // Autre fonctionnalité de
};                                        // la classe de fermeture.
```

Dans cet exemple, l'expression lambda se borne à transmettre son paramètre `x` à `normalize`. Si `normalize` traite différemment les lvalues et les rvalues, cette expression lambda est mal écrite, car elle passe toujours une lvalue (le paramètre `x`) à `normalize` même si l'argument qu'elle a reçu était une rvalue.

Pour que l'expression lambda soit correcte, elle doit transmettre parfaitement `x` à `normalize`. Pour cela, deux modifications du code sont nécessaires. Premièrement, `x` doit devenir une référence universelle (voir le conseil 24) et, deuxièmement, elle doit être passée à `normalize` *via* `std::forward` (voir le conseil 25). Conceptuellement, ces changements sont triviaux :

```
auto f = [](auto&& x)
         { return func(normalize(std::forward<???>(x))); };
```

Toutefois, pour passer du concept à la réalité, nous devons déterminer le type à indiquer à `std::forward`, c'est-à-dire trouver ce que nous devons mettre à la place de `???`.

Habituellement, nous utilisons la transmission parfaite dans le contexte d'une fonction template qui prend un paramètre de type `T` et nous écrivons donc `std::forward<T>`. Mais, dans l'expression lambda générique, aucun paramètre de type `T` n'est disponible. Il existe bien un `T` dans la fonction template `operator()` à l'intérieur de la classe de fermeture générée par cette expression lambda, mais il est impossible d'y faire référence à partir de l'expression.

Le conseil 28 explique que si un argument lvalue est passé à un paramètre qui est une référence universelle, le type de ce paramètre devient une référence lvalue. Et si

© Dunod – Toute reproduction non autorisée est un délit.
Copyright © 2016 Dunod.

une rvalue est passée, il devient une référence rvalue. Cela signifie que, dans notre expression lambda, nous pouvons examiner le type du paramètre `x` pour déterminer si l'argument passé était une lvalue ou une rvalue. `decltype` est là pour nous y aider (voir le conseil 3). Si une lvalue (rvalue) a été passée, `decltype(x)` génère un type qui correspond à une référence lvalue (rvalue).

Le conseil 28 explique également que, lors de l'appel à `std::forward`, les conventions veulent que l'argument de type soit une référence lvalue pour indiquer une lvalue et une non-référence pour indiquer une rvalue. Dans notre expression lambda, si `x` est lié à une lvalue, `decltype(x)` va produire une référence lvalue. Ce fonctionnement reste conforme aux conventions. En revanche, si `x` est lié à une rvalue, `decltype(x)` va donner une référence rvalue à la place de la non-référence d'usage.

Mais examinons l'exemple d'implémentation C++14 de `std::forward` tiré du conseil 28 :

```
template<typename T>                          // Dans l'espace de
T&& forward(remove_reference_t<T>& param)     // noms std.
{
  return static_cast<T&&>(param);
}
```

Si du code client souhaite transmettre parfaitement une rvalue de type `Widget`, il instancie normalement `std::forward` avec le type `Widget` (c'est-à-dire un type qui n'est pas une référence) et le template `std::forward` devient la fonction suivante :

```
Widget&& forward(Widget& param)           // Instanciation de
{                                         // std::forward lorsque
  return static_cast<Widget&&>(param);    // T est Widget.
}
```

Mais voyons ce qui se passe si le code client souhaite transmettre parfaitement la même rvalue de type `Widget` et si, au lieu de suivre la convention qui veut que `T` ne soit pas un type référence, il a spécifié une référence rvalue. Autrement dit, examinons le fonctionnement lorsque `T` est `Widget&&`. Après l'instanciation initiale de `std::forward` et l'application de `std::remove_reference_t`, mais avant la réduction de référence (à nouveau, voir le conseil 28), `std::forward` devient :

```
Widget&& && forward(Widget& param)          // Instanciation de
{                                           // std::forward lorsque
  return static_cast<Widget&& &&>(param);   // T est Widget&&
}                                           // (avant la réduction
                                            // de référence).
```

En appliquant la règle de réduction de référence, qui stipule qu'une référence rvalue à une référence rvalue devient une seule référence rvalue, nous obtenons l'instanciation suivante :

Copyright © 2016 Dunod.

```
Widget&& forward(Widget& param)                 // Instanciation de
{                                               // std::forward lorsque
  return static_cast<Widget&&>(param);          // T est Widget&&
}                                               // (après la réduction
                                                // de référence).)
```

Si nous comparons cette instanciation à celle qui résulte de l'appel de `std::forward` lorsque `T` est `Widget`, nous constatons qu'elles sont identiques. Autrement dit, instancier `std::forward` avec une référence rvalue donne le même résultat que son instanciation avec une non-référence.

Voilà une très bonne nouvelle, car `decltype(x)` conduit à un type référence rvalue lorsqu'une rvalue est passée au paramètre `x` de notre expression lambda. Nous avons établi précédemment que si une lvalue est passée à notre expression lambda, `decltype(x)` produit le type d'usage à passer à `std::forward`, et nous réalisons à présent que, pour les rvalues, `decltype(x)` donne un type non conventionnel pour `std::forward` mais que nous obtenons le même résultat qu'avec le type conventionnel. Par conséquent, que ce soit pour les lvalues ou pour les rvalues, passer `decltype(x)` à `std::forward` produit l'effet escompté. Notre expression lambda à transmission parfaite peut donc s'écrire de la manière suivante :

```
auto f =
  [](auto&& param)
  {
    return
      func(normalize(std::forward<decltype(param)>(param)));
  };
```

À partir de là, il est très facile d'obtenir une expression lambda à transmission parfaite qui accepte non pas un mais un nombre quelconque de paramètres, car C++14 reconnaît les expressions lambda variadiques :

```
auto f =
  [](auto&&... params)
  {
    return
    func(normalize(std::forward<decltype(params)>(params)...));
  };
```

À retenir

- Utiliser `decltype` sur les paramètres `auto&&` pour les passer à `std::forward`.

Copyright © 2016 Dunod.
© Dunod – Toute reproduction non autorisée est un délit.

CONSEIL N° 34. PRÉFÉRER LES EXPRESSIONS LAMBDA À STD::BIND

`std::bind` est le successeur C++11 des fonctions `std::bind1st` et `std::bind2nd` de C++98, mais, officieusement, cette fonction fait partie de la bibliothèque standard depuis 2005. C'était au moment où le comité de normalisation avait adopté un document nommé TR1, qui incluait la spécification de `bind`. (Dans TR1, `bind` se trouvait dans un espace de noms différent et était donc `std::tr1::bind`, à la place de `std::bind`, avec quelques différences dans les détails de l'interface.) Cette petite histoire signifie que certains programmeurs ont une dizaine d'années d'expérience avec `std::bind` et qu'ils ne sont peut-être pas enclins à abandonner un outil qui les a bien servis. Cela se comprend parfaitement, mais, dans ce cas, le changement est bénéfique car, en C++11, les expressions lambda sont quasiment toujours préférables à `std::bind`. En C++14, l'attirance pour les expressions lambda n'est pas simplement plus forte, elle est juste irrésistible.

Ce conseil suppose que vous maîtrisez `std::bind`. Dans le cas contraire, commencez par en apprendre au moins les bases. Cela vous sera de toute manière profitable, car vous aurez certainement l'occasion de rencontrer `std::bind` dans une base de code que vous devrez examiner ou maintenir.

Comme dans le conseil 32, nous allons appeler *objets liaisons* les objets fonctions retournés par `std::bind`.

Voici la principale raison de préférer les expressions lambda à `std::bind` : elles sont plus faciles à lire. Par exemple, supposons que nous ayons une fonction de configuration d'une alarme sonore :

```
// typedef pour un moment précis (voir le conseil 9 pour la syntaxe).
using Time = std::chrono::steady_clock::time_point;

// Voir le conseil 10 pour les "classes enum".
enum class Sound { Beep, Siren, Whistle };

// typedef pour une durée.
using Duration = std::chrono::steady_clock::duration;

// À l'instant t, produire un son s pendant une durée d.
void setAlarm(Time t, Sound s, Duration d);
```

Supposons également que, à un certain endroit du programme, nous ayons déterminé qu'une alarme doit être déclenchée une heure après qu'elle a été fixée, cela pour une durée de 30 secondes. En revanche, le son de l'alarme n'est pas indiqué. Nous pouvons écrire une expression lambda qui modifie l'interface de `setAlarm` afin que seul le son puisse être précisé :

```
// setSoundL ("L" pour "expression lambda") est un objet fonction qui
// permet de préciser un son pour une alarme de 30 secondes
// déclenchée une heure après qu'elle a été fixée.
```

Copyright © 2016 Dunod.

```
auto setSoundL =
  [](Sound s)
  {
    // Rendre les éléments de std::chrono disponibles,
    // sans qualification.
    using namespace std::chrono;

    setAlarm(steady_clock::now() + hours(1),     // Alarme à déclencher
             s,                                  // dans une heure
             seconds(30));                       // pendant 30 secondes.
  };
```

L'appel à `setAlarm` dans l'expression lambda est mis en exergue. Il s'agit d'un appel de fonction d'apparence normale et même le lecteur peu habitué aux expressions lambda voit que le paramètre `s` passé à l'expression lambda est transmis en argument à `setAlarm`.

En C++14, nous pouvons alléger ce code en profitant des suffixes standard pour les secondes (`s`), millisecondes (`ms`), heures (`h`), etc., qui se fondent sur la prise en charge par C++11 des littéraux définis par l'utilisateur. Ces suffixes étant disponibles dans l'espace de noms `std::literals`, nous révisons le code précédent:

```
auto setSoundL =
  [](Sound s)
  {
    using namespace std::chrono;
    using namespace std::literals;          // Pour les suffixes de C++14.

    setAlarm(steady_clock::now() + 1h,      // C++14, mais avec la
             s,                             // même signification
             30s);                          // que précédemment.
  };
```

Notre première tentative d'écriture de l'appel à `std::bind` correspondant est donné ci-après. Elle comporte une erreur, que nous corrigerons par la suite, mais le code juste est plus compliqué et cette version simplifiée soulève déjà des problèmes importants :

```
using namespace std::chrono;           // Comme précédemment.
using namespace std::literals;

using namespace std::placeholders;     // Nécessaire à l'utilisation
                                       // de "_1".

auto setSoundB =                       // "B" pour "bind".
  std::bind(setAlarm,
            steady_clock::now() + 1h,  // Incorrect ! Voir ci-après.
            _1,
            30s);
```

Nous aurions voulu mettre en exergue l'appel à `setAlarm` comme nous l'avons fait dans l'expression lambda, mais cet appel n'existe pas. Le lecteur de ce code doit simplement savoir qu'appeler `setSoundB` invoque `setAlarm` avec l'heure et la durée

Copyright © 2016 Dunod.
© Dunod – Toute reproduction non autorisée est un délit.

précisées dans l'appel à `std::bind`. Pour le non-initié, le paramètre fictif « `_1` » est, au fond, magique. Quant au lecteur informé, il devra relier mentalement le chiffre indiqué par ce paramètre fictif avec l'emplacement correspondant dans la liste des paramètres de `std::bind` afin de comprendre que le premier argument de l'appel à `setSoundB` est passé en second argument de `setAlarm`. Puisque le type de cet argument n'est pas identifié dans l'appel à `std::bind`, le lecteur devra consulter la déclaration de `setAlarm` pour connaître le type d'argument à passer à `setSoundB`.

Cependant, nous l'avons indiqué, le code n'est pas tout à fait juste. Dans l'expression lambda, il est clair que « `steady_clock::now() + 1h` » est un argument de `setAlarm`. Son évaluation se fera au moment de l'appel à `setAlarm`. Cela semble logique : nous voulons que l'alarme se déclenche une heure après l'invocation de `setAlarm`. Cependant, dans l'appel à `std::bind`, « `steady_clock::now() + 1h` » est passé en argument non pas à `setAlarm` mais à `std::bind`. Autrement dit, cette expression sera évaluée au moment de l'appel à `std::bind` et l'heure calculée par cette expression sera enregistrée dans l'objet liaison résultant. En conséquence, l'alarme sera déclenchée non pas une heure après l'appel à `setAlarm` mais *une heure après l'appel à `std::bind`* !

Pour résoudre ce problème, nous devons demander à `std::bind` de reporter l'évaluation de l'expression jusqu'à l'appel de `setAlarm`. Cela se fait en imbriquant un second appel à `std::bind` dans le premier :

```
auto setSoundB =
  std::bind(setAlarm,
            std::bind(std::plus<>(), steady_clock::now(), 1h),
            _1,
            30s);
```

Si vous avez l'habitude du template `std::plus` de C++98, vous êtes peut-être surpris de constater que, dans ce code, aucun type n'est spécifié entre les crochets obliques. Autrement dit, nous avons écrit « `std::plus<>` » à la place de « `std::plus<`*`type`*`>` ». En C++14, nous pouvons généralement omettre l'argument de type avec les templates d'opérateurs standard ; nous ne l'avons donc pas indiqué dans ce code. Puisqu'il en va autrement en C++11, l'appel à `std::bind` équivalent à l'expression lambda est le suivant :

```
using namespace std::chrono;                    // Comme précédemment.
using namespace std::placeholders;

auto setSoundB =
  std::bind(setAlarm,
            std::bind(std::plus<steady_clock::time_point>(),
                      steady_clock::now(),
                      hours(1)),
            _1,
            seconds(30));
```

Si, à ce stade, l'expression lambda ne vous semble pas plus attrayante, peut-être devriez-vous faire contrôler votre vue.

Copyright © 2016 Dunod.

Un nouveau problème survient si `setAlarm` est surchargée. Supposons qu'il existe une surcharge prenant un quatrième paramètre afin de préciser le volume de l'alarme :

```
enum class Volume { Normal, Loud, LoudPlusPlus };

void setAlarm(Time t, Sound s, Duration d, Volume v);
```

Avec l'expression lambda, nous obtenons le même fonctionnement que précédemment, car la résolution de la surcharge choisit la version à trois arguments de `setAlarm` :

```
auto setSoundL =                                    // Comme précédemment.
  [](Sound s)
  {
    using namespace std::chrono;

    setAlarm(steady_clock::now() + 1h,              // Parfait, appelle la
             s,                                     // version à 3 arguments
             30s);                                  // de setAlarm.
  };
```

En revanche, avec l'appel à `std::bind`, la compilation échoue :

```
auto setSoundB =                                    // Erreur ! Quelle
  std::bind(setAlarm,                               // variante de setAlarm ?
            std::bind(std::plus<>(),
                      steady_clock::now(),
                      1h),
            _1,
            30s);
```

En effet, le compilateur n'a aucun moyen de déterminer laquelle des deux fonctions `setAlarm` il doit passer à `std::bind`. Il ne dispose que d'un nom de fonction, qui, pris de façon isolée, est ambigu.

Pour que l'appel à `std::bind` puisse être compilé, nous devons convertir `setAlarm` dans le type de pointeur de fonction approprié :

```
using SetAlarm3ParamType = void(*)(Time t, Sound s, Duration d);

auto setSoundB =                                                // À présent
  std::bind(static_cast<SetAlarm3ParamType>(setAlarm),          // valide.
            std::bind(std::plus<>(),
                      steady_clock::now(),
                      1h),
            _1,
            30s);
```

Voilà qui amène une autre différence entre les expressions lambda et `std::bind`. Dans l'opérateur d'appel de fonction pour `setSoundL` (c'est-à-dire l'opérateur d'appel de fonction de la classe de fermeture de l'expression lambda), l'appel à `setAlarm`

Copyright © 2016 Dunod.
© Dunod – Toute reproduction non autorisée est un délit.

correspond à une invocation de fonction normale et le compilateur peut le convertir en code inline :

```
setSoundL(Sound::Siren);      // Le corps de setAlarm peut être
                              // mis "inline" ici.
```

En revanche, l'appel à `std::bind` passe un pointeur de fonction sur `setAlarm` et cela signifie que, dans l'opérateur d'appel de fonction pour `setSoundB` (c'est-à-dire l'opérateur d'appel de fonction pour l'objet liaison), l'appel à `setAlarm` se fait par l'intermédiaire d'un pointeur de fonction. Puisque les compilateurs sont moins enclins à convertir les appels *via* des pointeurs de fonctions en code inline, les appels à `setAlarm` au travers de `setSoundB` seront moins sujets à une conversion inline que ceux qui se font par `setSoundL` :

```
setSoundB(Sound::Siren);      // Le corps de setAlarm est moins
                              // sujet à une mise "inline" ici.
```

Les expressions lambda conduisent donc un code potentiellement plus rapide que celui obtenu avec `std::bind`.

L'exemple de `setAlarm` implique un seul appel de fonction simple. Si le code comprend des traitements plus complexes, la balance penche nettement en faveur des expressions lambda. Par exemple, examinons l'expression lambda C++14 suivante qui indique si son argument se trouve entre une valeur minimale (`lowVal`) et une valeur maximale (`highVal`), `lowVal` et `highVal` étant deux variables locales :

```
auto betweenL =
  [lowVal, highVal]
  (const auto& val)                          // C++14.
  { return lowVal <= val && val <= highVal; };
```

Nous pouvons implémenter le même fonctionnement avec `std::bind`, mais le code est un tantinet plus obscur :

```
using namespace std::placeholders;           // Comme précédemment.

auto betweenB =
  std::bind(std::logical_and<>(),            // C++14.
            std::bind(std::less_equal<>(), lowVal, _1),
            std::bind(std::less_equal<>(), _1, highVal));
```

En C++11, nous devons préciser les types à comparer. Voici l'appel à `std::bind` correspondant :

```
auto betweenB =                              // Version C++11.
  std::bind(std::logical_and<bool>(),
            std::bind(std::less_equal<int>(), lowVal, _1),
            std::bind(std::less_equal<int>(), _1, highVal));
```

Copyright © 2016 Dunod.

Bien entendu, en C++11, l'expression lambda ne peut pas prendre un paramètre `auto` et nous devons donc préciser un type :

```
auto betweenL =                                  // Version C++11.
  [lowVal, highVal]
  (int val)
  { return lowVal <= val && val <= highVal; };
```

Dans un cas comme dans l'autre, nous pensons que vous serez d'accord pour dire que la version avec l'expression lambda n'est pas simplement plus courte mais également plus compréhensible et facile à maintenir.

Nous avons indiqué précédemment que, pour les novices de `std::bind`, ses paramètres fictifs (par exemple `_1`, `_2`, etc.) ressemblaient énormément à de la magie. Cependant, le comportement opaque ne se limite pas aux paramètres fictifs. Supposons que nous ayons une fonction pour créer des copies compressées de `Widget` :

```
enum class CompLevel { Low, Normal, High };  // Niveau de compression.

Widget compress(const Widget& w,             // Créer une copie
                CompLevel lev);              // compressée de w.
```

Nous voulons créer un objet fonction qui permet d'indiquer le taux de compression d'un `Widget` `w` particulier. L'utilisation suivante de `std::bind` créera un tel objet :

```
Widget w;

using namespace std::placeholders;

auto compressRateB = std::bind(compress, w, _1);
```

Le `w` que nous passons à `std::bind` doit être mémorisé afin qu'il puisse être utilisé lors de l'appel ultérieur à `compress`. Il est stocké à l'intérieur de l'objet `compressRateB`, mais de quelle manière (par valeur ou par référence) ? La solution retenue fait une différence, car si `w` est modifié entre l'appel à `std::bind` et celui à `compressRateB`, son stockage par référence reflétera le changement, contrairement à son stockage par valeur.

La solution est donc de le stocker par valeur[1], mais la seule manière de le savoir est de se rappeler le fonctionnement de `std::bind` ; rien ne l'indique dans l'appel à `std::bind`. Comparons cela à l'utilisation d'une expression lambda où la capture par valeur ou par référence de `w` est explicite :

1. `std::bind` effectue toujours une copie de ses arguments, mais le code appelant peut obtenir un effet équivalent à un stockage par référence en appliquant `std::ref` à un argument. Avec le code suivant :

```
auto compressRateB = std::bind(compress, std::ref(w), _1);
```

`compressRateB` se comporte comme s'il détenait non pas une copie mais une référence à `w`.

© Dunod – Toute reproduction non autorisée est un délit.
Copyright © 2016 Dunod.

```
auto compressRateL =                          // w est capturée par
  [w](CompLevel lev)                          // valeur ; lev est
  { return compress(w, lev); };               // passé par valeur.
```

Le passage des paramètres à l'expression lambda est également explicite. Dans cet exemple, il est clair que le paramètre `lev` est passé par valeur :

```
compressRateL(CompLevel::High);               // L'argument est
                                              // passé par valeur.
```

Mais, dans l'appel à l'objet produit par `std::bind`, comment l'argument est-il passé ?

```
compressRateB(CompLevel::High);               // Comment l'argument
                                              // est-il passé ?
```

Une fois encore, la seule façon de le savoir est de se rappeler du fonctionnement de `std::bind`. (Tous les arguments passés aux objets liaisons le sont par référence, car l'opérateur d'appel de fonction de tels objets se fonde sur la transmission parfaite.)

En comparaison des expressions lambda, le code qui utilise `std::bind` est donc moins lisible, moins expressif et potentiellement moins performant. En C++14, il n'y a aucune bonne raison d'employer `std::bind`. En revanche, en C++11, `std::bind` peut se justifier dans deux situations spécifiques :

- **Capture par déplacement**. Les expressions lambda de C++11 ne proposent pas la capture par déplacement, mais il est possible de l'obtenir en combinant expression lambda et `std::bind`. Les détails de la mise en œuvre sont donnés dans le conseil 32, qui explique également que la prise en charge de la capture généralisée en C++14 ôte le besoin de cette implémentation.
- **Objets fonctions polymorphiques**. Puisque l'opérateur d'appel de fonction sur un objet liaison utilise la transmission parfaite, il peut accepter des arguments de n'importe quel type (à condition de respecter les contraintes de la transmission parfaite décrites au conseil 30). Cela peut se révéler utile si nous voulons lier un objet avec un template d'opérateur d'appel de fonction. Par exemple, prenons la classe suivante :

```
class PolyWidget {
public:
    template<typename T>
    void operator()(const T& param);
    …
};
```

Copyright © 2016 Dunod.

Voici comment lier un `PolyWidget` avec `std::bind` :

```
PolyWidget pw;

auto boundPW = std::bind(pw, _1);
```

L'appel de `boundPW` peut ensuite se faire avec différents types d'arguments :

```
boundPW(1930);              // Passer un int à
                            // PolyWidget::operator().

boundPW(nullptr);           // Passer nullptr à
                            // PolyWidget::operator().

boundPW("Rosebud");         // Passer une chaîne littérale à
                            // PolyWidget::operator()
```

Il n'existe aucune approche équivalente avec une expression lambda en C++11. En revanche, en C++14, il suffit d'utiliser une expression lambda avec un paramètre `auto` :

```
auto boundPW = [pw](const auto& param)    // C++14.
               { pw(param); };
```

Il s'agit évidemment de cas particuliers, qui ont de plus tendance à se raréfier, car les compilateurs qui prennent en charge les expressions lambda de C++14 sont de plus en plus courants.

Lorsque `bind` a été ajouté officieusement à C++ en 2005, il constituait une nette amélioration par rapport à ses prédécesseurs de 1998. L'arrivée des expressions lambda dans C++11 a rendu `std::bind` quelque peu obsolète. Depuis C++14, il n'existe plus aucune bonne raison de l'utiliser.

À retenir

- Le code qui utilise les expressions lambda est plus lisible, plus expressif et potentiellement plus performant que celui qui emploie `std::bind`.
- En C++11 uniquement, `std::bind` peut se révéler utile pour implémenter la capture par déplacement ou pour lier des objets à des templates d'opérateurs d'appel de fonction.

Copyright © 2016 Dunod.
© Dunod – Toute reproduction non autorisée est un délit.

7

L'API de concurrence

L'une des plus grandes avancées de C++11 réside dans l'incorporation de la concurrence au niveau du langage et de la bibliothèque. Les programmeurs habitués des autres API de gestion des threads (par exemple les pthreads ou les threads de Windows) sont parfois surpris de la rigidité des fonctionnalités de concurrence en C++, mais elle vient des contraintes imposées aux développeurs de compilateurs. Toutefois, grâce aux garanties qui en découlent au niveau du langage, les programmeurs peuvent, pour la première fois dans l'histoire de C++, écrire des applications multithreads au comportement cohérent sur toutes les plates-formes. Nous disposons ainsi d'une base solide pour construire des bibliothèques d'importance. Par ailleurs, les éléments pour la concurrence disponibles dans la bibliothèque standard (tâches, futurs, threads, mutex, variables de condition, objets atomiques, etc.) ne forment que le début de ce qui va devenir un riche jeu d'outils pour le développement de logiciels concurrents en C++.

Dans les conseils suivants, n'oubliez pas que la bibliothèque standard comprend deux templates pour les futurs : `std::future` et `std::shared_future`. Leurs différences ont souvent peu d'importance et nous parlons simplement de *futurs*, quelle que soit la variante.

CONSEIL N° 35. PRÉFÉRER LA PROGRAMMATION MULTITÂCHE PLUTÔT QUE MULTITHREAD

Si nous souhaitons exécuter une fonction `doAsyncWork` de façon asynchrone, deux possibilités s'offrent à nous. Nous pouvons créer un `std::thread` et lui confier l'exécution de `doAsyncWork`. Dans ce cas, nous utilisons une approche *basée sur les threads* :

Copyright © 2016 Dunod.
© Dunod – Toute reproduction non autorisée est un délit.

```
int doAsyncWork();

std::thread t(doAsyncWork);
```

Ou bien, nous pouvons passer `doAsyncWork` à `std::async`. Dans ce cas, la stratégie est *basée sur les tâches* :

```
auto fut = std::async(doAsyncWork);        // "fut" pour "futur".
```

Avec cette solution, l'objet de fonction transmis à `std::async` (par exemple `doAsyncWork`) est considérée comme une *tâche*.

De façon générale, l'approche multitâche est supérieure à son équivalent multithread, et le peu de code que nous venons de donner en illustre déjà certaines raisons. Dans notre exemple, `doAsyncWork` retourne une valeur et nous pouvons raisonnablement supposer qu'elle intéresse le code appelant. Dans le cas d'une invocation *via* un thread, il n'existe aucune manière simple d'accéder à cette valeur. En revanche, dans la solution basée sur une tâche, il est très facile de l'obtenir car le futur renvoyé par `std::async` dispose d'une fonction `get`. Celle-ci se révèle encore plus importante si `doAsyncWork` lève une exception, car `get` y donne également accès. Avec les threads, une exception lancée par `doAsyncWork` va conduire à la terminaison du programme (par un appel à `std::terminate`).

Mais il existe une différence plus fondamentale entre la programmation multithread et la programmation multitâche : le niveau d'abstraction apporté par les tâches est plus élevé, ce qui nous affranchit des détails de la gestion des threads. Cette observation nous amène à récapituler les trois sens donnés à « thread » dans un logiciel C++ concurrent :

- Les *threads matériels* correspondent aux threads qui effectuent réellement les calculs. Dans les architectures des ordinateurs modernes, chaque cœur du processeur offre un ou plusieurs threads matériels.
- Les *threads logiciels* (également appelés *threads système*) sont les threads que le système d'exploitation[1] gère sur l'ensemble de ses processus et dont il planifie l'exécution par des threads matériels. En général, il est possible de créer plus de threads logiciels qu'il n'existe de threads matériels, car, lorsqu'un thread logiciel est bloqué (par exemple sur une entrée-sortie ou l'attente d'un mutex ou d'une variable de condition), les performances peuvent être augmentées en exécutant d'autres threads non bloqués.
- Les *`std::thread`* sont des objets d'un processus C++ qui représentent les threads logiciels sous-jacents. Certains objets `std::thread` représentent des descripteurs (*handles*) « nuls », autrement dit ne correspondent à aucun thread logiciel, car ils ont été construits par défaut (ils n'ont pas de fonction à exécuter), ont été déplacés (le `std::thread` destinataire représente le thread logiciel sous-jacent), ont été bloqués avec `join` (la fonction qu'ils exécutaient s'est terminée) ou ont

1. À supposer qu'il y en ait un, car certains systèmes embarqués en sont dépourvus.

Copyright © 2016 Dunod.

été détachés avec `detach` (le lien entre eux et le thread logiciel sous-jacent a été coupé).

Les threads logiciels sont une ressource limitée. Si nous tentons d'en créer plus que ne peut en fournir le système, une exception `std::system_error` est lancée. C'est le cas même si la fonction à exécuter ne peut pas lever d'exception. Prenons l'exemple de `doAsyncWork` déclarée `noexcept` :

```
int doAsyncWork() noexcept;          // Voir le conseil 14 pour noexcept.
```

L'instruction suivante pourrait déclencher une exception :

```
std::thread t(doAsyncWork);          // Lance une exception s'il n'y a
                                     // plus de threads disponibles.
```

Un logiciel bien écrit doit tenir compte de cette possibilité, mais comment ? Une solution consiste à exécuter `doAsyncWork` sur le thread courant, mais elle risque de conduire à une mauvaise répartition de la charge et, si le thread courant est un thread de l'interface graphique, à des problèmes de réactivité. Une autre solution consiste à attendre que des threads logiciels existants arrivent au terme de leur exécution et de retenter la création d'un nouveau `std::thread`, mais il est possible que des threads existants soient en attente d'une action que `doAsyncWork` est censé réaliser (par exemple renvoyer un résultat ou notifier une variable de condition).

Même si nous ne sommes pas à court de threads, nous pouvons rencontrer des problèmes en raison d'une *demande excédentaire*. Cela se produit lorsque le nombre de threads logiciels prêts à s'exécuter (non bloqués) est supérieur au nombre de threads matériels. Dans ce cas, le gestionnaire des threads (qui fait en général partie du système d'exploitation) accorde des intervalles de temps matériel aux threads logiciels. Lorsque l'intervalle de temps d'un thread est terminé et qu'un autre débute, un changement de contexte est réalisé. Ces changements de contexte pèsent sur la gestion des threads par le système et peuvent se révéler particulièrement coûteux lorsque le thread matériel attribué à un thread logiciel se trouve sur un cœur différent de celui utilisé lors de l'intervalle de temps précédent accordé au même thread logiciel. Dans ce cas, (1) les caches du processeur pour ce thread logiciel sont en général invalides (ils contiennent peu de données et quelques instructions utiles) et (2) l'exécution du « nouveau » thread logiciel sur ce cœur pollue les caches du processeur associés aux « anciens » threads qui ont été exécutés sur ce cœur et qui risquent fort de l'être à nouveau lors de l'intervalle de temps suivant.

Il est difficile d'éviter la demande excédentaire, car le rapport optimal entre threads logiciels et threads matériels dépend de la fréquence à laquelle les threads logiciels deviennent exécutables. Cette fréquence évolue de façon dynamique, par exemple lorsqu'un programme passe d'une section qui sollicite énormément les entrées-sorties à une portion qui effectue de nombreux calculs. Le rapport optimal dépend également du coût des changements de contexte et de l'utilisation des caches du processeur par les threads logiciels. D'autre part, le nombre de threads matériels et les détails des caches

Copyright © 2016 Dunod.
© Dunod – Toute reproduction non autorisée est un délit.

du processeur (par exemple leur taille et leur rapidité) dépendent de l'architecture de la machine. Autrement dit, même si nous parvenons à régler notre application de façon à éviter les demandes excédentaires sur une plate-forme (tout en exploitant au mieux son matériel), rien ne garantit que notre solution sera efficace sur d'autres sortes de machines.

Notre vie serait beaucoup plus simple si nous pouvions confier la résolution de ces problèmes à quelqu'un d'autre. C'est exactement le rôle de `std::async` :

```
auto fut = std::async(doAsyncWork);     // La gestion des threads revient
                                        // au développeur de la
                                        // bibliothèque standard.
```

Cet appel place la responsabilité de la gestion des threads sur le dos de celui qui implémente la bibliothèque standard de C++. Par exemple, la probabilité de recevoir une exception en raison d'un manque de threads est considérablement réduite, car cet appel n'y conduira probablement jamais. Vous vous demandez certainement comment cela est possible. En effet, si nous demandons plus de threads logiciels que le système ne peut en fournir, quelle différence cela fait-il que ce soit en créant des `std::thread` ou en appelant `std::async` ? Cela compte car, avec un appel à `std::async` de cette forme (c'est-à-dire avec la stratégie de démarrage par défaut ; voir le conseil 36), rien ne garantit qu'un nouveau thread logiciel sera créé. En revanche, il permet au gestionnaire de s'organiser pour que la fonction indiquée (dans cet exemple `doAsyncWork`) soit exécutée sur le thread qui demande le résultat de `doAsyncWork` (c'est-à-dire sur le thread qui invoque `get` ou `wait` sur `fut`). N'importe quel planificateur digne de ce nom exploite cette liberté lorsque le système est trop sollicité ou qu'il est à court de threads.

Toutefois, nous avons indiqué précédemment qu'une « exécution sur le thread qui a besoin du résultat » peut conduire à des problèmes de répartition de charge et l'entrée en scène du gestionnaire des tâches et de `std::async` ne les fait pas disparaître pour autant. En ce qui concerne la répartition de charge, le gestionnaire aura cependant une vision plus complète que nous sur ce qui se passe sur la machine, car il ne gère pas uniquement les threads de notre code mais également ceux de tous les processus.

Avec `std::async`, les questions de réactivité liées à un thread de l'interface graphique persistent, car le planificateur ne peut pas savoir quels threads ont des exigences de réactivité fortes. Dans ce cas, nous pouvons passer la stratégie de démarrage `std::launch::async` à `std::async`. De cette manière, la fonction s'exécutera sur un thread réellement différent (voir le conseil 36).

Les gestionnaires de threads modernes exploitent des pools de threads de niveau système pour éviter les demandes excédentaires et améliorent la répartition de charge sur les cœurs matériels grâce à des algorithmes de vol de travail. La norme C++ n'impose pas l'utilisation des pools de threads ou du vol de travail et, pour être honnête, certains aspects techniques de la spécification de la concurrence en C++11 ne permettent pas de les employer aussi facilement que nous le souhaiterions. Néanmoins, certaines versions de la bibliothèque standard tirent profit de ces technologies et nous pouvons penser que les améliorations vont se poursuivre dans ce domaine.

Copyright © 2016 Dunod.

En basant notre programmation concurrente sur les tâches, nous pourrons bénéficier automatiquement des avantages de ces technologies lorsqu'elles deviendront plus répandues. En revanche, si nous manipulons directement des `std::thread`, nous devons nous occuper des problèmes de manque de threads, de demandes excédentaires et de répartition de la charge, sans compter que nos solutions doivent s'accorder à celles mises en œuvre dans les programmes qui s'exécutent sur d'autres processus de la même machine.

Au contraire d'une approche multithread, une conception multitâche nous évite la gestion manuelle des threads et fournit une voie naturelle pour examiner les résultats (valeurs de retour ou exceptions) des fonctions exécutées de façon asynchrone. Il existe néanmoins des cas où les threads peuvent se révéler appropriés :

- **Lorsque nous devons accéder à l'API de l'implémentation sous-jacente des threads**. L'API de concurrence de C++ est en général implémentée au-dessus d'une API de plus bas niveau spécifique à la plate-forme, comme les pthreads ou les threads de Windows. C++ n'offre pas toute la richesse de ces API. (Par exemple, les notions de priorité et d'affinité des threads n'existent pas en C++.) Pour que nous puissions avoir accès à l'API de l'implémentation sous-jacente des threads, les objets `std::thread` proposent habituellement la fonction membre `native_handle`. Il n'existe aucun équivalent à cette fonctionnalité avec les `std::future` (c'est-à-dire les objets retournés par `std::async`).
- **Lorsque nous devons et sommes en mesure d'optimiser l'utilisation des threads dans notre application**. Cela peut être le cas si, par exemple, nous développons un logiciel serveur dont le profil d'exécution est connu et qui constituera le seul processus important implanté sur une machine dont les caractéristiques matérielles sont figées.
- **Lorsque nous devons implémenter la technologie des threads en dehors de l'API de concurrence de C++**, par exemple pour proposer des pools de threads alors que l'implémentation de C++ sur la plate-forme ne les propose pas.

Cependant, ces cas sont plutôt rares. En général, une conception basée sur les tâches plutôt qu'une programmation avec des threads devra être privilégiée.

À retenir

- L'API de `std::thread` n'offre aucun accès direct aux valeurs de retour des fonctions qui s'exécutent de façon asynchrone et, si ces fonctions lancent des exceptions, le programme est terminé.
- La programmation basée sur les threads impose une gestion manuelle du manque de threads, de la demande excédentaire, de la répartition de charge et de l'adaptation aux nouvelles plates-formes.
- La programmation basée sur les tâches *via* `std::async` et la stratégie de démarrage par défaut nous affranchit de la quasi-totalité de ces problèmes.

Copyright © 2016 Dunod.
© Dunod – Toute reproduction non autorisée est un délit.

CONSEIL N° 36. SPÉCIFIER STD::LAUNCH::ASYNC SI L'ASYNCHRONISME EST PRIMORDIAL

Lorsque nous appelons `std::async` pour exécuter une fonction (ou tout autre objet invocable), nous avons généralement à l'idée une exécution asynchrone de cette fonction. Mais ce n'est pas forcément ce que nous demandons à `std::async`. En réalité, nous demandons que la fonction soit exécutée conformément à une *stratégie de démarrage*. Il existe deux stratégies standard, chacune représentée par un énumérateur dans l'`enum` délimité de `std::launch`. (Pour de plus amples informations sur les `enum` délimités, consulter le conseil 10.) Supposons qu'une fonction `f` soit passée à `std::async` en vue de son exécution :

- **Avec la stratégie de démarrage** `std::launch::async`, `f` doit être exécutée de façon asynchrone, c'est-à-dire sur un thread différent.
- **Avec la stratégie de démarrage** `std::launch::deferred`, `f` peut s'exécuter uniquement lorsque `get` ou `wait` est invoquée sur le futur renvoyé par `std::async`[1]. Autrement dit, l'exécution de `f` est *reportée* jusqu'à ce qu'un tel appel soit effectué. Lorsque `get` ou `wait` est invoquée, `f` est exécutée de façon synchrone, autrement dit le code appelant est bloqué jusqu'à la terminaison de `f`. Si ni `get` ni `wait` n'est appelée, `f` n'est jamais exécutée.

La stratégie de démarrage par défaut de `std::async`, celle utilisée si aucune autre n'est précisée, ne correspond à aucune des deux précédentes. En réalité, il s'agit de l'une ou de l'autre. Les deux appels suivants ont exactement la même signification :

```
auto fut1 = std::async(f);                          // Exécuter f en utilisant
                                                    // la stratégie de démarrage
                                                    // par défaut.

auto fut2 = std::async(std::launch::async |         // Exécuter f de façon
                       std::launch::deferred,       // asynchrone ou
                       f);                          // reportée.
```

La stratégie par défaut conduit donc à une exécution synchrone ou asynchrone de `f`. Comme le souligne le conseil 35, cette souplesse permet à `std::async` et aux composants de gestion des threads de la bibliothèque standard d'assumer la création et la destruction des threads, d'éviter la demande excédentaire et de gérer la répartition

1. Il s'agit d'une simplification. L'important n'est pas le futur sur lequel `get` ou `wait` est invoquée, mais l'état partagé auquel le futur fait référence. (Le conseil 38 traite des relations entre les futurs et les états partagés.) Puisque `std::future` prend en charge le déplacement et peut servir à construire un `std::shared_future`, et puisqu'un `std::shared_future` peut être copié, l'objet futur qui fait référence à l'état partagé issu de l'appel à `std::async` avec `f` est probablement différent de celui renvoyé par `std::async`. Cette explication étant quelque peu compliquée, il est fréquent de cacher la pleine vérité et de simplement parler de l'invocation de `get` ou de `wait` sur le futur retourné par `std::async`.

Copyright © 2016 Dunod.

de charge. C'est en partie grâce à tout cela que la programmation concurrente avec `std::async` est si commode.

L'utilisation de la stratégie de démarrage par défaut de `std::async` présente quelques conséquences intéressantes. Supposons qu'un thread `t` exécute l'instruction suivante :

```
auto fut = std::async(f);        // Exécuter f avec la stratégie
                                 // de démarrage par défaut.
```

- **Il est impossible de savoir si `f` s'exécutera de façon concurrente à `t`**, car l'exécution de `f` peut être reportée.
- **Il est impossible de savoir si `f` s'exécutera sur un thread différent de celui qui invoque `get` ou `wait` sur `fut`.** Si ce thread est `t`, nous ne pouvons pas savoir si `f` s'exécute sur un thread différent de `t`.
- **Il peut être impossible de savoir si `f` s'exécutera**, car il peut être impossible de garantir que `get` ou `wait` sera invoquée sur `fut` quel que soit le chemin emprunté par le programme.

La souplesse de planification de la stratégie de démarrage par défaut s'accorde souvent mal avec l'utilisation des variables `thread_local`, car, si `f` lit ou modifie une telle zone de *mémoire locale de thread* (TLS, *thread-local storage*), il n'est pas possible de désigner les variables du thread qui seront manipulées :

```
auto fut = std::async(f);        // TLS pour f potentiellement pour
                                 // un thread indépendent, mais
                                 // potentiellement pour le thread
                                 // qui invoque get ou wait sur fut.
```

Cela affecte également les boucles `wait` avec temporisation, car l'appel de `wait_for` ou de `wait_until` sur une tâche (voir le conseil 35) qui est reportée mène à la valeur `std::launch::deferred`. Autrement dit, la boucle suivante qui semble finir par s'arrêter, risque en réalité de s'exécuter indéfiniment :

```
using namespace std::literals;          // Pour les suffixes de durée
                                        // de C++14 ; voir le conseil 34.

void f()                                // f s'endort pendant 1 seconde,
{                                       // puis se réveille.
  std::this_thread::sleep_for(1s);
}

auto fut = std::async(f);               // Exécuter f de façon asynchrone
                                        // (conceptuellement).

while (fut.wait_for(100ms) !=           // Boucler jusqu'à ce que
       std::future_status::ready)       // f soit terminée... ce qui
{                                       // peut ne jamais arriver !
  ...
}
```

© Dunod – Toute reproduction non autorisée est un délit.
Copyright © 2016 Dunod.

Si `f` s'exécute de façon concurrente avec le thread qui appelle `std::async` (c'est-à-dire si la stratégie de démarrage choisie pour `f` est `std::launch::async`), ce code ne pose pas de problème (`f` se termine un jour). En revanche, si `f` est reportée, `fut.wait_for` retournera toujours `std::future_status::deferred`, qui ne sera jamais égal à `std::future_status::ready`, et la boucle ne s'arrêtera jamais.

Il est facile de passer à côté de ce genre de bogue pendant le développement et les tests unitaires, car il risque de se manifester uniquement en cas de charge importante. Cela correspond aux conditions qui mènent la machine vers une demande excédentaire ou un épuisement des threads, car la probablité de report d'une tâche est plus élevée. En effet, si le matériel n'est pas menacé de demande excédentaire ou d'épuisement des threads, il n'y a aucune raison que le gestionnaire d'exécution ne planifie pas une exécution concurrente de la tâche.

La correction du problème est simple : il suffit d'examiner le futur qui correspond à l'appel à `std::async` pour savoir si la tâche a été reportée et, dans l'affirmative, éviter d'entrer dans la boucle temporisée. Malheureusement, il n'existe aucune manière directe de demander à un futur si sa tâche a été reportée. À la place, nous devons appeler une fonction temporisée, comme `wait_for`, sans véritablement attendre quelque chose, mais simplement déterminer si la valeur de retour est `std::future_status::deferred`. Oublions l'incongruité du code et appelons `wait_for` avec une temporisation nulle :

```
auto fut = std::async(f);                    // Comme précédemment.

if (fut.wait_for(0s) ==                      // Si la tâche est
    std::future_status::deferred)            // reportée...
{
                              // ...invoquer wait ou get sur fut
  ...                         // pour appeler f de façon synchrone.

} else {                     // La tâche n'est pas reportée.
  while (fut.wait_for(100ms) !=              // Pas de boucle infinie
         std::future_status::ready) {        // (en supposant que f
                                             // se termine).

    ...                      // La tâche n'est ni reportée ni prête, par
                             // conséquent effectuer le travail concurrent
                             // en attendant.
  }

  ...                        // fut est prêt.

}
```

En résumé, ces différentes considérations impliquent que nous pouvons utiliser `std::async` avec la stratégie de démarrage par défaut tant que les conditions suivantes sont remplies :

- La tâche ne doit pas s'exécuter de façon concurrente avec le thread qui appelle `get` ou `wait`.

Copyright © 2016 Dunod.

- Les variables `thread_local` lues ou modifiées peuvent être celles de n'importe quel thread.
- `get` ou `wait` est assurée d'être invoquée sur le futur renvoyé par `std::async` ou l'exécution de la tâche n'est pas indispensable.
- Le code qui utilise `wait_for` ou `wait_until` tient compte du fait que la tâche puisse être reportée.

Lorsque l'une de ces conditions ne peut pas être satisfaite, il est sans doute préférable de garantir que `std::async` planifiera une exécution asynchrone de la tâche. Pour cela, il suffit de passer `std::launch::async` en premier argument lors de l'appel :

```
auto fut = std::async(std::launch::async, f);      // Démarrer f de
                                                   // façon asynchrone.
```

En réalité, il est assez pratique de disposer d'une fonction qui opère comme `std::async`, mais qui utilise automatiquement la stratégie de démarrage `std::launch::async`, d'autant qu'elle est facile à écrire. Voici sa version C++11 :

```
template<typename F, typename... Ts>
inline
std::future<typename std::result_of<F(Ts...)>::type>
reallyAsync(F&& f, Ts&&... params)          // Retourner le futur
{                                           // pour un appel asynchrone
  return std::async(std::launch::async,     // à f(params...).
                    std::forward<F>(f),
                    std::forward<Ts>(params)...);
}
```

Cette fonction reçoit un objet invocable `f` et aucun ou plusieurs paramètres `params`, qu'elle transmet parfaitement (voir le conseil 25) à `std::async`, en choisissant la stratégie de démarrage `std::launch::async`. À l'instar de `std::async`, elle renvoie un `std::future` qui résulte de l'appel de `f` avec `params`. Il est facile de déterminer le type de ce résultat, car le trait de type `std::result_of` nous le fournit. (Voir le conseil 9 pour des informations générales sur les traits de type.)

`reallyAsync` s'utilise exactement comme `std::async` :

```
auto fut = reallyAsync(f);          // Exécuter f de façon asynchrone ;
                                    // exception lancée si std::async
                                    // le peut.
```

En C++14, nous simplifions la déclaration de la fonction en profitant de la déduction du type de retour de `reallyAsync` :

```
template<typename F, typename... Ts>
inline
auto                                            // C++14.
reallyAsync(F&& f, Ts&&... params)
```

© Dunod – Toute reproduction non autorisée est un délit.
Copyright © 2016 Dunod.

```
{
  return std::async(std::launch::async,
                    std::forward<F>(f),
                    std::forward<Ts>(params)...);
}
```

Avec cette version, il apparaît clairement que `reallyAsync` ne fait rien d'autre qu'invoquer `std::async` avec la stratégie de démarrage `std::launch::async`.

À retenir

- La stratégie de démarrage par défaut de `std::async` autorise l'exécution synchrone ou asynchrone d'une tâche.
- Cette souplesse mène à une incertitude de fonctionnement lors des accès à des `thread_local`, implique que la tâche puisse ne jamais s'exécuter et affecte la logique du programme pour des appels temporisés à `wait`.
- Il faut utiliser `std::launch::async` si l'exécution asynchrone d'une tâche est primordiale.

CONSEIL N° 37. RENDRE LES STD::THREAD NON JOIGNABLES PAR TOUS LES CHEMINS

Chaque objet `std::thread` peut être dans l'état *joignable* ou *non joignable*. Un `std::thread` joignable correspond à un thread d'exécution asynchrone sous-jacent qui est ou peut être en cours d'exécution. Par exemple, un `std::thread` qui représente un thread sous-jacent bloqué ou en attente de sa reprise est joignable. Les objets `std::thread` qui correspondent à des threads dont l'exécution est arrivée à son terme sont également considérés joignables.

Un `std::thread` non joignable est évidemment un `std::thread` qui n'est pas joignable. Voici des cas d'objets `std::thread` non joignables :

- `std::thread` **construits par défaut**. Ces `std::thread` n'ont aucune fonction à exécuter et ne correspondent donc à aucun thread d'exécution sous-jacent.
- `std::thread` **qui ont été déplacés**. Suite à un déplacement, le thread d'exécution sous-jacent auquel correspondait un `std::thread` (si c'était le cas) est désormais associé à un `std::thread` différent.
- `std::thread` **qui ont été joints**. Après un appel à `join`, l'objet `std::thread` ne correspond plus au thread d'exécution sous-jacent qui est terminé.
- `std::thread` **qui ont été détachés**. Un appel à `detach` coupe le lien entre un objet `std::thread` et le thread d'exécution sous-jacent qu'il représente.

Le fait qu'un `std::thread` soit joignable est important car, si son destructeur est invoqué, l'exécution du programme se termine. Par exemple, supposons que nous ayons une fonction `doWork` qui prend en paramètres une fonction de filtrage, `filter`, et une valeur maximale, `maxVal`. `doWork` vérifie que toutes les conditions préalables à

Copyright © 2016 Dunod.

son action sont remplies, puis elle effectue son traitement avec toutes les valeurs entre zéro et `maxVal` qui traversent le filtre. Si le filtrage et la vérification des conditions sont deux opérations qui prennent beaucoup de temps, nous pouvons envisager de les effectuer de façon concurrente.

Pour cela, notre préférence irait vers une conception multitâche (voir le conseil 35), mais supposons que nous voulions fixer la priorité du thread qui se charge du filtrage. Le conseil 35 explique que cette opération doit se faire au travers du descripteur natif du thread, mais il n'est accessible qu'au travers de l'API de `std::thread`. L'API des tâches (c'est-à-dire des futurs) ne permet pas d'y accéder. C'est pourquoi notre solution se fonde non pas sur les tâches mais les threads. Voici le code correspondant :

```
constexpr auto tenMillion = 10000000;              // Voir le conseil 15
                                                   // pour constexpr.

bool doWork(std::function<bool(int)> filter,       // Indique si le calcul
            int maxVal = tenMillion)               // a été réalisé ;
{                                                  // voir le conseil 5
                                                   // pour std::function.

  std::vector<int> goodVals;                       // Valeurs conformes
                                                   // au filtre.

  std::thread t([&filter, maxVal, &goodVals]       // Remplir goodVals.
                {
                  for (auto i = 0; i <= maxVal; ++i)
                   { if (filter(i)) goodVals.push_back(i); }
                });

  auto nh = t.native_handle();                     // Utiliser le descripteur
  …                                                // natif de t pour fixer
                                                   // la priorité de t.
  if (conditionsAreSatisfied()) {
    t.join();                                      // Laisser t se terminer.
    performComputation(goodVals);
    return true;                                   // Le calcul a été fait.
  }

  return false;                                    // Le calcul n'a pas été fait.
}
```

Avant que nous n'expliquions pourquoi ce code pose problème, remarquons que la valeur d'initialisation de `tenMillion` pourrait être plus lisible si nous profitions de la possibilité que nous offre C++14 de séparer les milliers par une apostrophe :

```
constexpr auto tenMillion = 10'000'000;          // C++14.
```

Remarquons également que fixer la priorité de `t` après que son exécution a débuté, c'est un peu songer à fermer la cage quand les oiseaux se sont envolés. Une meilleure approche serait de démarrer `t` dans l'état suspendu (nous permettant alors de régler sa priorité avant qu'il ne commence ses calculs), mais ne nous laissons pas distraire

Copyright © 2016 Dunod.
© Dunod – Toute reproduction non autorisée est un délit.

par un tel code. S'il vous intéresse, vous pouvez consulter le conseil 39 qui explique comment créer des thread suspendus.

Revenons à `doWork`. Si *`conditionsAreSatisfied()`* renvoie `true`, tout va bien. En revanche, si la fonction renvoie `false` ou lève une exception, l'objet `std::thread t` sera joignable lorsque son destructeur est invoqué à la fin de `doWork`. Cela déclenchera la terminaison de l'exécution du programme.

Le destructeur de `std::thread` se comporte ainsi car les deux autres options évidentes sont pires encore :

- `join` **implicite**. Dans ce cas, le destructeur d'un `std::thread` attendrait que le thread d'exécution asynchrone sous-jacent se termine. Cela peut paraître acceptable, mais il serait difficile de localiser l'origine des problèmes de performance qui pourraient en résulter. Par exemple, il ne serait pas évident de comprendre que `doWork` attend que son filtre soit appliqué à toutes les valeurs si *`conditionsAreSatisfied()`* a déjà renvoyé `false`.
- `detach` **implicite**. Dans ce cas, le destructeur d'un `std::thread` couperait le lien entre l'objet `std::thread` et son thread d'exécution sous-jacent, mais ce dernier ne s'arrête pas pour autant. Cela ne semble pas moins raisonnable que l'option `join`, mais les problèmes de débogage induits risquent d'être pires. Par exemple, dans `doWork`, `goodVals` est une variable locale capturée par référence. Elle est également modifiée dans l'expression lambda (*via* l'appel à `push_back`). Supposons que pendant l'exécution asynchrone de l'expression lambda, *`conditionsAreSatisfied()`* retourne `false`. Dans ce cas, `doWork` se termine et ses variables locales (y compris `goodVals`) sont détruites. Sa trame de pile est effacée et l'exécution du thread se poursuit au point d'appel `doWork`. Les instructions qui suivent ce point vont appeler d'autres fonctions et au moins l'une d'elles va occuper une partie ou toute la zone de mémoire dans laquelle se trouvait la trame de pile de `doWork`. Appelons cette fonction `f`. Pendant que `f` s'exécute, l'expression lambda qui avait été initiée par `doWork` poursuit son exécution asynchrone. Elle peut appeler `push_back` sur la zone de mémoire précédemment occupée dans la pile par `goodVals`, mais qui se trouve à présent quelque part dans la trame de `f`. Cet appel va donc modifier la mémoire qui était utilisée par `goodVals` et, du point de vue de `f`, le contenu de sa trame de pile peut être modifié à n'importe quel moment ! Imaginez tout le plaisir que procurera le débogage d'un tel comportement.

Le comité de normalisation a décidé que les conséquences de la destruction d'un thread joignable étaient suffisamment terribles pour qu'elle soit interdite (en spécifiant que la destruction d'un thread joignable provoque la terminaison du programme).

En raison de ce choix, c'est à nous de faire en sorte que tout objet `std::thread` que nous utilisons soit rendu non joignable dans tous les chemins qui se trouvent en dehors de la portée de sa définition. Cependant, il peut être très compliqué de traiter tous les chemins, car cela inclut le flux jusqu'à la fin de la portée ainsi que les sorties directes avec `return`, `continue`, `break`, `goto` ou une exception. Les chemins peuvent être très nombreux.

Copyright © 2016 Dunod.

Chaque fois que nous devons effectuer une action sur tout chemin qui sort d'un bloc, l'approche classique consiste à placer cette action dans le destructeur d'un objet local. De tels objets sont appelés *objets RAII*, et les classes correspondantes sont appelées *classes RAII*. (*RAII* est l'acronyme de *Resource Acquisition Is Initialization*, ou l'« acquisition d'une ressource est une initialisation », même si la technique se fonde non pas sur une initialisation mais une destruction.) La bibliothèque standard regorge de classes RAII. Il s'agit notamment des conteneurs STL (le destructeur de chaque conteneur détruit le contenu du conteneur et libère sa mémoire), des pointeurs intelligents standard (les conseils 18 à 20 expliquent que le destructeur de `std::unique_ptr` invoque son supprimeur sur l'objet pointé et que les destructeurs de `std::shared_ptr` et de `std::weak_ptr` décrémentent les compteurs de références), des objets `std::fstream` (les destructeurs ferment les fichiers associés), etc. Pourtant, il n'existe aucune classe RAII standard pour les objets `std::thread`, probablement parce que le comité de normalisation, en ayant écarté `join` et `detach` des options par défaut, ne savait tout simplement pas ce qu'une telle classe devait faire.

Heureusement, l'écriture de cette classe ne pose aucune difficulté. Par exemple, la classe suivante permet au code appelant de préciser si `join` ou `detach` doit être appelée lorsqu'un objet `ThreadRAII` (un objet RAII pour un `std::thread`) est détruit :

```
class ThreadRAII {
public:
  enum class DtorAction { join, detach };     // Voir le conseil 10 pour
                                              // des infos sur enum class.

  ThreadRAII(std::thread&& t, DtorAction a)   // Dans le destructeur,
  : action(a), t(std::move(t)) {}             // effectuer l'action a sur t.

  ~ThreadRAII()
  {                                           // Voir ci-après pour le
    if (t.joignable()) {                      // test de joignabilité.

      if (action == DtorAction::join) {
        t.join();
      } else {
        t.detach();
      }
    }
  }

  std::thread& get() { return t; }            // Voir ci-après.

private:
  DtorAction action;
  std::thread t;
};
```

Nous pensons que ce code est facile à comprendre de lui-même, mais les points suivants pourraient aider :

© Dunod – Toute reproduction non autorisée est un délit.
Copyright © 2016 Dunod.

- Le constructeur accepte uniquement des `std::thread` rvalues, car nous voulons déplacer dans l'objet `ThreadRAII` le `std::thread` passé. (Rappelons que les objets `std::thread` ne sont pas copiables.)
- L'ordre choisi pour les paramètres du constructeur est intuitif (préciser tout d'abord le `std::thread`, puis l'action du destructeur, est plus sensé que l'inverse), mais la liste d'initialisation des membres correspond à l'ordre de déclaration des données membres qui place l'objet `std::thread` en dernier. Dans cette classe, l'ordre n'a pas d'importance mais, de façon générale, il est possible que l'initialisation d'une donnée membre dépende d'une autre et, puisque les objets `std::thread` peuvent démarrer l'exécution d'une fonction immédiatement après leur initialisation, il est préférable que leur déclaration arrive en dernier dans une classe. De cette manière, au moment de leur construction, toutes les données membres qui les précèdent auront déjà été initialisées et pourront donc être manipulées en toute sécurité par le thread asynchrone qui correspond à la donnée membre `std::thread`.
- `ThreadRAII` fournit une fonction `get` qui donne accès à l'objet `std::thread` sous-jacent. Cela équivaut aux fonctions `get` des pointeurs intelligents standard qui donnent accès aux pointeurs bruts sous-jacents. Grâce à cette fonction, `ThreadRAII` n'a pas besoin de reproduire l'intégralité de l'interface de `std::thread` et les objets `ThreadRAII` peuvent être employés dans des contextes qui requièrent des objets `std::thread`.
- Avant que le destructeur de `ThreadRAII` n'invoque une fonction membre sur l'objet `std::thread` `t`, il s'assure que `t` est joignable. Ce contrôle est obligatoire car l'invocation de `join` ou de `detach` sur un thread non joignable conduit à un comportement indéfini. Il est possible qu'un client ait construit un `std::thread`, l'ait utilisé pour créer un `ThreadRAII`, ait invoqué `get` pour obtenir un accès à `t`, et ait effectué un déplacement à partir de `t` ou ait appelé `join` ou `detach` sur `t`. Chacune de ces actions rend `t` non joignable.
  Examinons la portion de code suivante :

```
if (t.joignable()) {

  if (action == DtorAction::join) {
    t.join();
  } else {
    t.detach();
  }
}
```

Si vous pensez qu'il cache un cas de concurrence, car, entre l'exécution de `t.joignable()` et l'invocation de `join` ou de `detach`, un autre thread pourrait rendre `t` non joignable, votre intuition est louable mais vos craintes sont infondées. En effet, un objet `std::thread` ne peut passer de l'état joignable à l'état non joignable uniquement au travers d'un appel de fonction, comme `join`, `detach` ou une opération de déplacement. Au moment où le destructeur d'un objet `ThreadRAII` est invoqué, aucun autre thread ne peut invoquer une

Copyright © 2016 Dunod.

fonction membre de cet objet. En cas d'appels simultanés, il existe bien une condition de concurrence, mais elle se trouve non pas dans le destructeur mais dans le code client qui tente d'invoquer en même temps deux fonctions membres (le destructeur et une autre fonction) sur un même objet. De façon générale, les appels simultanés à des fonctions membres sur un même objet sont sûrs uniquement si ces fonctions sont toutes `const` (voir le conseil 16).

Voici comment nous utilisons `ThreadRAII` dans notre exemple `doWork` :

```
bool doWork(std::function<bool(int)> filter,   // Comme précédemment.
            int maxVal = tenMillion)
{
  std::vector<int> goodVals;                   // Comme précédemment.

  ThreadRAII t(                                // Utiliser un objet RAII.
    std::thread([&filter, maxVal, &goodVals]
                {
                  for (auto i = 0; i <= maxVal; ++i)
                    { if (filter(i)) goodVals.push_back(i); }
                }),
                ThreadRAII::DtorAction::join   // Action RAII.
  );

  auto nh = t.get().native_handle();
  ...

  if (conditionsAreSatisfied()) {
    t.get().join();
    performComputation(goodVals);
    return true;
  }

  return false;
}
```

Dans le destructeur de `ThreadRAII`, nous avons choisi d'effectuer un `join` sur le thread qui s'exécute de façon asynchrone car, comme nous l'avons vu précédemment, un `detach` pourrait conduire à des problèmes de débogage cauchemardesques. Cependant, nous avons également indiqué qu'un `join` pourrait soulever des problèmes de performances, qui, pour être francs, seraient également difficiles à déboguer. Entre un comportement indéfini (obtenu par `detach`), la terminaison du programme (obtenue par l'utilisation d'un `std::thread`) et des soucis de performances, ces derniers nous semblent le moins mauvais choix.

Hélas, le conseil 39 montre que l'utilisation de `ThreadRAII` pour effectuer un `join` lors de la destruction d'un `std::thread` peut non seulement conduire à des problèmes de performances, mais également au blocage du programme. La solution « appropriée » à ce type de problème serait d'indiquer à l'expression lambda, qui s'exécute de façon asynchrone, que nous n'avons plus besoin d'elle et qu'elle doit se terminer au plus tôt.

Copyright © 2016 Dunod.
© Dunod – Toute reproduction non autorisée est un délit.

Malheureusement, C++11 ne reconnaît pas les *threads interruptibles*. Il est possible de les implémenter manuellement, mais ce travail sort du cadre de cet ouvrage[1].

Le conseil 17 explique que, en raison de la déclaration d'un destructeur dans `ThreadRAII`, le compilateur ne générera aucune opération de déplacement. Cependant, rien n'empêche d'avoir des objets `ThreadRAII` déplaçables. Si le compilateur générait ces fonctions, elles auraient le comportement approprié. Par conséquent, nous pouvons demander explicitement leur création :

```
class ThreadRAII {
public:
  enum class DtorAction { join, detach };           // Comme précédemment.

  ThreadRAII(std::thread&& t, DtorAction a)         // Comme précédemment.
  : action(a), t(std::move(t)) {}

  ~ThreadRAII()
  {
    ...                                             // Comme précédemment.
  }

  ThreadRAII(ThreadRAII&&) = default;               // Support du
  ThreadRAII& operator=(ThreadRAII&&) = default;    // déplacement.

  std::thread& get() { return t; }                  // Comme précédemment.

private:                                            // Comme précédemment.
  DtorAction action;
  std::thread t;
};
```

À retenir

- Rendre les `std::thread` non joignables sur tous les chemins.
- Appeler `join` lors de la destruction peut conduire à des problèmes de performances difficiles à déboguer.
- Appeler `detach` lors de la destruction peut conduire à un comportement indéfini difficile à déboguer.
- Déclarer les objets `std::thread` en dernier dans la liste des membres.

1. Il est réalisé de manière très intéressante dans la section 9.2 de l'ouvrage C++ *Concurrency in Action* par Anthony Williams (Manning Publications, 2012).

Copyright © 2016 Dunod.

CONSEIL N° 38. ÊTRE CONSCIENT DU COMPORTEMENT VARIABLE DU DESTRUCTEUR DU DESCRIPTEUR DE THREAD

Le conseil 37 explique qu'un `std::thread` joignable correspond à un thread d'exécution système sous-jacent. Le futur d'une tâche non reportée (voir le conseil 36) possède une relation comparable avec un thread système. Nous pouvons donc considérer les objets `std::thread` et les objets futurs comme des *descripteurs* (*handles*) de threads système.

De ce point de vue, il est intéressant que les destructeurs des `std::thread` et des futurs aient des comportements aussi différents. Comme nous l'avons noté au conseil 37, la destruction d'un `std::thread` joignable met fin au programme, car les deux autres possibilités, `join` implicite et `detach` implicite, semblaient pires encore. Pourtant, le comportement du destructeur d'un futur semble correspondre parfois à un `join` implicite, parfois à un `detach` implicite, et parfois à aucun des deux. Il ne provoque jamais la terminaison du programme. Cette diversité de comportement du descripteur d'un thread mérite un examen plus approfondi.

Observons tout d'abord qu'un futur constitue l'une des extrémités d'un canal de communication au travers duquel l'appelé transmet un résultat à l'appelant[1]. L'appelé, qui s'exécute habituellement de façon asynchrone, écrit le résultat de son traitement sur le canal de communication, en général au travers d'un objet `std::promise`, et l'appelant lit ce résultat en utilisant un futur. La figure 7.1 illustre ce fonctionnement, la flèche en pointillés représentant le flux des informations de l'appelé vers l'appelant.

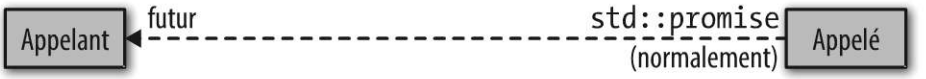


Figure 7.1 — Communication au travers d'un futur.

Mais, où est enregistré le résultat de l'appelé ? Puisque l'appelé peut se terminer avant que l'appelant n'invoque `get` sur le futur correspondant, le résultat ne peut pas être stocké dans le `std::promise` de l'appelé. En effet, cet objet est local à l'appelé et il est détruit lorsque celui-ci se termine.

Le résultat ne peut pas être placé dans le futur de l'appelant, car, notamment, un `std::future` peut servir à créer un `std::shared_future` (la propriété du résultat de l'appelé est alors transférée du `std::future` au `std::shared_future`), qui peut ensuite être copié à plusieurs reprises après la destruction du `std::future` d'origine. Puisque certains types de résultats ne pourront pas être copiés (ceux réservés au déplacement) et que le résultat peut avoir une durée de vie au moins aussi longue que le dernier

1. Le conseil 39 explique que le canal de communication associé à un futur peut avoir d'autres usages. Mais, dans ce conseil, nous considérons qu'il sert uniquement de mécanisme de transmission d'un résultat depuis l'appelé vers l'appelant.

Copyright © 2016 Dunod.
© Dunod – Toute reproduction non autorisée est un délit.

futur qui y fait référence, comment déterminer, parmi tous les futurs qui correspondent potentiellement à l'appelé, celui qui contient le résultat ?

Puisqu'aucun des objets associés à l'appelé et qu'aucun des objets associés à l'appelant ne peuvent servir à mémoriser le résultat de l'appelé, il est placé dans un endroit extérieur à ces objets. Cet emplacement est appelé *état partagé*. Il est généralement représenté par un objet sur le tas, mais ses type, interface et implémentation ne sont pas spécifiés par la norme. Les auteurs de la bibliothèque standard peuvent implémenter les états partagés comme bon leur semble.

La figure 7.2 montre comment nous pouvons envisager la relation entre l'appelé, l'appelant et l'état partagé, où les flèches en pointillés représentent toujours le flux des informations.



Figure 7.2 — Mémorisation du résultat dans un état partagé.

L'existence de l'état partagé est importante, car le comportement du destructeur d'un futur, sujet de ce conseil, est déterminé par l'état partagé associé au futur :

- **Le destructeur du dernier futur faisant référence à un état partagé pour une tâche non reportée lancée *via* `std::async` reste bloqué** jusqu'à la terminaison de la tâche. En substance, le destructeur d'un tel futur effectue un `join` implicite sur le thread qui exécute la tâche asynchrone.
- **Le destructeur de tous les autres futurs détruit simplement l'objet futur.** Pour les tâches qui s'exécutent de façon asynchrone, cela équivaut à un `detach` implicite sur le thread sous-jacent. S'il s'agit du dernier futur d'une tâche reportée, celle-ci ne s'exécutera donc jamais.

Ces règles semblent plus complexes qu'elles ne le sont. En réalité, nous avons un comportement « normal » simple et une exception isolée. Le comportement normal est celui d'un destructeur de futur qui détruit l'objet futur. Il n'effectue aucun `join` ni `detach`, et n'exécute rien. Il détruit simplement les données membres du futur. (En vérité, il décrémente également le compteur de références présent dans l'état partagé manipulé par les futurs qui y font référence et le `std::promise` de l'appelé. Ce compteur de références permet à la bibliothèque de savoir quand l'état partagé peut être détruit. Pour de plus amples informations sur le comptage de références, consultez le conseil 19.)

L'exception à ce comportement normal se produit uniquement pour un futur qui présente l'ensemble des caractéristiques suivantes :

- **Il fait référence à un état partagé qui a été créé suite à un appel à `std::async`.**
- **La stratégie de démarrage de la tâche est `std::launch::async`** (voir le conseil 36), soit parce qu'elle a été choisie par le système d'exécution, soit parce qu'elle a été indiquée dans l'appel à `std::async`.

Copyright © 2016 Dunod.

- **Le futur est le dernier qui fait référence à l'état partagé.** Pour les `std::future`, ce sera toujours le cas. Pour les `std::shared_future`, si d'autres `std::shared_future` font référence au même état partagé que le futur en cours de destruction, ce dernier respecte le comportement normal (il détruit simplement ses données membres).

Le comportement spécial du destructeur d'un futur ne s'applique que si toutes ces conditions sont satisfaites. Il reste bloqué jusqu'à ce que la tâche qui s'exécute de façon asynchrone se termine. D'un point de vue pratique, cela correspond à un `join` implicite avec le thread qui exécute la tâche créée par `std::async`.

Très souvent, cette exception au comportement normal du destructeur d'un futur est résumée par « les futurs de `std::async` bloquent dans leur destructeur ». Si, en première approximation, ce raccourci est correct, il faut parfois être plus précis. Vous connaissez désormais la vérité, dans toute sa beauté.

Mais vous vous demandez peut-être pourquoi il existe une règle particulière pour les états partagés des tâches non reportées qui sont lancées par `std::async`. Votre interrogation est légitime. Pour ce que nous en savons, le comité de normalisation a voulu éviter les problèmes associés à un `detach` implicite (voir le conseil 37), mais n'a pas souhaité adopter une stratégie aussi radicale que la terminaison obligatoire du programme (comme ils l'ont fait pour les `std::thread` joignables ; voir également le conseil 37). Il a donc opté pour un compromis, un `join` implicite. Ce choix n'est pas allé sans controverse et des débats ont eu lieu sur l'abandon de ce comportement en C++14. Finalement, rien n'a changé et le comportement des destructeurs des futurs reste cohérent entre C++11 et C++14.

Puisque l'API des futurs ne permet pas de déterminer si un futur fait référence à un état partagé issu d'un appel à `std::async`, il n'est pas possible de savoir si le destructeur d'un objet futur arbitraire sera bloqué dans l'attente de la terminaison d'une tâche asynchrone. Ce constat a des implications intéressantes :

```
// Le destructeur de ce conteneur pourrait bloquer, car un ou
// plusieurs des futurs qu'il contient peut faire référence à un état
// partagé pour une tâche non reportée lancée via std::async.
std::vector<std::future<void>> futs;   // Voir le conseil 39 pour des
                                       // infos sur std::future<void>.

class Widget {                         // Des objets Widget pourraient
public:                                // bloquer dans leurs destructeurs.
  ...

private:
  std::shared_future<double> fut;
};
```

Bien entendu, si nous avons le moyen de savoir qu'un futur donné ne satisfait pas aux conditions nécessaires au comportement spécial du destructeur (par exemple, en raison de la logique du programme), nous pouvons être assurés que le destructeur de ce futur ne bloquera pas. Par exemple, seuls les états partagés qui proviennent d'appels à

© Dunod – Toute reproduction non autorisée est un délit.
Copyright © 2016 Dunod.

`std::async` peuvent conduire au comportement spécial, mais il existe d'autres façons de créer des états partagés. L'une d'elles passe par `std::packaged_task`. Un objet `std::packaged_task` prépare une fonction (ou tout autre objet invocable) pour une exécution asynchrone de sorte que son résultat soit placé dans un état partagé. Un futur qui fait référence à cet état partagé peut être obtenu à l'aide de la fonction `get_future` de `std::packaged_task` :

```
int calcValue();                          // Fonction à exécuter.

std::packaged_task<int()>                 // Emballer calcValue pour qu'elle
  pt(calcValue);                          // s'exécute de façon asynchrone.

auto fut = pt.get_future();               // Obtenir un futur pour pt.
```

Nous savons ainsi que le futur `fut` ne fait pas référence à un état partagé créé par un appel à `std::async`. Son destructeur se comportera donc normalement.

Après qu'il a été créé, le `std::packaged_task pt` peut s'exécuter sur un thread. (Il peut également être exécuté par un appel à `std::async`, mais si nous voulions exécuter une tâche avec `std::async`, il n'y a pas vraiment de raison de créer un `std::packaged_task`, car `std::async` réalise la même chose que `std::packaged_task` avant de planifier l'exécution de la tâche.)

Puisque les `std::packaged_task` ne sont pas copiables, nous devons convertir `pt` en rvalue (*via* `std::move` ; voir le conseil 23) lorsqu'il est passé au constructeur de `std::thread` :

```
std::thread t(std::move(pt));             // Exécuter pt sur t.
```

Cet exemple donne une idée du comportement normal du destructeur de futur, mais il est plus facile à constater lorsque les instructions sont réunies dans un bloc :

```
{                                         // Début du bloc.

  std::packaged_task<int()>
    pt(calcValue);

  auto fut = pt.get_future();

  std::thread t(std::move(pt));

  …                                       // Voir ci-après.

}                                         // Fin du bloc.
```

Le code le plus intéressant est représenté par les « … » qui suivent la création de l'objet `std::thread t` et précèdent la fin du bloc. C'est ce qui arrive à `t` dans ce code qui est intéressant. Il y a trois possibilités élémentaires :

- **Il n'arrive rien à** `t`. Dans ce cas, `t` sera joignable à la fin de la portée, ce qui provoquera la terminaison du programme (voir le conseil 37).

Copyright © 2016 Dunod.

- **Un** `join` **est effectué sur** `t`. Dans ce cas, il est inutile que le destructeur de `fut` bloque, car le `join` est déjà présent dans le code appelant.
- **Un** `detach` **est effectué sur** `t`. Dans ce cas, il est inutile que le destructeur de `fut` invoque `detach`, car cette opération est déjà réalisée par le code appelant.

Autrement dit, lorsque nous avons un futur qui correspond à un état partagé issu d'un `std::packaged_task`, il est en général inutile d'adopter une stratégie de destruction particulière, car le choix entre terminaison, jonction et détachement sera effectué par le code qui manipule le `std::thread` utilisé pour exécuter le `std::packaged_task`.

À retenir

- Le destructeur d'un futur se contente normalement de détruire les données membres du futur.
- Le dernier futur qui fait référence à un état partagé pour une tâche non reportée lancée *via* `std::async` bloque jusqu'à la terminaison de la tâche.

CONSEIL N° 39. ENVISAGER LES FUTURS `VOID` POUR COMMUNIQUER PONCTUELLEMENT UN ÉVÉNEMENT

Une tâche devra parfois avertir une autre tâche asynchrone de l'arrivée d'un événement particulier, car cette seconde tâche ne peut pas effectuer son travail tant que cet événement ne s'est pas produit. Il peut s'agir, par exemple, de l'initialisation d'une structure de données, de la fin d'une étape d'un calcul ou de la détection d'une valeur importante sur un capteur. Dans ce cas, quelle est la meilleure manière de mettre en place cette communication interthreads ?

Une approche évidente consiste à utiliser une variable de condition (*condvar*). Si nous appelons *tâche de détection* la tâche qui détecte la condition, et *tâche de réaction* celle qui réagit à la condition, la stratégie est simple : la tâche de réaction attend sur une variable de condition et la tâche de détection notifie cette condvar lorsque l'événement se produit. Prenons les variables suivantes :

```
std::condition_variable cv;              // condvar pour un événement.

std::mutex m;                            // mutex utilisé avec cv.
```

Le code de la tâche de détection ne saurait être plus simple :

```
...                                       // Détecter l'événement.

cv.notify_one();                          // Avertir la tâche de réaction.
```

Copyright © 2016 Dunod.

© Dunod – Toute reproduction non autorisée est un délit.

Si plusieurs tâches de réaction doivent être notifiées, nous pouvons remplacer `notify_one` par `notify_all`, mais, pour le moment, supposons qu'il n'existe qu'une seule tâche de réaction.

Le code de la tâche de réaction est un peu plus compliqué car, avant d'appeler `wait` sur la condvar, il doit verrouiller un mutex au travers d'un objet `std::unique_lock`. (Le verrouillage d'un mutex avant l'attente sur une variable de condition reste classique dans les bibliothèques de gestion des threads. La nécessité de verrouiller le mutex *via* un objet `std::unique_lock` fait simplement partie de l'API C++11.) Voici l'approche conceptuelle :

```
...                                     // Préparer la réaction.

{                                       // Ouvrir une section critique.

  std::unique_lock<std::mutex> lk(m);   // Verrouiller le mutex.

  cv.wait(lk);                          // Attendre la notification ;
                                        // ce code n'est pas correct !

  ...                                   // Réagir à l'événement
                                        // (m est verrouillé).

}                                       // Fermer la section critique ;
                                        // déverrouiller m via
                                        // le destructeur de lk.

...                                     // Poursuivre la réaction
                                        // (m est à présent libéré).
```

Le premier problème de cette approche réside dans ce qu'on appelle une *mauvaise odeur du code* (*code smell*) : même si ce code est opérationnel, il y a quelque chose qui sent mauvais (qui ne semble pas correct). Dans ce cas, l'odeur émane du besoin d'utiliser un mutex. Les mutex servent à contrôler les accès à des données partagées, mais il est parfaitement possible que les tâches de détection et de réaction n'aient pas besoin de cette médiation. Par exemple, la tâche de détection peut être responsable de l'initialisation d'une structure de données globale, laissant à la tâche de réaction le soin de l'utiliser. Si la tâche de détection n'accède jamais à la structure de données après l'avoir initialisée et si la tâche de réaction n'y accède jamais avant que la tâche de détection n'indique qu'elle est prête, la logique du programme fait en sorte que les deux tâches ne se rencontrent jamais. Le mutex est donc inutile. Le fait que la solution fondée sur une condvar exige un mutex laisse derrière lui comme un relent de mauvaise conception.

Même en laissant cela de côté, deux autres problèmes doivent absolument être pris en compte :

- **Si la tâche de détection notifie la condvar avant que la tâche de réaction n'appelle `wait`, celle-ci va bloquer**. Pour que la notification d'une condvar réveille une autre tâche, il faut que celle-ci soit en attente sur la condvar. Si

Copyright © 2016 Dunod.

la tâche de détection effectue sa notification avant que la tâche de réaction n'appelle `wait`, elle va manquer la notification et attendre indéfiniment.

- **L'instruction `wait` ne prend pas en compte les réveils intempestifs**. Avec les API de gestion des threads (dans de nombreux langages, pas seulement en C++), il est possible que le code qui attend sur une variable de condition soit réveillé même en l'absence de notification sur la condvar. Il s'agit de *réveils intempestifs*. Un code propre doit les prendre en compte en vérifiant que la condition d'attente a bien été satisfaite et procède à ce contrôle immédiatement après le réveil. Avec l'API des condvar de C++ cette vérification est exceptionnellement simple, car une expression lambda (ou tout autre objet fonction) peut être utilisée pour tester la condition d'attente à passer à `wait`. Autrement dit, nous pouvons écrire ainsi l'appel à `wait` dans la tâche de réaction :

```
cv.wait(lk,
        []{ return si l'événement s'est produit; });
```

Pour tirer parti de cette possibilité, la tâche de réaction doit être capable de déterminer si la condition d'attente est vraie. Mais, dans le scénario que nous examinons, la condition d'attente correspond à l'occurrence d'un événement que le thread de détection est chargé de reconnaître. Le thread de réaction n'a peut-être pas le moyen de déterminer que l'événement qu'il attend a eu lieu et c'est pour cela qu'il attend sur une variable de condition !

Dans de nombreuses situations, la mise en place d'une communication entre des tâches à l'aide d'une condvar convient parfaitement, mais cela ne semble pas le cas ici.

Une autre approche se fonde sur un drapeau booléen partagé. Ce drapeau est initialement fixé à `false` et, lorsque la tâche de détection reconnaît l'événement attendu, elle lève ce drapeau :

```
std::atomic<bool> flag(false);        // Drapeau partagé ; voir le
                                      // conseil 40 pour std::atomic.

...                                   // Détecter l'événement.

flag = true;                          // Notifier la tâche de réaction.
```

De son côté, la tâche de réaction consulte simplement l'état du drapeau. Lorsqu'elle voit qu'il est positionné, elle sait que l'événement attendu s'est produit :

```
...                                   // Se préparer à réagir.

while (!flag);                        // Attendre l'événement.

...                                   // Réagir à l'événement.
```

Cette approche ne souffre pas des inconvénients de la conception à base de condvar. Nul besoin d'un mutex, aucun problème si la tâche de détection positionne le drapeau

Copyright © 2016 Dunod.
© Dunod – Toute reproduction non autorisée est un délit.

avant que la tâche de réaction ne le consulte, et rien de comparable à un réveil intempestif. Très bien.

En revanche, le coût associé à la consultation du drapeau dans la page de réaction est beaucoup moins bien. Pendant que la tâche attend le positionnement du drapeau, elle est essentiellement bloquée, même si elle est en cours d'exécution. Elle occupe donc un thread matériel dont une autre tâche pourrait profiter, elle entraîne un changement de contexte chaque fois que sa tranche de temps débute ou se termine, et elle peut entretenir l'exécution d'un cœur qui pourrait sinon être arrêté afin d'économiser l'énergie. Une tâche réellement bloquée ne présenterait aucun de ces défauts. C'est là l'avantage de l'approche fondée sur une condvar, car une tâche qui se trouve dans un appel à `wait` est réellement bloquée.

Une alternative très répandue consiste à combiner drapeau et condvar. Un drapeau indique si l'événement concerné a eu lieu, mais l'accès à cet indicateur est synchronisé par un mutex. Puisque le mutex empêche les accès concurrents au drapeau, il est inutile, comme l'explique le conseil 40, que cet indicateur soit un `std::atomic` ; un simple `bool` suffit. Voici la tâche de détection mise en œuvre dans cette solution :

```
std::condition_variable cv;              // Comme précédemment.
std::mutex m;

bool flag(false);                        // Non un std::atomic.

…                                        // Détecter l'événement.

{
  std::lock_guard<std::mutex> g(m);      // Verrouiller m via
                                         // le constructeur de g.

  flag = true;                           // Avertir la tâche de réaction
                                         // (partie 1).

}                                        // Déverrouiller m via
                                         // le destructeur de g.

cv.notify_one();                         // Notifier la tâche de réaction
                                         // (partie 2).
```

Et voici la tâche de réaction :

```
…                                        // Se préparer à réagir.

{                                        // Comme précédemment.
  std::unique_lock<std::mutex> lk(m);    // Comme précédemment.

  cv.wait(lk, [] { return flag; });      // Utiliser une expression
                                         // lambda pour éviter les
                                         // réveils intempestifs.

  …                                      // Réagir à l'événement
                                         // (m est verrouillé).
```

Copyright © 2016 Dunod.

```
}

...                                    // Poursuivre la réaction
                                       // (m est à présent libéré).
```

Cette approche évite les problèmes décrits. Elle reste opérationnelle que la tâche de réaction appelle `wait` avant ou après la notification de la tâche de détection, n'est pas sensible aux éventuels réveils intempestifs et ne nécessite aucune interrogation active. Pourtant, une odeur perdure. En effet, la tâche de détection communique d'une drôle de manière avec la tâche de réaction. En notifiant la variable de condition, la tâche de réaction est avertie de l'arrivée probable de l'événement attendu, mais elle doit interroger le drapeau pour en être certaine. En positionnant le drapeau, la tâche de réaction sait que l'événement s'est bien produit, mais la tâche de détection doit néanmoins notifier la variable de condition pour que la tâche de réaction se réveille et consulte le drapeau. La solution fonctionne, mais elle ne semble pas très propre.

Une alternative consiste à éviter les variables de condition, les mutex et les drapeaux en faisant en sorte que la tâche de réaction appelle `wait` sur un futur fixé par la tâche de détection. Cette idée peut sembler bizarre, car le conseil 38 explique qu'un futur représente l'extrémité de réception d'un canal de communication entre un appelé et un appelant (généralement asynchrone) et qu'il n'existe aucune relation appelé-appelant entre les tâches de détection et de réaction. Cependant, le conseil 38 note également qu'un canal de communication dont l'extrémité d'émission est un `std::promise` et dont l'extrémité de réception est un futur peut servir à autre chose qu'une simple communication appelé-appelant. Ces canaux de communication sont utilisables lorsque nous avons besoin de transmettre des informations d'un endroit du programme à un autre. Nous allons nous en servir dans notre exemple pour transmettre une information depuis la tâche de détection vers la tâche de réaction. Cette information sera l'arrivée de l'événement.

La conception est simple. La tâche de détection possède un objet `std::promise` (c'est-à-dire l'extrémité d'émission du canal de communication) et la tâche de réaction dispose du futur correspondant. Lorsque la tâche de détection constate que l'événement attendu s'est produit, elle *fixe* la valeur du `std::promise` (c'est-à-dire écrit sur le canal de communication). Pendant ce temps, la tâche de réaction attend par `wait` sur son futur. Cet appel à `wait` bloque la tâche de réaction jusqu'à ce que la valeur du `std::promise` ait été fixée.

`std::promise` et les futurs (`std::future` et `std::shared_future`) sont des templates qui ont besoin d'un paramètre de type. Il indique le type des données transmises au travers du canal de communication. Cependant, dans notre cas, nous n'avons aucune donnée à transmettre, car la tâche de réaction s'intéresse uniquement au fait que son futur a été fixé. Pour ces templates, nous avons donc besoin d'un type qui indique qu'aucune donnée n'est transmise sur le canal de communication, autrement dit `void`. La tâche de détection utilise donc un `std::promise<void>`, et la tâche de réaction, un `std::future<void>` ou un `std::shared_future<void>`. La tâche de détection fixe son `std::promise<void>` lorsque l'événement se produit, et la tâche de réaction attend en appelant `wait` sur son futur. Même si la tâche de réaction ne reçoit aucune donnée de la part de la tâche de détection, le canal de communication lui permet de savoir

© Dunod – Toute reproduction non autorisée est un délit.
Copyright © 2016 Dunod.

que la tâche de détection a « envoyé » sa donnée `void` en appelant `set_value` sur son `std::promise`.

Avec la déclaration suivante :

```
std::promise<void> p;                    // Objet promise pour le
                                         // canal de communication.
```

Le code de la tâche de détection est simple :

```
...                                      // Détecter l'événement.

p.set_value();                           // Avertir la tâche de réaction.
```

Tout comme celui de la tâche de réaction :

```
...                                      // Se préparer à réagir.

p.get_future().wait();                   // Attendre sur le futur
                                         // qui correspond à p.

...                                      // Réagir à l'événement.
```

À l'instar de la solution à base d'un drapeau, cette conception ne nécessite aucun mutex, fonctionne même si la tâche de détection fixe son `std::promise` avant que la tâche de réaction n'appelle `wait`, et n'est pas sujette aux réveils intempestifs. (Seules les variables de condition sont concernées par ce problème.) Et, comme dans le cas de l'approche fondée sur la condvar, la tâche de réaction est réellement bloquée après son appel à `wait` et ne consomme donc aucune ressource système pendant l'attente. Tout est parfait.

En réalité, pas vraiment. Certes, l'approche fondée sur un futur permet d'éviter tous ces écueils, mais d'autres dangers nous guettent. Par exemple, le conseil 38 explique qu'un état partagé se trouve entre un `std::promise` et un futur, et que les états partagés sont généralement alloués dynamiquement. Nous devons donc supposer que cette conception implique le coût d'une allocation et d'une désallocation sur le tas.

Mais le plus important est que la valeur d'un `std::promise` ne peut être fixée qu'une seule fois. Le canal de communication entre un `std::promise` et un futur représente un mécanisme *ponctuel* : il ne peut être employé à plusieurs reprises. Il s'agit là d'une différence importante par rapport aux conceptions à base de condvar et de drapeau, dans lesquelles les communications multiples sont possibles. (Une condvar peut être modifiée à plusieurs reprises, et un drapeau peut toujours être abaissé et levé à nouveau.)

Cette restriction n'est pas aussi contraignante que vous pourriez le penser. Supposons que nous voulions créer un thread système dans un état suspendu. Autrement dit, nous voulons que tout le surcoût associé à la création d'un thread soit déporté afin d'éviter la latence de sa création lorsque nous sommes prêts à lui faire exécuter un traitement. Ou bien, nous voulons le créer dans cet état afin de pouvoir le configurer

Copyright © 2016 Dunod.

avant de démarrer son exécution. Pour cela, nous avons besoin de configurer sa priorité ou son affinité. L'API de concurrence du C++ ne le permet pas, mais `std::thread` offre la fonction membre `native_handle`, qui donne accès à l'API de gestion des threads de la plate-forme (habituellement les threads POSIX ou les threads Windows). Grâce à cette API de plus bas niveau, nous pouvons généralement configurer les caractéristiques d'un thread, notamment sa priorité et son affinité.

En supposant que nous voulions suspendre un thread une seule fois (après sa création, mais avant qu'il n'exécute la fonction indiquée), le choix d'une conception fondée sur un futur `void` est raisonnable. Voici la technique de base :

```
std::promise<void> p;

void react();                       // Fonction pour la tâche de réaction.

void detect()                       // Fonction pour la tâche de détection.
{
  std::thread t([]                  // Créer un thread.
                {
                  p.get_future().wait();     // Suspendre t jusqu'à ce
                  react();                   // que le futur soit fixé.
                });
  ...                               // Ici, t est suspendu avant
                                    // l'appel à react().

  p.set_value();                    // Réactiver t, et donc appeler
                                    // react().

  ...                               // Autres opérations.

  t.join();                         // Rendre t non joignable
}                                   // (voir le conseil 37).
```

Puisqu'il est important que `t` soit non joignable en dehors de `detect`, l'utilisation d'une classe RAII comme la classe `ThreadRAII` du conseil 37 est recommandée. Voici le code qui en découle :

```
void detect()
{
  ThreadRAII tr(                            // Utiliser un objet RAII.
    std::thread([]
                {
                  p.get_future().wait();
                  react();
                }),
    ThreadRAII::DtorAction::join            // Risqué ! (voir ci-après).
  );

  ...                                       // Le thread dans tr
                                            // est suspendu ici.

  p.set_value();                            // Réactiver le thread
                                            // dans tr.
```

Copyright © 2016 Dunod.
© Dunod – Toute reproduction non autorisée est un délit.

```
    ...
}
```

Mais il n'est pas aussi fiable qu'il peut paraître. Le problème vient de la première section « ... » (celle avec le commentaire « Le thread dans `tr` est suspendu ici. »). Si une exception est levée dans le code correspondant, `set_value` n'est jamais invoquée sur `p`. Autrement dit, l'appel à `wait` dans l'expression lambda ne retourne jamais. Par conséquent, le thread qui exécute l'expression lambda ne se termine pas. Le problème est là, car l'objet RAII `tr` a été configuré pour effectuer un `join` sur ce thread dans son destructeur. En résumé, si une exception est lancée dans la première région de code « ... », cette fonction va bloquer car le destructeur de `tr` ne se terminera jamais.

Il existe des solutions pour corriger ce problème, mais nous les laissons en exercice au lecteur[1]. Nous préférons vous montrer comment le code d'origine (c'est-à-dire sans utiliser `ThreadRAII`) peut être modifié pour suspendre puis réactiver non pas une mais plusieurs tâches de réaction. Il s'agit d'une généralisation simple, car elle consiste à utiliser dans le code `react` des `std::shared_future` à la place d'un `std::future`. Dès lors que nous savons que la fonction membre `share` de `std::future` transfère la propriété de son état partagé à l'objet `std::shared_future` produit par `share`, le code s'écrit pratiquement de lui-même. La seule subtilité vient du fait que chaque tâche de réaction a besoin de sa propre copie du `std::shared_future` qui fait référence à l'objet partagé. Le `std::shared_future` obtenu de `share` doit donc être capturé par valeur par les expressions lambda qui s'exécutent sur les threads de réaction :

```
std::promise<void> p;                    // Comme précédemment.

void detect()                            // À présent, plusieurs
{                                        // threads de réaction.

  auto sf = p.get_future().share();      // Le type de sf est
                                         // std::shared_future<void>.

  std::vector<std::thread> vt;           // Conteneur pour les threads
                                         // de réaction.

  for (int i = 0; i < threadsToRun; ++i) {
    vt.emplace_back([sf]{ sf.wait();       // Attendre sur la copie
                          react(); });     // locale de sf ; voir le
  }                                        // conseil 42 pour des infos
                                           // sur emplace_back.

  ...                                  // Détecter le blocage si ce code
                                       // "..." lance une exception !

  p.set_value();                       // Réactiver tous les threads.
```

1. Pour commencer vos recherches, vous pouvez consulter notre article du 24 décembre 2013 publié sur le site The View From Aristeia et intitulé « ThreadRAII + Thread Suspension = Trouble? » (*http://scottmeyers.blogspot.com/2013/12/threadraii-thread-suspension-trouble.html*).

Copyright © 2016 Dunod.

```
  ...

  for (auto& t : vt) {                  // Rendre tous les threads
    t.join();                           // non joignables ; voir le
  }                                     // conseil 2 pour des infos
}                                       // sur "auto&".
```

Il est remarquable qu'une conception fondée sur les futurs puisse arriver à ce fonctionnement et c'est pourquoi vous devez l'envisager pour les communications ponctuelles d'un événement.

À retenir

- Pour la communication simple d'un événement, les conceptions fondées sur une condvar exigent un mutex superflu, posent des contraintes sur la progression relative des tâches de détection et de réaction, et demandent aux tâches de réaction de vérifier que l'événement a eu lieu.
- Les conceptions qui reposent sur un drapeau évitent ces problèmes, mais mettent en place une interrogation non bloquante.
- Une condvar et un drapeau peuvent être combinés, mais le mécanisme de communication résultant est quelque peu guindé.
- L'utilisation de `std::promise` et de futurs évite ces problèmes, mais cette approche utilise le tas pour le stockage des états partagés et se limite à des communications ponctuelles.

CONSEIL N° 40. UTILISER `STD::ATOMIC` POUR LA CONCURRENCE, `VOLATILE` POUR LA MÉMOIRE SPÉCIALE

Pauvre `volatile`, si incompris. Il ne devrait même pas être cité dans ce chapitre, car il n'a aucun rapport avec la programmation concurrente. Pourtant, dans d'autres langages, comme Java et C#, il a son utilité dans ce type de programmation, et, même en C++, certains compilateurs ont connoté `volatile` d'une sémantique qui le rend approprié au développement de logiciels concurrents (mais uniquement lorsqu'ils sont compilés avec ces compilateurs). C'est pourquoi il est intéressant de discuter de `volatile` dans un chapitre sur la concurrence, ne serait-ce que pour dissiper le flou qui l'entoure.

La fonctionnalité C++ que les programmeurs confondent parfois avec `volatile` – celle qui n'a réellement aucun rapport avec ce chapitre – est le template `std::atomic`. Les instanciations de ce template (par exemple `std::atomic<int>`, `std::atomic<bool>`, `std::atomic<Widget*>`, etc.) apportent des opérations qui, vues des autres threads, sont atomiques. Après qu'un `std::atomic` a été construit, les opérations sur cet objet se comportent comme si elles se trouvaient à l'intérieur d'une section critique protégée par un mutex, alors qu'elles sont généralement implémentées à l'aide d'instructions machines spéciales dont l'efficacité est bien supérieure à l'emploi d'un mutex.

Prenons le code suivant qui utilise `std::atomic` :

© Dunod – Toute reproduction non autorisée est un délit.
Copyright © 2016 Dunod.

```
std::atomic<int> ai(0);      // Initialiser ai à 0.
ai = 10;                     // Fixer ai à 10 de façon atomique.
std::cout << ai;             // Lire la valeur de ai de façon atomique.
++ai;                        // Incrémenter ai à 11 de façon atomique.
--ai;                        // Décrémenter ai à 10 de façon atomique.
```

Pendant l'exécution de ces instructions, les autres threads qui lisent `ai` ne verraient que la valeur 0, 10 ou 11. Aucune autre valeur n'est possible (en supposant évidemment que `ai` ne soit modifié que par ce seul thread).

Focalisons-nous sur deux aspects de cet exemple. Premièrement, dans l'instruction « `std::cout << ai;` », le fait que `ai` soit un `std::atomic` garantit uniquement que la lecture de `ai` est atomique. Rien n'assure que l'intégralité de l'instruction s'exécute de façon atomique. Entre le moment où la valeur de `ai` est lue et celui où `operator<<` est invoqué pour l'écrire sur la sortie standard, un autre thread peut avoir modifié `ai`. Le comportement de l'instruction n'en est pas affecté, car `operator<<` pour les `int` utilise un passage par valeur du `int` à afficher (la valeur envoyée sur la sortie sera donc celle lue depuis `ai`). Cependant, il est important de comprendre que la seule partie atomique de cette instruction est la lecture de la valeur de `ai`.

Le second aspect intéressant de cet exemple réside dans le comportement des deux dernières instructions, l'incrémentation et la décrémentation de `ai`. Il s'agit de deux opérations de type lecture-modification-écriture (RMW, *read-modify-write*), mais elles s'exécutent de façon atomique. Voilà l'une des caractéristiques les plus appréciables des types `std::atomic` : dès lors qu'un objet `std::atomic` a été construit, toutes les fonctions membres invoquées sur cet objet, y compris celles qui contiennent des opérations RMW, sont vues par les autres threads comme atomiques.

À l'opposé, le code analogue qui utilise `volatile` n'apporte, dans un contexte multithread, aucune garantie particulière :

```
volatile int vi(0);          // Initialiser vi à 0.
vi = 10;                     // Fixer vi à 10.
std::cout << vi;             // Lire la valeur de vi.
++vi;                        // Incrémenter vi à 11.
--vi;                        // Décrémenter vi à 10.
```

Pendant l'exécution de ce code, les autres threads qui liraient la valeur de `vi` peuvent obtenir n'importe quel entier, comme –12, 68, 4090727, ou autre. Un tel code présente un comportement indéfini, car les instructions modifient `vi` et, si d'autres threads consultent `vi` au même moment, nous avons des lectures et des écritures simultanées sur une zone de mémoire qui n'est ni `std::atomic`, ni protégée par un mutex. Nous sommes devant la définition d'une situation de concurrence.

Copyright © 2016 Dunod.

Prenons un exemple concret du comportement différent des `std::atomic` et des `volatile` dans un programme multithread. Il met en œuvre un compteur de chaque type, incrémenté par plusieurs threads. Les compteurs sont initialisés à 0 :

```
std::atomic<int> ac(0);      // Compteur "atomique".

volatile int vc(0);          // Compteur "volatil".
```

Nous incrémentons ensuite chaque compteur une fois dans deux threads qui s'exécutent de façon simultanée :

```
/*- - - - -  Thread 1  - - - - - */     /*- - - - -  Thread 2  - - - - - */

             ++ac;                                  ++ac;
             ++vc;                                  ++vc;
```

Lorsque les deux threads sont terminés, la valeur de `ac` (celle du `std::atomic`) doit être égale à 2, car chaque incrémentation est réalisée sous forme d'une opération indivisible. En revanche, la valeur de `vc` peut ne pas être égale à 2, car ses incrémentations peuvent ne pas se faire de façon atomique. Chaque incrémentation comprend la lecture de la valeur de `vc`, l'incrémentation de la valeur qui a été lue, et l'écriture du résultat dans `vc`. Mais rien ne garantit que ces trois opérations sont réalisées de façon atomique sur un objet `volatile`. Il est donc possible que les opérations individuelles de deux incrémentations de `vc` s'entrelacent de la façon suivante :

1. Le thread 1 lit la valeur de `vc`, c'est-à-dire 0.
2. Le thread 2 lit la valeur de `vc`, c'est-à-dire toujours 0.
3. Le thread 1 incrémente la valeur 0 qu'il a lue et obtient 1, puis écrit cette valeur dans `vc`.
4. Le thread 2 incrémente la valeur 0 qu'il a lue et obtient 1, puis écrit cette valeur dans `vc`.

La valeur finale de `vc` est donc 1, même si la variable a été incrémentée deux fois.

Il ne s'agit pas du seul résultat possible. En général, il est impossible de prévoir la valeur finale de `vc`, car cette variable est impliquée dans une situation de concurrence et la norme stipule que de telles situations conduisent à un comportement indéfini, autrement dit que le compilateur peut générer un code qui effectue, littéralement, n'importe quoi. Bien entendu, les compilateurs ne profitent pas de cette liberté de façon inconsidérée. Ils mettent en place des optimisations qui sont valides lorsque les programmes ne présentent pas des situations de concurrence, mais elles conduisent à des comportements inattendus et imprévisibles dans ceux qui en contiennent.

Les opérations RMW ne sont pas les seuls cas de concurrence où les `std::atomic` réussissent et où les `volatile` échouent. Supposons qu'une tâche calcule une valeur nécessaire à une seconde tâche. Lorsque la première a terminé son calcul, elle communique son résultat à la seconde. Le conseil 39 explique que la première tâche

Copyright © 2016 Dunod.
© Dunod – Toute reproduction non autorisée est un délit.

peut indiquer à la seconde tâche la disponibilité de la valeur souhaitée en utilisant un `std::atomic<bool>`. Voici une mise en œuvre possible de la tâche de calcul :

```
std::atomic<bool> valAvailable(false);

auto imptValue = computeImportantValue();      // Calculer la valeur.

valAvailable = true;                           // Indiquer à l'autre tâche
                                               // qu'elle est disponible.
```

En tant qu'êtres humains lisant ce code, nous savons qu'il est essentiel que l'affectation de `imptValue` se fasse avant l'affectation de `valAvailable`, mais tous les compilateurs y voient deux affectations de variables indépendantes. En règle générale, les compilateurs sont autorisés à changer l'ordre des affectations qui n'ont pas de rapport l'une avec l'autre. Prenons, par exemple, les affectations suivantes (où `a`, `b`, `x` et `y` sont des variables indépendantes) :

```
a = b;
x = y;
```

Le compilateur peut généralement les réordonner ainsi :

```
x = y;
a = b;
```

Même si le compilateur ne change pas l'ordre, le matériel sous-jacent peut le faire (ou peut le faire croire à d'autres cœurs), car cela permet parfois d'améliorer les performances.

En revanche, l'utilisation de `std::atomic` impose des restrictions sur les possibilités de réorganisation du code. L'une d'elles est qu'aucun code qui, dans le code source, précède l'écriture d'une variable `std::atomic` ne peut se faire (ou ne peut apparaître aux autres cœurs comme se faisant) ensuite[1]. Prenons le code suivant :

```
auto imptValue = computeImportantValue();      // Calculer la valeur.

valAvailable = true;                           // Indiquer à l'autre tâche
                                               // qu'elle est disponible.
```

1. Ce n'est vrai que pour les `std::atomic` qui utilisent la *cohérence séquentielle*. Il s'agit à la fois du seul modèle de concurrence et du modèle par défaut pour les objets `std::atomic` qui emploient la syntaxe illustrée dans cet ouvrage. C++11 reconnaît également des modèles de cohérence dont les règles de réorganisation du code sont plus souples. Avec ces modèles *lâches* (dans le sens *assouplis*), nous pouvons créer des logiciels qui s'exécutent plus rapidement sur certaines architectures matérielles, mais leur utilisation donne des logiciels *beaucoup* plus difficiles à mettre au point, à comprendre et à maintenir. Il n'est pas rare que, même pour les experts, des erreurs subtiles se glissent dans le code fondé sur des opérations atomiques assouplies et vous devez vous limiter autant que possible à la cohérence séquentielle.

Copyright © 2016 Dunod.

Il faut non seulement que le compilateur conserve l'ordre des affectations de `imptValue` et de `valAvailable`, mais également que le code qu'il génère oblige le matériel sous-jacent à faire de même. Par conséquent, en déclarant `valAvailable` comme un `std::atomic`, nous nous assurons que la contrainte d'ordre essentielle – tous les threads doivent voir que `imptValue` ne change pas après que `valAvailable` a été modifiée – est maintenue.

En déclarant `valAvailable volatile`, ces restrictions sur l'ordre du code ne s'imposent plus :

```
volatile bool valAvailable(false);

auto imptValue = computeImportantValue();

valAvailable = true;          // D'autres threads peuvent voir cette
                              // affectation avant celle de imptValue !
```

Dans ce cas, le compilateur peut inverser l'ordre des affectations de `imptValue` et de `valAvailable`. Même s'il ne le fait pas, le code machine généré pourrait ne pas empêcher le matériel sous-jacent de donner à du code qui s'exécute sur d'autres cœurs la possibilité de voir la modification de `valAvailable` avant celle de `imptValue`.

Ces deux problèmes – aucune garantie d'atomicité d'exécution et contraintes insuffisantes sur l'ordre du code – expliquent pourquoi `volatile` a peu d'utilité dans la programmation concurrente, mais cela n'indique rien sur son éventuel intérêt. En bref, il permet d'indiquer au compilateur qu'il manipule une zone de mémoire au comportement anormal.

Si nous écrivons une valeur dans une zone de mémoire « normale », cette valeur est conservée jusqu'à ce qu'elle soit remplacée. Supposons que nous ayons un `int` normal :

```
int x;
```

et une séquence d'opérations sur cette variable :

```
auto y = x;                   // Lire x.
y = x;                        // Lire x à nouveau.
```

le compilateur peut optimiser le code généré en supprimant l'affectation de `y` car elle est redondante avec son initialisation.

Dans le cas d'une mémoire normale, si nous écrivons une valeur à un emplacement donné, ne la lisons jamais, puis écrivons à nouveau une valeur à cet emplacement mémoire, la première écriture peut être supprimée car elle n'est jamais utilisée. Supposons les deux instructions adjacentes suivantes :

```
x = 10;                       // Écrire x.
x = 20;                       // Écrire x à nouveau.
```

Copyright © 2016 Dunod.
© Dunod – Toute reproduction non autorisée est un délit.

Le compilateur peut retirer la première écriture. Supposons à présent que le code source comprenne les instructions suivantes :

```
auto y = x;                 // Lire x.
y = x;                      // Lire x à nouveau.

x = 10;                     // Écrire x.
x = 20;                     // Écrire x à nouveau.
```

Le compilateur peut considérer qu'il a été écrit ainsi :

```
auto y = x;                 // Lire x.

x = 20;                     // Écrire x.
```

Au cas où vous demanderiez qui écrirait du code avec de telles lectures redondantes et écritures superflues, techniquement appelées « chargements redondants » (*redundant loads*) et « stockages morts » (*dead stores*), sachez que les programmeurs ne le feraient pas directement, tout au moins nous l'espérons. Cependant, après que le compilateur a étudié le code source et réalisé l'instanciation des templates, l'inlining et les différentes sortes d'optimisation par réorganisation, il n'est pas rare que le résultat comprenne des chargements redondants et des stockages morts dont le compilateur peut se débarrasser.

Ces optimisations sont valides uniquement lorsque la mémoire se comporte normalement. Ce n'est pas le cas de la mémoire « spéciale ». La mémoire spéciale la plus répandue est probablement celle employée pour les *entrées-sorties mappées en mémoire* (*memory-mapped I/O*). Les emplacements dans cette mémoire sont utilisés pour communiquer avec les périphériques, par exemple les capteurs ou les affichages externes, les imprimantes, les ports réseau, etc., non pour lire ou écrire de la mémoire normale (c'est-à-dire de la RAM). Dans ce contexte, reprenons le code qui contient des lectures semble-t-il redondantes :

```
auto y = x;                 // Lire x.
y = x;                      // Lire x à nouveau.
```

Si `x` correspond à la valeur mesurée par un capteur de température, la seconde lecture de `x` n'est pas redondante car la température peut avoir changé entre la première et la seconde lecture.

La situation peut être comparable pour des écritures à première vue superflues. Par exemple :

```
x = 10;                     // Écrire x.
x = 20;                     // Écrire x à nouveau.
```

Si `x` représente le port de contrôle d'un émetteur radio, ce code pourrait envoyer des commandes à la radio, les valeurs 10 et 20 correspondant à des commandes différentes.

Copyright © 2016 Dunod.

L'optimisation sur la première affectation changerait l'ordre des commandes envoyées à la radio.

Pour indiquer au compilateur qu'il manipule de la mémoire spéciale, nous pouvons employer `volatile`. Ce mot clé signifie « ne pas optimiser les opérations effectuées sur cette mémoire ». Par conséquent, si `x` correspond à de la mémoire spéciale, elle doit être déclarée `volatile` :

```
volatile int x;
```

Voyons son effet sur notre séquence de code d'origine :

```
auto y = x;           // Lire x.
y = x;                // Lire x à nouveau (ne peut pas être optimisé).

x = 10;               // Écrire x (ne peut pas être optimisé).
x = 20;               // Écrire x à nouveau.
```

C'est exactement le comportement que nous voulons si `x` correspond à de la mémoire mappée (ou a été mappée sur un emplacement mémoire partagé entre plusieurs processus, etc.).

Petite question : dans la dernière partie du code, quel est le type de `y`, `int` ou `volatile int`[1] ?

Les chargements redondants et les stockages morts apparents doivent donc être conservés lorsque la mémoire manipulée est spéciale. Cela explique pourquoi les `std::atomic` ne conviennent pas dans ce contexte. Le compilateur a le droit d'éliminer de telles opérations redondantes sur les `std::atomic`. Le code est différent de celui qui utilise des `volatile`, mais, en mettant de côté cet aspect pour le moment et en nous focalisant sur les possibilités du compilateur, nous pouvons dire que, conceptuellement, le compilateur peut prendre le code suivant :

```
std::atomic<int> x;

auto y = x;       // Conceptuellement, lire x (voir ci-après).
y = x;            // Conceptuellement, lire x à nouveau (voir ci-après)

x = 10;               // Écrire x.
x = 20;               // Écrire x, à nouveau.
```

et l'optimiser ainsi :

1. Le type de `y` est obtenu par déduction `auto`, selon les règles décrites au conseil 2. Elles stipulent que les qualificatifs `const` et `volatile` sont retirés pour les déclarations de types non pointeurs et non références (ce qui est le cas de `y`). Le type de `y` est donc un simple `int`. Cela signifie que les lectures et écritures redondantes pour `y` peuvent être enlevées. Dans l'exemple, le compilateur doit effectuer l'initialisation et l'affectation de `y`, car `x` est `volatile`. La seconde lecture de `x` peut donc produire une valeur différente de la première.

Copyright © 2016 Dunod.
© Dunod – Toute reproduction non autorisée est un délit.

```
auto y = x;            // Conceptuellement lire x (voir ci-après).
x = 20;                // Écrire x.
```

Avec la mémoire spéciale, ce comportement est clairement inacceptable.

En réalité, la compilation de ces deux instructions échouera si `x` est un `std::atomic` :

```
auto y = x;            // Erreur !
y = x;                 // Erreur !
```

En effet, les opérations de copie pour `std::atomic` sont supprimées (voir le conseil 11), cela pour une bonne raison. Examinons ce qui se produirait si l'initialisation de `y` à partir de `x` passait la compilation. Puisque `x` est un `std::atomic`, le type déduit pour `y` serait également `std::atomic` (voir le conseil 2). Nous avons indiqué précédemment que l'un des avantages des `std::atomic` résidait dans l'atomicité de toutes leurs opérations. Mais, pour que la construction par copie de `y` à partir de `x` soit atomique, le compilateur devrait générer du code qui lit `x` et écrit `y` en une seule opération atomique. En général, le matériel n'en est pas capable et la construction par copie n'est pas prise en charge pour les types `std::atomic`. L'affectation par copie est supprimée pour la même raison. Voilà pourquoi la compilation de l'affectation de `x` à `y` échoue. (Puisque les opérations de déplacement ne sont pas déclarées explicitement dans `std::atomic`, `std::atomic` ne propose ni la construction ni l'affectation par déplacement, conformément aux règles de génération des fonctions spéciales par le compilateur décrites dans le conseil 17.)

Il est possible de placer la valeur de `x` dans `y`, mais il faut pour cela utiliser les fonctions membres `load` et `store` de `std::atomic`. La fonction `load` lit la valeur d'un `std::atomic` de façon atomique, tandis que la fonction `store` l'écrit de façon atomique. Voici le code qui permet d'initialiser `y` à partir de `x`, puis de stocker la valeur de `x` dans `y` :

```
std::atomic<int> y(x.load());          // Lire x.

y.store(x.load());                     // Lire x à nouveau.
```

La compilation se passe bien mais, en raison de la lecture de `x` (*via* `x.load()`) avec un appel de fonction séparé de l'initialisation ou de la modification de `y`, il est évident qu'il ne faut pas espérer que l'une ou l'autre de ces instructions soit exécutée comme une seule opération atomique.

Le compilateur peut « optimiser » ce code en plaçant la valeur de `x` dans un registre au lieu de la lire à deux reprises :

Copyright © 2016 Dunod.

```
register = x.load();              // Lire x dans register.

std::atomic<int> y(register);     // Initialiser y avec la valeur
                                  // de register.

y.store(register);                // Placer la valeur de register dans y.
```

Vous le constatez, la variable `x` n'est lue qu'une seule fois, mais ce type d'optimisation doit être évité dans le cas de la mémoire spéciale. (L'optimisation est interdite avec les variables `volatile`.)

À présent, la situation doit être plus claire :

- `std::atomic` est utile pour la programmation concurrente, mais pas pour l'accès à la mémoire spéciale.
- `volatile` est utile pour l'accès à la mémoire spéciale, mais pas pour la programmation concurrente.

Puisque `std::atomic` et `volatile` ont des utilités différentes, nous pouvons les employer ensemble :

```
volatile std::atomic<int> vai;        // Les opérations sur vai sont
                                      // atomiques et ne doivent pas
                                      // être optimisées.
```

Cela servira si `vai` correspond à un emplacement d'entrée-sortie mappé en mémoire auquel plusieurs threads accèdent de façon concurrente.

Pour finir, notons que certains développeurs préfèrent employer les fonctions membres `load` et `store` de `std::atomic` même s'ils n'y sont pas obligés, car le code source indique alors que les variables concernées ne sont pas « normales ». Il est intéressant de souligner ce fait. L'accès à un `std::atomic` est en général plus lent qu'un accès à un objet non `std::atomic`, et nous avons déjà vu que l'utilisation des `std::atomic` empêche le compilateur d'appliquer certaines formes de réorganisation du code. En invoquant `load` et `store` de `std::atomic`, il est plus facile d'identifier les endroits où l'évolutivité risque d'être mise à mal. Du point de vue de la conformité, ne pas voir d'appel à `store` pour une variable servant à communiquer une information à d'autres threads (par exemple un drapeau qui indique la disponibilité de données) pourrait indiquer que la variable n'a pas été déclarée `std::atomic` alors qu'elle aurait dû l'être.

Toutefois, il s'agit essentiellement d'une question de style, en cela très différente du choix entre `std::atomic` et `volatile`.

À retenir

- `std::atomic` est utilisé pour les données auxquelles plusieurs threads accèdent sans passer par des mutex. Il s'agit d'un outil d'écriture de logiciels concurrents.
- `volatile` est utilisé lorsque les lectures et les écritures d'une mémoire ne doivent pas être optimisées. Il s'agit d'un outil de manipulation de la mémoire spéciale.

Copyright © 2016 Dunod.
© Dunod – Toute reproduction non autorisée est un délit.

8

Finitions

Pour chaque technique ou fonctionnalité générale de C++, il existe des situations dans lesquelles son utilisation est sensée et d'autres où ce n'est pas le cas. Il est souvent assez facile de décrire quand l'utilisation d'une technique ou d'une fonctionnalité est légitime, mais ce chapitre donne deux exceptions. La technique générale concerne le passage par valeur, la fonctionnalité générale, le placement. Leur bonne utilisation est conditionnée par un nombre de facteurs si important que le meilleur conseil que nous puissions vous donner est d'*envisager* leur utilisation. Mais elles sont des piliers de la programmation moderne efficace en C++. Les conseils qui suivent apportent donc les informations dont vous aurez besoin pour déterminer si elles conviennent à votre logiciel.

CONSEIL N° 41. ENVISAGER UN PASSAGE PAR VALEUR POUR LES PARAMÈTRES COPIABLES DONT LE DÉPLACEMENT EST BON MARCHÉ ET QUI SONT TOUJOURS COPIÉS

Certains paramètres de fonction sont faits pour être copiés[1]. Par exemple, une fonction membre `addName` pourrait copier son paramètre dans un conteneur privé. Pour des raisons d'efficacité, elle copierait les arguments lvalue, mais déplacerait les arguments rvalue :

1. Dans ce conseil, « copier » un paramètre signifie généralement l'utiliser en tant que source d'une opération de copie ou de déplacement. Rappelons que la terminologie de C++ ne permet pas de différencier une copie réalisée par une opération de copie et une copie réalisée par une opération de déplacement.

Copyright © 2016 Dunod.
© Dunod – Toute reproduction non autorisée est un délit.

```
class Widget {
public:
  void addName(const std::string& newName)      // Prendre une lvalue ;
  { names.push_back(newName); }                 // la copier.

  void addName(std::string&& newName)           // Prendre une rvalue ;
  { names.push_back(std::move(newName)); }      // la déplacer ; voir le
  …                                             // conseil 25 pour des
                                                // infos sur std::move.
private:
  std::vector<std::string> names;
};
```

Cela fonctionne, mais nous devons écrire deux fonctions pour effectuer essentiellement la même chose. C'est un peu pénible : deux fonctions à déclarer, deux fonctions à implémenter, deux fonctions à documenter, deux fonctions à maintenir. Une horreur !

Par ailleurs, le code objet comprendra deux fonctions, ce qui risque d'être rédhibitoire si l'empreinte mémoire du programme est un critère essentiel. Dans ce cas, les deux fonctions deviendront probablement inline, ce qui éliminera les problèmes de gonflement liés à leur existence. Mais, si elles sont mises inline partout, avons-nous vraiment besoin de deux fonctions dans le code objet ?

Une solution alternative consiste à convertir `addName` en un template qui prend une référence universelle (voir le conseil 24) :

```
class Widget {
public:
  template<typename T>                          // Prendre des lvalues
  void addName(T&& newName)                     // et des rvalues ;
  {                                             // copier les lvalues,
    names.push_back(std::forward<T>(newName));  // déplacer les rvalues ;
  }                                             // voir le conseil 25
                                                // pour des infos sur
  …                                             // std::forward.

};
```

Le code source à gérer est moindre, mais l'emploi de références universelles amène d'autres complications. Puisque `addName` est un template, son implémentation doit généralement aller dans un fichier d'en-tête. Cela peut conduire à plusieurs fonctions dans le code objet, car le template est non seulement instancié différemment pour les lvalues et les rvalues, mais également pour les `std::string` et les types qui peuvent être convertis en `std::string` (voir le conseil 25). Par ailleurs, certains types d'arguments ne peuvent pas être passés *via* des références universelles (voir le conseil 30) et, si du code client transmet des types d'arguments impropres, les messages d'erreur du compilateur peuvent être perturbants (voir le conseil 27).

Il serait préférable de trouver une solution qui permette d'écrire des fonctions comme `addName` de sorte que les lvalues soient copiées, que les rvalues soient déplacées, qu'il n'y ait qu'une fonction à gérer (dans le code source et dans le code objet), et que les difficultés liées aux références universelles soient évitées. Bonne nouvelle, elle

Copyright © 2016 Dunod.

existe. Il suffit d'oublier l'une des premières règles apprise lors de nos débuts en tant que programmeur C++, à savoir éviter de passer par valeur des objets dont le type est défini par l'utilisateur. Pour des paramètres comme `newName` dans des fonctions comme `addName`, le passage par valeur peut être une stratégie parfaitement raisonnable.

Avant d'expliquer pourquoi le passage par valeur conviendrait si bien à `newName` et à `addName`, étudions sa mise en œuvre :

```
class Widget {
public:
  void addName(std::string newName)          // Prendre une lvalue ou
  { names.push_back(std::move(newName)); }   // une rvalue ; la déplacer.

  ...

};
```

La seule partie un peu compliquée de ce code réside dans l'application de `std::move` au paramètre `newName`. En général, `std::move` est employé avec des références rvalue, mais, dans ce cas, nous savons que (1) `newName` est un objet totalement indépendant de l'argument passé par l'appelant et modifier `newName` n'affectera pas ce dernier, et que (2) il s'agit de la dernière utilisation de `newName` et donc que son déplacement n'aura aucun impact sur le reste de la fonction.

Puisqu'il n'existe qu'une seule fonction `addName`, nous évitons la duplication tant dans le code source que dans le code objet. Puisque nous n'utilisons pas une référence universelle, nous ne subissons pas le gonflement des fichiers d'en-tête, les cas de dysfonctionnements étranges, ni les messages d'erreur perturbants. Mais, qu'en est-il de l'efficacité de cette conception ? Le passage *par valeur* n'est-il pas coûteux ?

En C++98, il était raisonnable de le supposer. Quel que soit l'argument transmis par l'appelant, le paramètre `newName` aurait été *construit par copie*. En revanche, en C++11, `addName` sera construit par copie uniquement pour les lvalues, alors qu'il sera *construit par déplacement* pour les rvalues. Voyons cela :

```
Widget w;

...

std::string name("Bart");

w.addName(name);                  // Appeler addName avec une lvalue.

...

w.addName(name + "Jenne");        // Appeler addName avec une rvalue
                                  // (voir ci-après).
```

Dans le premier appel à `addName` (passage de `name`), l'initialisation du paramètre `newName` se fait avec une lvalue. `newName` est donc construit par copie, comme il le serait en C++98. Dans le second appel, `newName` est initialisé avec l'objet `std::string` qui

Copyright © 2016 Dunod.
© Dunod – Toute reproduction non autorisée est un délit.

résulte d'un appel à `operator+` pour un `std::string` (c'est-à-dire l'opération d'ajout). Cet objet étant une rvalue, `newName` est construit par déplacement.

Ainsi, les lvalues sont copiées et les rvalues sont déplacées, exactement comme nous le voulions. Super !

C'est effectivement très bien, mais il ne faut pas oublier certaines mises en garde. Cela sera plus facile si nous récapitulons les trois versions de `addName` envisagées :

```
class Widget {                                    // Approche 1 :
public:                                           // surcharges pour
  void addName(const std::string& newName)        // les lvalues et
  { names.push_back(newName); }                   // les rvalues.

  void addName(std::string&& newName)
  { names.push_back(std::move(newName)); }
  …

private:
  std::vector<std::string> names;
};

class Widget {                                    // Approche 2 :
public:                                           // utiliser une
  template<typename T>                            // référence universelle.
  void addName(T&& newName)
  { names.push_back(std::forward<T>(newName)); }
  …
};

class Widget {                                    // Approche 3 :
public:                                           // passage par valeur.
  void addName(std::string newName)
  { names.push_back(std::move(newName)); }
  …
};
```

Les deux premières versions seront dites « approches par référence », car elles se fondent toutes deux sur le passage par référence des paramètres.

Voici les deux scénarios d'appel étudiés :

```
Widget w;
…
std::string name("Bart");

w.addName(name);                        // Passer une lvalue.
…
w.addName(name + "Jenne");              // Passer une rvalue.
```

Examinons à présent le coût, sur le plan des opérations de copie et de déplacement, de l'ajout d'un nom à un `Widget`, dans le contexte des deux scénarios d'appel et de chacune des trois implémentations de `addName`. Nous ignorerons les éventuelles optimisations des opérations de copie et de déplacement que les compilateurs peuvent

Copyright © 2016 Dunod.

réaliser car elles dépendent du contexte et du compilateur, et, en pratique, ne changent pas le fond de l'analyse.

- **Surcharge :** que l'argument transmis soit une lvalue ou une rvalue, il est lié à une référence nommée `newName`. Du point de vue des opérations de copie et de déplacement, le coût est nul. Dans la surcharge pour une lvalue, `newName` est copié dans `Widget::names`. Dans la surcharge pour une rvalue, il est déplacé. Résumé des coûts : une copie pour les lvalues, un déplacement pour les rvalues.
- **Utilisation d'une référence universelle :** comme pour la surcharge, l'argument de l'appelant est lié à la référence `newName`. Le coût de cette opération est nul. En raison de l'application de `std::forward`, les arguments lvalue `std::string` sont copiés dans `Widget::names`, tandis que les arguments rvalue `std::string` sont déplacés. En résumé, les coûts pour les arguments `std::string` sont identiques à ceux de la surcharge : une copie pour les lvalues, un déplacement pour les rvalues.
  Le conseil 25 explique que si l'argument passé n'est pas un `std::string`, il sera transmis à un constructeur de `std::string`, ce qui pourra ne provoquer aucune opération de copie ou de déplacement d'un `std::string`. Les fonctions qui prennent en arguments des références universelles peuvent donc être très efficaces, mais, puisque cela n'affecte pas notre analyse dans le contexte de ce conseil, nous supposerons simplement que les appelants transmettent toujours des arguments `std::string`.
- **Passage par valeur :** que l'argument soit une lvalue ou une rvalue, le paramètre `newName` doit être construit. Dans le cas d'une lvalue, cela coûte une construction par copie. Dans le cas d'une rvalue, le coût est celui d'une construction par déplacement. Dans le corps de la fonction, `newName` est systématiquement déplacé dans `Widget::names`. Résumé des coûts : une copie plus un déplacement pour les lvalues, deux déplacements pour les rvalues. En comparaison des approches par référence, nous avons donc un déplacement supplémentaire pour les lvalues et les rvalues.

Reprenons l'intitulé de ce conseil :

> Envisager un passage par valeur pour les paramètres copiables dont le déplacement est bon marché et qui sont toujours copiés.

Si nous l'avons écrit de cette manière, c'est pour une bonne raison ; quatre en réalité :

1. Nous devons uniquement *envisager* le passage par valeur. Oui, il permet de n'écrire qu'une seule fonction. Oui, il génère une seule fonction dans le code objet. Oui, il évite les problèmes associés aux références universelles. En revanche, son coût est plus élevé que celui des autres approches et, comme nous le verrons plus loin, certains cas cachent des dépenses que nous n'avons pas encore abordées.

Copyright © 2016 Dunod.
© Dunod – Toute reproduction non autorisée est un délit.

Étudions une classe qui comprend une donnée membre `std::unique_ptr<std::string>` et le mutateur associé. Puisque `std::unique_ptr` est un type réservé au déplacement, l'approche « par surcharge » pour ce mutateur est constituée d'une seule fonction :

2. Le passage par valeur doit être envisagé uniquement pour les *paramètres copiables*. Les paramètres qui ne satisfont pas à ce critère sont certainement d'un type réservé au déplacement car, s'ils ne sont pas copiables et bien que la fonction réalise toujours une copie, cette copie doit être créée par le constructeur de déplacement[1]. Rappelons que, par rapport à la surcharge, le passage par valeur a l'avantage d'exiger l'écriture d'une seule fonction. Mais, pour les types réservés au déplacement, il est inutile de fournir une surcharge pour les arguments lvalue. En effet, la copie d'une lvalue implique l'appel du constructeur de copie, qui est désactivé pour les types réservés au déplacement. Autrement dit, seuls les arguments rvalue doivent être pris en charge et, dans ce cas, l'approche « par surcharge » n'a besoin que d'une seule surcharge : celle qui prend une référence rvalue.

```
class Widget {
public:
  …
  void setPtr(std::unique_ptr<std::string>&& ptr)
  { p = std::move(ptr); }

private:
  std::unique_ptr<std::string> p;
};
```

Voici comment l'employer dans le code appelant :

```
Widget w;
…

w.setPtr(std::make_unique<std::string>("Modern C++"));
```

Le `std::unique_ptr<std::string>` rvalue retourné par `std::make_unique` (voir le conseil 21) est passé comme une référence rvalue à `setPtr`, où il est déplacé dans la donnée membre `p`. Le coût total correspond à un déplacement.
Supposons que `setPtr` prenne son paramètre par valeur :

```
class Widget {
public:
  …
```

1. C'est en raison de phrases comme celle-ci qu'il serait bon de disposer d'une terminologie qui distingue les copies effectuées par des opérations de copie et celles effectuées par des opérations de déplacement.

Copyright © 2016 Dunod.

```
  void setPtr(std::unique_ptr<std::string> ptr)
  { p = std::move(ptr); }

  ...
};
```

Le même appel construit par déplacement le paramètre `ptr`, et `ptr` est ensuite affecté à la donnée membre de `p`. Le coût total est donc celui de deux déplacements, c'est-à-dire deux fois le coût de l'approche « par surcharge ».

3. Le passage par valeur doit être envisagé uniquement pour les paramètres dont le *déplacement est bon marché*. Lorsque le coût des déplacements est faible, nous pouvons accepter un déplacement supplémentaire. Dans le cas contraire, un déplacement inutile équivaut à une copie inutile et c'est justement l'importance d'éviter les opérations de copie inutiles qui recommande d'éviter le passage par valeur en C++98 !
4. Le passage par valeur doit être envisagé uniquement pour les paramètres qui sont *toujours copiés*. Pour comprendre l'importance de ce point, supposons que, avant de copier son paramètre dans le conteneur `names`, `addName` vérifie si le nom est trop court ou trop long. Dans l'affirmative, la demande d'ajout du nom est ignorée. Voici comment nous pourrions écrire une implémentation avec passage par valeur :

```
class Widget {
public:
  void addName(std::string newName)
  {
    if ((newName.length() >= minLen) &&
        (newName.length() <= maxLen))
      {
        names.push_back(std::move(newName));
      }
  }

  ...

private:
  std::vector<std::string> names;
};
```

Cette fonction impose des coûts de construction et de destruction de `newName`, même si rien n'est ajouté à `names`. C'est un prix que les approches par référence ne demanderaient pas de payer.

Même lorsque la fonction réalise une copie inconditionnelle d'un type copiable dont le déplacement est bon marché, le passage par valeur peut ne pas convenir. En effet, une fonction a deux façons de copier un paramètre : par *construction* (c'est-à-dire construction par copie ou construction par déplacement) et par *affectation* (c'est-à-dire affectation par copie ou affectation par déplacement). `addName` utilise la construction : son paramètre `newName` est passé à `vector::push_back`, et, dans cette fonction, `newName`

Copyright © 2016 Dunod.
© Dunod – Toute reproduction non autorisée est un délit.

sert à créer un nouvel élément à la fin du `std::vector` en utilisant la construction par copie Dans le cas d'une fonction qui emploie la construction pour copier ses paramètres, l'analyse précédente est terminée : l'utilisation du passage par valeur implique le coût d'un déplacement supplémentaire pour les arguments lvalue et rvalue.

Le cas de la copie d'un paramètre par affectation est plus complexe. Supposons, par exemple, que nous ayons une classe qui représente des mots de passe. Puisqu'un mot de passe peut être modifié, elle offre un mutateur, `changeTo`. Avec une stratégie de passage par valeur, nous pouvons implémenter `Password` de la manière suivante :

```
class Password {
public:
  explicit Password(std::string pwd)     // Passage par valeur.
  : text(std::move(pwd)) {}              // Construire text.

  void changeTo(std::string newPwd)      // Passage par valeur.
  { text = std::move(newPwd); }          // Affecter text.

  ...

private:
  std::string text;                      // Texte du mot de passe.
};
```

En stockant le mot de passe sous forme d'un texte en clair, nous allons faire hurler l'équipe chargée de la sécurité des logiciels, mais ignorons cela et examinons le code suivant :

```
std::string initPwd("Supercalifragilisticexpialidocious");

Password p(initPwd);
```

Aucune surprise ici : `p.text` est construit à partir du mot de passe indiqué et, en utilisant le passage par valeur dans le constructeur, nous avons le coût d'une construction par déplacement d'un `std::string`, qui ne serait pas nécessaire avec la surcharge ou la transmission parfaite. Tout va bien.

Un utilisateur de ce programme pourrait ne pas être très satisfait du mot de passe, car « Supercalifragilisticexpialidocious » se trouve dans de nombreux dictionnaires. Il pourrait donc mener des actions qui conduiraient à l'exécution d'un code équivalent à celui :

```
std::string newPassword = "Beware the Jabberwock";

p.changeTo(newPassword);
```

Nous pouvons toujours discuter de la robustesse de ce nouveau mot de passe par rapport à l'ancien, mais c'est le problème de l'utilisateur. Le nôtre est que, en raison de la copie par affectation du paramètre `newPwd` dans `changeTo`, la stratégie de passage par valeur choisie par la fonction risque de faire exploser les coûts.

Copyright © 2016 Dunod.

Puisque l'argument passé à `changeTo` est une lvalue (`newPassword`), le constructeur de copie de `std::string` est invoqué pour construire le paramètre `newPwd`. Ce constructeur alloue de la mémoire pour stocker le nouveau mot de passe. `newPwd` est ensuite affecté par déplacement à `text`, ce qui provoque la désallocation de la mémoire occupée par `text`. Nous avons donc deux opérations de gestion de la mémoire dynamique dans `changeTo` : l'une pour allouer de la mémoire au nouveau mot de passe, l'autre pour libérer la mémoire associée à l'ancien mot de passe.

Mais, dans ce cas, puisque l'ancien mot de passe (« Supercalifragilisticexpialidocious ») est plus long que le nouveau (« Beware the Jabberwock »), il est inutile d'allouer ou de libérer de la mémoire. Avec l'approche par surcharge, il est probable qu'aucune de ces actions n'aurait lieu :

```
class Password {
public:
  ...

  void changeTo(const std::string& newPwd)      // Surcharge pour
  {                                             // les lvalues.

    text = newPwd;      // Réutilisation possible de la mémoire de text
                        // si text.capacity() >= newPwd.size().
  }

  ...

private:
  std::string text;                             // Comme précédemment.
};
```

Dans ce scénario, le coût du passage par valeur comprend une allocation et une désallocation mémoire supplémentaires. Ces coûts secondaires seront très supérieurs à celui d'une opération de déplacement d'un `std::string`.

Il est intéressant de noter que, si l'ancien mot de passe est plus court que le nouveau, il est généralement impossible d'éviter l'allocation-désallocation au cours de l'affectation et, dans ce cas, le passage par valeur affiche la même vitesse d'exécution que le passage par référence. Le coût de la copie d'un paramètre par affectation peut donc dépendre de la valeur des objets concernés par l'affectation ! Ce type d'analyse vaut pour tous les types de paramètres qui contiennent des valeurs stockées dans une mémoire allouée dynamiquement. Tous les types ne sont pas concernés, mais ils sont nombreux, notamment `std::string` et `std::vector`.

En général, cette augmentation potentielle du coût s'applique uniquement lors du passage d'arguments lvalue, car l'allocation et la désallocation de la mémoire sont habituellement nécessaires uniquement pour les véritables opérations de copie (en dehors des déplacements). Pour les arguments rvalue, les déplacements font quasiment toujours l'affaire.

Par conséquent, le coût supplémentaire du passage par valeur pour les fonctions qui copient un paramètre par affectation dépend du type du paramètre passé, du rapport

Copyright © 2016 Dunod.
© Dunod – Toute reproduction non autorisée est un délit.

entre le nombre d'arguments lvalue et rvalue, de l'utilisation de la mémoire allouée dynamiquement par le type et, le cas échéant, de l'implémentation des opérateurs d'affectation du type et de la probabilité que la mémoire associée à la cible de l'affectation soit au moins aussi vaste que celle associée à la source de l'affectation. Pour un `std::string`, il dépend également de la mise en œuvre de l'optimisation des petites chaînes (SSO, *small string optimization* ; voir le conseil 29) et de la possibilité de placer les valeurs affectées dans le tampon SSO.

Nous l'avions dit, lorsque des paramètres sont copiés par affectation, l'analyse du coût du passage par valeur est complexe. Habituellement, l'approche la plus pratique consiste à adopter la surcharge ou les références universelles, et d'employer le passage par valeur uniquement s'il est démontré que le code obtenu se révèle efficace avec le type de paramètre concerné.

Lorsque le logiciel doit s'exécuter le plus rapidement possible, le passage par valeur ne sera probablement pas une stratégie viable, car les déplacements même bon marché devront certainement être évités. Par ailleurs, il n'est pas toujours facile de connaître le nombre de déplacements impliqués. Dans l'exemple de `Widget::addName`, un passage par valeur ne comprend qu'une seule opération de déplacement supplémentaire. Mais supposons que `Widget::addName` ait appelé `Widget::validateName` et que cette fonction utilise également le passage par valeur. (On peut imaginer qu'elle a une bonne raison de toujours copier son paramètre, par exemple pour le stocker dans une structure de données qui contient toutes les valeurs validées.) Et supposons que `validateName` ait appelé une troisième fonction qui utilise également le passage par valeur...

Vous comprenez où cela nous mène. Lorsque nous sommes en présence d'une chaîne d'appels de fonctions, chacune employant le passage par valeur car « il ne coûte qu'un déplacement supplémentaire », le prix total des appels peut ne pas être tolérable. Avec un passage de paramètres par référence, les chaînes d'appels ne conduisent pas à cette accumulation d'un surcoût.

Il existe un autre problème, sans lien avec les performances, qu'il est bon de ne pas oublier. Contrairement au passage par référence, le passage par valeur est sujet au problème de *slicing*. Puisque ce sujet a été largement traité en C++98, nous n'allons pas nous y attarder. Sachez simplement que si une fonction accepte un paramètre dont le type est une classe de base *ou tout type qui en dérive*, il ne faut pas déclarer un paramètre de ce type passé par valeur car cela « couperait » les caractéristiques de la classe dérivée pour tout objet de type dérivé passé :

```
class Widget { … };                          // Classe de base.

class SpecialWidget: public Widget { … };    // Classe dérivée

void processWidget(Widget w);   // Fonction pour tout type de Widget,
                                // y compris les types dérivé ;
…                               // problème de slicing.

SpecialWidget sw;
```

Copyright © 2016 Dunod.

```
...

processWidget(sw);              // processWidget voit un Widget,
                                // non un SpecialWidget !
```

Si vous n'êtes pas familier de ce problème, les moteurs de recherche et Internet vous fourniront toutes les informations nécessaires. Vous découvrirez que l'existence de ce problème explique en partie (avant même les questions d'efficacité) la mauvaise réputation du passage par valeur en C++98. Ce n'est pas sans raison que l'une des premières choses que vous avez probablement apprises sur la programmation en C++ était d'éviter de passer par valeur un objet de type défini par l'utilisateur.

C++11 n'apporte aucun changement fondamental à cet égard. De façon générale, le passage par valeur entraîne une baisse des performances et peut toujours conduire au problème de *slicing*. En revanche, la nouveauté de C++11 réside dans la distinction entre les arguments lvalue et rvalue. L'implémentation de fonctions qui tirent profit de la sémantique de déplacement pour les rvalues de type copiable impose la surcharge ou les références universelles, mais ces deux approches ont des inconvénients. Pour le cas particulier où des types copiables, au déplacement bon marché, sont passés à des fonctions qui en effectuent toujours une copie, et où le problème de *slicing* est absent, un passage par valeur peut représenter une alternative facile à mettre en place et pratiquement aussi efficace que les approches par référence, sans souffrir de leurs inconvénients.

À retenir

- Pour les paramètres copiables, peu coûteux à déplacer et toujours copiés, le passage par valeur peut être quasiment aussi efficace que le passage par référence, il est plus facile à implémenter et il peut générer un code objet plus concis.
- La copie des paramètres par construction peut être beaucoup plus coûteuse que leur copie par affectation.
- Le passage par valeur souffre du problème de *slicing* et ne convient donc pas aux paramètres dont le type est une classe de base.

CONSEIL N° 42. ENVISAGER LE PLACEMENT PLUTÔT QUE L'INSERTION

Supposons que nous ayons un conteneur de `std::string`. Il semblerait logique que le type d'un objet ajouté et passé à une fonction d'insertion (c'est-à-dire `insert`, `push_front`, `push_back` ou, pour `std::forward_list`, `insert_after`) soit un `std::string`. C'est en effet le type des éléments du conteneur.

Aussi logique que cela puisse être, ce n'est pas toujours vrai. Prenons le code suivant :

Copyright © 2016 Dunod.
© Dunod – Toute reproduction non autorisée est un délit.

```
std::vector<std::string> vs;        // Conteneur de std::string.

vs.push_back("xyzzy");              // Ajouter une chaîne littérale.
```

Le conteneur mémorise des `std::string`, mais nous avons une chaîne de caractères littérale, c'est-à-dire une suite de caractères placés entre guillemets, que nous passons à `push_back`. Une chaîne littérale n'est pas un `std::string` et l'argument que nous passons à `push_back` n'a donc pas le même type que les éléments du conteneur.

La méthode `push_back` de `std::vector` est surchargée pour les lvalues et les rvalues :

```
template <class T,                          // Du C++11 standard.
          class Allocator = allocator<T>>
class vector {
public:
  …
  void push_back(const T& x);               // Insérer une lvalue.
  void push_back(T&& x);                    // Insérer une rvalue.
  …
};
```

Avec l'appel

```
vs.push_back("xyzzy");
```

le compilateur détecte une incohérence entre le type de l'argument (`const char[6]`) et le type du paramètre pris par `push_back` (une référence à un `std::string`). Il résout ce problème en générant du code qui crée un objet `std::string` temporaire à partir de la chaîne littérale, puis en passant cet objet temporaire à `push_back`. Autrement dit, il fait comme si l'appel avait été écrit de la manière suivante :

```
vs.push_back(std::string("xyzzy"));   // Créer un std::string temporaire
                                      // et le passer à push_back.
```

Le code compile et s'exécute, tout le monde est satisfait. Tout le monde, à l'exception du mordu des performances car il sait que ce code n'est pas aussi efficace qu'il le devrait.

Pour créer un nouvel élément dans un conteneur de `std::string`, un constructeur de `std::string` doit évidemment être invoqué, mais le code précédent en appelle non pas un mais deux, sans mentionner l'invocation du destructeur de `std::string`. Voici ce qui se produit à l'exécution lors de l'appel à `push_back` :

1. Un objet `std::string` temporaire est créé à partir de la chaîne littérale `"xyzzy"` ; il n'a pas de nom, alors appelons-le *temp*. La construction de *temp* correspond à la première construction d'un `std::string`. Puisqu'il s'agit d'un objet temporaire, *temp* est une rvalue.

Copyright © 2016 Dunod.

2. *temp* est passé à la surcharge de `push_back` pour les rvalues, où il est lié au paramètre `x` de type référence rvalue. Une copie de `x` est ensuite construite en mémoire pour le `std::vector`. Cette construction, la *seconde*, correspond à la création d'un nouvel objet dans le `std::vector`. (Le constructeur utilisé pour copier `x` dans le `std::vector` est le constructeur de déplacement car `x`, étant une référence rvalue, est converti en rvalue avant d'être copié. Pour de plus amples informations sur la conversion en rvalues des paramètres de type référence rvalue, consultez le conseil 25.)
3. Immédiatement après le retour de `push_back`, *temp* est détruit, ce qui invoque le destructeur de `std::string`.

Le mordu de performances remarque que s'il était possible de prendre la chaîne littérale et de la passer directement au code de l'étape 2 qui construit l'objet `std::string` dans le `std::vector`, la construction et la destruction de *temp* seraient évitées. L'efficacité serait maximale, et ce fanatique des performances serait satisfait.

Puisque vous êtes un programmeur C++, il y a de fortes chances que vous soyez obsédé par les performances. Et si ce n'est pas le cas, vous leur accordez certainement une petite attention. Nous sommes donc heureux de vous annoncer qu'une solution permet d'obtenir une efficacité maximale lors de l'appel à `push_back`. En réalité, il ne s'agit pas d'un appel à `push_back` ; cette fonction n'est pas la bonne, il faut utiliser `emplace_back`.

`emplace_back` réalise exactement ce que nous souhaitons : elle se sert des arguments passés pour construire un `std::string` directement dans le `std::vector`. Plus aucun objet temporaire n'intervient :

```
vs.emplace_back("xyzzy");    // Construire un std::string directement
                             // dans vs à partir de "xyzzy".
```

`emplace_back` se fonde sur la transmission parfaite et, tant que nous évitons ses limitations (voir le conseil 30), nous pouvons passer à `emplace_back` n'importe quel nombre d'arguments avec n'importe quelle combinaison de types. Par exemple, si nous souhaitons créer un `std::string` dans `vs` *via* le constructeur de `std::string` qui prend un caractère et un nombre de répétitions, voici ce que nous devons écrire :

```
vs.emplace_back(50, 'x');    // Insérer un std::string constitué
                             // de 50 caractères 'x'.
```

`emplace_back` est disponible avec tous les conteneurs standard qui prennent en charge `push_back`. De même, tous les conteneurs standard qui prennent en charge `push_front` disposent de `emplace_front`. Et tous ceux qui offrent `insert` (c'est-à-dire tous les conteneurs sauf `std::forward_list` et `std::array`) reconnaissent `emplace`. Les conteneurs associatifs proposent `emplace_hint` pour compléter leurs fonctions `insert` qui prend un itérateur « indice », et `std::forward_list` apporte `emplace_after` pour aller avec `insert_after`.

Les fonctions de placement sont plus efficaces que les fonctions d'insertion car elles disposent d'une interface plus souple. Les fonctions d'insertion prennent des *objets à*

Copyright © 2016 Dunod.
© Dunod – Toute reproduction non autorisée est un délit.

insérer, tandis que les fonctions de placement prennent des *arguments de construction des objets à insérer*. Cette différence leur permet d'éviter la création et la destruction d'objets temporaires, dont les fonctions d'insertion peuvent avoir besoin.

Puisque nous pouvons passer à une fonction de placement un argument du type des éléments du conteneur (la fonction effectue alors une construction par copie ou déplacement), le placement peut être employé même lorsqu'une fonction d'insertion n'a pas besoin d'un objet temporaire. Dans ce cas, l'insertion et le placement réalisent fondamentalement la même opération. Prenons par exemple ce code :

```
std::string queenOfDisco("Donna Summer");
```

Les deux appels suivants sont valides et mènent au même résultat sur le conteneur :

```
vs.push_back(queenOfDisco);         // Construire par copie queenOfDisco
                                    // à la fin de vs.

vs.emplace_back(queenOfDisco);      // Idem.
```

Les fonctions de placement ont donc les mêmes utilisations que les fonctions d'insertion, mais elles travaillent parfois plus efficacement et, tout au moins en théorie, ne devraient jamais être moins efficaces. Dans ce cas, pourquoi ne pas les employer systématiquement ?

Parce que, si en théorie il n'y a pas de différence entre la théorie et la pratique, en pratique il y en a une. Avec les implémentations actuelles de la bibliothèque standard, il existe des situations dans lesquelles le placement est plus rapide que l'insertion mais, malheureusement, il en existe d'autres où les fonctions d'insertion sont plus efficaces. Ces cas ne sont pas faciles à caractériser car ils dépendent du type des arguments passés, des conteneurs employés, des emplacements dans le conteneur où se fait l'insertion ou le placement, de la sûreté vis-à-vis des exceptions des constructeurs des types contenus, et, pour les conteneurs qui interdisent les doublons (c'est-à-dire `std::set`, `std::map`, `std::unordered_set`, `std::unordered_map`), du fait que la valeur ajoutée est, ou non, déjà présente. Le conseil habituel sur les performances s'applique donc : pour déterminer qui du placement ou de l'insertion est le plus rapide, il faut les tester.

Cette conclusion est évidemment peu satisfaisante et vous serez content d'apprendre qu'une heuristique peut aider à identifier les situations dans lesquelles les fonctions de placement ont toutes les chances de convenir. Si les conditions suivantes sont vérifiées, le placement sera presque à coup sûr plus efficace que l'insertion :

- **La valeur à ajouter est non pas affectée mais construite dans le conteneur.** L'exemple donné au début de ce conseil (ajouter un `std::string` de valeur `"xyzzy"` à un `std::vector`) a montré l'ajout d'une valeur à la fin de `vs` – un emplacement où il n'existe encore aucun objet. La nouvelle valeur doit donc être construite dans le `std::vector`. Si nous modifions l'exemple de sorte que le nouveau `std::string` aille dans un emplacement déjà occupé par un objet, l'histoire est différente. Prenons le code suivant :

Copyright © 2016 Dunod.

```
std::vector<std::string> vs;          // Comme précédemment.

…                                     // Ajouter des éléments à vs.

vs.emplace(vs.begin(), "xyzzy");      // Ajouter "xyzzy" au
                                      // début de vs.
```

Pour ce code, peu d'implémentations construiront le `std::string` ajouté dans la zone de mémoire occupée par `vs[0]`. Elles effectueront plutôt une affectation par déplacement de la valeur à sa place. Mais l'affectation par déplacement a besoin d'un objet à déplacer. Autrement dit, un objet temporaire doit être créé pour servir de source au déplacement. Puisque le principal avantage du placement par rapport à l'insertion réside dans l'absence de création et de destruction d'objets temporaires, l'ajout de la valeur dans le container *via* une affectation remet en question l'intérêt du placement.

Malheureusement, l'ajout par construction ou par affectation d'une valeur au conteneur est généralement un choix laissé aux développeurs. Cependant, une fois encore, une heuristique peut nous aider. Les conteneurs orientés nœuds utilisent pratiquement toujours la construction pour ajouter de nouvelles valeurs et la plupart des conteneurs standard s'articulent autour des nœuds. Les seules exceptions sont `std::vector`, `std::deque` et `std::string`. (`std::array` fait également exception, mais il ne prend pas en charge l'insertion ou le placement, et ne nous intéresse donc pas.) Avec les conteneurs qui ne sont pas basés sur des nœuds, nous pouvons compter sur `emplace_back` pour employer une construction à la place de l'affectation afin de placer la nouvelle valeur. Il en va de même pour `emplace_front` de `std::deque`.

- **Le ou les types des arguments passés diffèrent de celui des éléments du conteneur**. Rappelons que l'avantage du placement par rapport à l'insertion découle généralement du fait que son interface ne nécessite aucune création et destruction d'un objet temporaire lorsque les arguments passés sont d'un type autre que celui des éléments du conteneur. Lorsqu'un objet de type `T` doit être ajouté à un *`container`*`<T>`, il n'y a aucune raison de supposer que le placement sera plus rapide que l'insertion, car aucun objet temporaire n'a besoin d'être créé pour satisfaire l'interface d'insertion.
- **Le conteneur ne devrait pas refuser la nouvelle valeur sous prétexte qu'elle forme un doublon**. Autrement dit, soit le conteneur accepte les doublons, soit la plupart des valeurs ajoutées seront uniques. Ce point est important car, pour détecter qu'une valeur se trouve déjà dans le conteneur, les implémentations du placement créent souvent un nœud avec cette valeur afin de la comparer aux nœuds existants dans le conteneur. Si la valeur ajoutée est absente du conteneur, le nœud est lié aux autres. En revanche, si la valeur est déjà présente, le placement est annulé et le nœud est détruit, gaspillant alors sa construction et sa destruction. De tels nœuds sont plus souvent créés par les fonctions de placement que celles d'insertion.

Copyright © 2016 Dunod.
© Dunod – Toute reproduction non autorisée est un délit.

Les appels suivants, déjà présentés dans ce conseil, satisfont à tous les critères précédents. Ils s'exécutent plus rapidement que les appels correspondant à `push_back`.

```
vs.emplace_back("xyzzy");    // Construire une nouvelle valeur à la fin
                             // du conteneur ; le paramètre n'est pas
                             // du type des éléments du conteneur ;
                             // le conteneur ne refuse pas les doublons.

vs.emplace_back(50, 'x');    // Idem.
```

Avant de décider d'utiliser les fonctions de placement, deux autres problèmes doivent être pris en compte. Le premier concerne la gestion des ressources. Supposons que nous ayons un conteneur de `std::shared_ptr<Widget>` :

```
std::list<std::shared_ptr<Widget>> ptrs;
```

Nous voulons ajouter un `std::shared_ptr` dont la libération se fera à l'aide d'un supprimeur personnalisé (voir le conseil 19). Le conseil 21 explique que nous devons utiliser autant que possible `std::make_shared` pour créer des `std::shared_ptr`, mais il note également que certaines situations nous en empêchent, notamment lorsque nous voulons spécifier un supprimeur personnalisé. Dans ce cas, nous devons employer directement `new` pour obtenir le pointeur brut qui sera géré par le `std::shared_ptr`.

Supposons que la fonction suivante serve de supprimeur personnalisé :

```
void killWidget(Widget* pWidget);
```

Le code fondé sur une fonction d'insertion peut s'écrire ainsi :

```
ptrs.push_back(std::shared_ptr<Widget>(new Widget, killWidget));
```

Voici une autre possibilité équivalente :

```
ptrs.push_back({ new Widget, killWidget });
```

Dans les deux versions, un `std::shared_ptr` temporaire doit être construit avant l'appel à `push_back`. Puisque le paramètre de `push_back` est une référence à un `std::shared_ptr`, il doit exister un `std::shared_ptr` auquel ce paramètre fera référence.

La création de ce `std::shared_ptr` temporaire serait évitée avec `emplace_back`, mais, dans ce cas, les bénéfices qu'il apporte outrepassent largement ses coûts. Examinons la séquence d'événements suivante :

1. Dans les appels précédents, un objet `std::shared_ptr<Widget>` temporaire est construit de façon à mémoriser le pointeur brut renvoyé par « `new Widget` ». Nommons *temp* cet objet.

Copyright © 2016 Dunod.

2. `push_back` prend *temp* par référence. Pendant l'allocation d'un nœud de liste pour stocker une copie de *temp*, une exception signalant un manque de mémoire est levée.
3. L'exception se propage hors de `push_back` et *temp* est détruit. Puisqu'il s'agit du seul `std::shared_ptr` qui fait référence au `Widget` dont il a la charge, il libère automatiquement ce `Widget` en invoquant dans ce cas `killWidget`.

Malgré l'exception, nous ne constatons aucune fuite de mémoire : le `Widget` créé par « `new Widget` » dans l'appel à `push_back` est libéré par le destructeur de `std::shared_ptr`, qui avait été créé pour le gérer (*temp*). Tout va bien.

Examinons à présent ce qui se passe si nous appelons `emplace_back` à la place de `push_back` :

```
ptrs.emplace_back(new Widget, killWidget);
```

1. Le pointeur brut obtenu par « `new Widget` » est transmis de façon parfaite à l'endroit de `emplace_back` où un nœud de liste doit être alloué. Cette allocation échoue et une exception liée à un manque de mémoire est lancée.
2. L'exception se propage en dehors de `emplace_back` et le pointeur brut, qui était la seule manière d'accéder au `Widget` sur le tas, est perdu. Ce `Widget` (et toutes ses ressources) représente une fuite de mémoire.

Dans ce scénario, rien ne va plus et la faute n'incombe pas à `std::shared_ptr`. Le même type de problème peut survenir en cas d'utilisation de `std::unique_ptr` et d'un supprimeur personnalisé. Fondamentalement, l'efficacité des classes de gestion de ressources, comme `std::shared_ptr` et `std::unique_ptr`, se fonde sur des ressources (comme les pointeurs bruts fournis par `new`) transmises *immédiatement* aux constructeurs des objets de gestion. Les fonctions comme `std::make_shared` et `std::make_unique` y procèdent de façon automatique et c'est l'une des raisons pour lesquelles elles sont si importantes.

Dans les appels aux fonctions d'insertion des conteneurs qui stockent des objets de gestion de ressources (par exemple `std::list<std::shared_ptr<Widget>>`), les types des paramètres des fonctions permettent généralement de garantir que rien n'intervient entre l'acquisition d'une ressource (par exemple une utilisation de `new`) et la construction de l'objet qui gère cette ressource. Dans les fonctions de placement, la transmission parfaite reporte la création des objets de gestion de ressources jusqu'à leur construction dans la mémoire du conteneur et, pendant ce temps, des exceptions peuvent être levées et conduire à des fuites de ressources. Tous les conteneurs standard sont sujets à ce problème. Lorsque des conteneurs d'objets de gestion de ressources sont employés, l'utilisation d'une fonction d'emplacement à la place de sa version d'insertion doit se faire avec prudence, car la meilleure efficacité du code obtenue risque de coûter une sûreté moindre vis-à-vis des exceptions.

Il est préférable d'éviter de passer des expressions comme « `new Widget` » à `emplace_back` ou à `push_back`, voire même à n'importe quelle autre fonction, car, le conseil 21 l'explique, cela ouvre la porte à des problèmes de sûreté vis-à-vis des

Copyright © 2016 Dunod.
© Dunod – Toute reproduction non autorisée est un délit.

exceptions, comme celui que nous venons de décrire. Pour fermer cette porte, nous devons transformer dans une instruction indépendante le pointeur fourni par « `new Widget` » en un objet de gestion de ressources. Celui-ci doit ensuite être transmis comme une rvalue à la fonction à laquelle nous voulions à l'origine passer « `new Widget` ». (Le conseil 21 détaille cette technique.) Voici donc comment écrire le code qui utilise `push_back` :

```
std::shared_ptr<Widget> spw(new Widget,    // Créer un Widget qui sera
                            killWidget);   // géré par spw.

ptrs.push_back(std::move(spw));            // Ajouter spw comme une rvalue.
```

La version fondée sur `emplace_back` est comparable :

```
std::shared_ptr<Widget> spw(new Widget, killWidget);
ptrs.emplace_back(std::move(spw));
```

Dans les deux cas, nous avons les coûts de création et de destruction de `spw`. Le choix du placement à la place de l'insertion était motivé par l'absence du coût de création d'un objet temporaire du type des éléments du conteneur. Bien que `spw` le soit conceptuellement, les fonctions de placement auront du mal à surpasser les fonctions d'insertion lorsque nous ajoutons des objets de gestion de ressources à un conteneur en suivant l'approche qui garantit que rien n'intervient entre l'acquisition d'une ressource et sa conversion en un objet de gestion de ressources.

L'autre aspect problématique des fonctions de placement concerne leur interaction avec les constructeurs `explicit`. Pour rendre honneur à la prise en charge des expressions régulières dans C++11, supposons que nous créions un conteneur d'objets d'expressions régulières :

```
std::vector<std::regex> regexes;
```

Distrait par les querelles de nos collègues sur le nombre idéal de consultations quotidiennes du compte Facebook d'un ami, nous écrivons par mégarde le code suivant :

```
regexes.emplace_back(nullptr);      // Ajouter nullptr au conteneur
                                    // d'expressions régulières ?
```

Nous ne remarquons pas notre erreur pendant la saisie et le compilateur accepte ce code sans broncher. Nous perdons alors beaucoup de temps à son débogage. À un moment donné, nous découvrons que nous avons inséré un pointeur nul dans notre conteneur d'expressions régulières. Mais comment est-ce possible ? Puisque les pointeurs ne sont pas des expressions régulières, le compilateur n'accepte pas le code suivant :

```
std::regex r = nullptr;             // Erreur ! Échec de la compilation.
```

Copyright © 2016 Dunod.

Il est intéressant de noter que la compilation échoue également si nous remplaçons `emplace_back` par `push_back` :

```
regexes.push_back(nullptr);       // Erreur ! Échec de la compilation.
```

Le curieux comportement auquel nous sommes confrontés vient du fait que les objets `std::regex` peuvent être construits à partir de chaînes de caractères. C'est pour cette raison que nous pouvons écrire le code utile suivant :

```
std::regex upperCaseWord("[A-Z]+");
```

La création d'un `std::regex` à partir d'une chaîne de caractères peut avoir un coût élevé. De façon à réduire la probabilité qu'une telle dépense soit faite par mégarde, le constructeur de `std::regex` qui prend un pointeur `const char*` est déclaré `explicit`. Voilà pourquoi la compilation des lignes suivantes échoue :

```
std::regex r = nullptr;           // Erreur ! Échec de la compilation.

regexes.push_back(nullptr);       // Erreur ! Échec de la compilation.
```

Dans les deux cas, nous demandons une conversion implicite depuis un pointeur vers un `std::regex`, mais la déclaration `explicit` de ce constructeur l'empêche.

En revanche, dans l'appel à `emplace_back`, nous ne prétendons pas passer un objet `std::regex`. À la place, nous passons un *argument de construction* d'un objet `std::regex`. Cela ne constitue pas une demande de conversion implicite mais équivaut au code suivant :

```
std::regex r(nullptr);            // Réussite de la compilation.
```

Si la réussite de la compilation ne soulève pas chez vous un grand enthousiasme, vous avez raison car ce code affiche un comportement indéfini. Le constructeur de `std::regex` qui prend un pointeur `const char*` exige que la chaîne désignée représente une expression régulière valide, ce qui n'est pas le cas d'un pointeur nul. Si nous écrivons et compilons un tel code, le mieux que nous puissions espérer est un plantage du programme à l'exécution. Si nous manquons de chance, nous allons passer beaucoup de temps auprès de notre débogueur.

Mettons un instant de côté `push_back`, `emplace_back` et le débogueur pour noter combien des syntaxes d'initialisation très proches conduisent à des résultats différents :

```
std::regex r1 = nullptr;          // Erreur ! Échec de la compilation.

std::regex r2(nullptr);           // Réussite de la compilation.
```

Selon la terminologie officielle de la norme, la syntaxe employée pour initialiser `r1` (avec le signe égal) correspond à une *initialisation par copie*. À l'opposé, la syntaxe

Copyright © 2016 Dunod.
© Dunod – Toute reproduction non autorisée est un délit.

d'initialisation de `r2` (avec les parenthèses, qui peuvent aussi être remplacées par des accolades) correspond à une *initialisation directe*. L'initialisation par copie ne peut pas se servir des constructeurs `explicit` ; l'initialisation directe y est autorisée. C'est pourquoi le code d'initialisation de `r1` ne compile pas, contrairement à celui de `r2`.

Revenons à `push_back` et à `emplace_back`, et, plus généralement, à la comparaison entre les fonctions d'insertion et les fonctions de placement. Puisque les fonctions de placement se fondent sur l'initialisation directe, elles peuvent employer les constructeurs `explicit`. Ce n'est pas le cas des fonctions d'insertion qui utilisent l'initialisation par copie. Nous avons donc les comportements suivants :

```
regexes.emplace_back(nullptr);    // Réussite de la compilation.
                                  // L'initialisation directe autorise
                                  // l'utilisation du constructeur
                                  // explicit de std::regex qui
                                  // prend un pointeur.

regexes.push_back(nullptr);       // Erreur ! L'initialisation par copie
                                  // interdit ce constructeur.
```

En conclusion, lorsque nous utilisons une fonction de placement, il faut faire particulièrement attention à passer des arguments corrects, car même les constructeurs `explicit` seront examinés par le compilateur lorsqu'il essaiera de trouver une interprétation valide de notre code.

À retenir

- En principe, les fonctions de placement sont plus efficaces que les fonctions d'insertion équivalentes, mais elles ne devraient jamais être moins efficaces.
- En pratique, elles seront plus rapides lorsque (1) la valeur ajoutée est non pas affectée mais construite dans le conteneur, (2) les types des arguments passés diffèrent du type des éléments du conteneur, et (3) le conteneur ne refuse pas les doublons ajoutés.
- Les fonctions de placement peuvent effectuer des conversions de type qui seraient rejetées par les fonctions d'insertion.

Copyright © 2016 Dunod.

Index

Symboles

&&, signification 160
= (signe égal), affectation contre initialisation 50
0 (zéro)
  surcharge et 58
  templates et 60
  type de 58

A

accolades, initialisation à 50
alias
  de templates 63
  déclaration *Voir* déclarations d'alias
apostrophe, séparateur de milliers 247
arguments
  champs de bits 208
  liés et déliés 233
`auto` 37
  avantages 38, 41
  classes proxy 43, 46
  déduction de type 18
  déduction de type de retour et initialiseurs à accolades 21
  initialiseurs à accolades 21
  lisibilité du code 42
  maintenance 42
  réduction de références et 196
  références universelles 163
  remaniement 42
  `std::function` pour les objets fonctions 39
  `std::initializer_list` 21
  type de retour arrière 25
avertissements du compilateur *Voir* compilateur, avertissements

B

blocs de contrôle 125
  définition 125
  `std::shared_ptr` et 127
  taille 130
Boost.TypeIndex 33
boucles temporisées 243

C

C avec des classes 84
C++ *Concurrency in Action* (livre) 252
C++03, définition 2
C++11, définition 2
C++14, définition 2
C++98
  définition 2
  spécifications d'exceptions 88

Copyright © 2016 Dunod.
© Dunod – Toute reproduction non autorisée est un délit.

captures
modes par défaut 212
par référence 213
pointeur `this` et 216
captures généralisées 219
définition 220, 221
captures par déplacement 219
émulation avec `std::bind` 222, 234
expression lambda et 234
captures par valeur 215
pointeurs et 215
problèmes 215
`std::move` et 277
`cbegin` 86
`cend` 86
`c++filt` 31
champs de bits en arguments 208
chargements redondants, définition 271
classes
proxy 45, 46
RAII pour les objets `std::thread` 263
code
exemples *Voir* exemples de classes/templates ; exemples de fonctions/templates
générique, opérations de déplacement et 201
inline dans les expressions lambda contre `std::bind` 231
code, réorganisation
`std::atomic` et 267
`volatile` et 269
cohérence
lâche, définition 268
séquentielle, définition 268
communication d'un événement
avec un futur 261
coût et efficacité de l'interrogation 260
drapeaux booléens 259
variables de condition 257
compilateur, avertissements 80
`noexcept` et 94
redéfinition de fonction virtuelle et 80
comportement indéfini, définition 6
compteur
de références, définition 123
faible, définition 141
concurrence, test de joignabilité sur `std::thread` 250
condvar *Voir* variables de condition
`const`
contre `constexpr` 96
fonctions membres `const` et sûreté vis-à-vis des threads 101
pointeurs et déduction de type 14
propagation de `const`, définition 204
`const T&&` 162
`constexpr`
conception d'interface et 100
contre `const` 96
objets `constexpr` 95
`constexpr` (fonctions) 96
arguments à l'exécution et 97
restrictions 98
`const_iterator`
contre `iterator` 84
conversion en `iterator` 85
constructeurs
appels avec des accolades ou des parenthèses 52
`explicit` 292
`explicit` et fonctions d'insertion 292
références universelles et 175, 184
contrats
étendus contre restreints 93
étendus, définition 93
restreints, définition 93
conversions de type
conditionnelles contre inconditionnelles 157
`std::move` contre `std::forward` 154

Copyright © 2016 Dunod.

conversions restrictives, définition 51
copie
    d'un objet, définition 4
    profonde, définition 150
    superficielle, définition 150
CRTP (*Curiously Recurring Template Pattern*) 129
Curiously Recurring Template Pattern (CRTP) 129

D

déclaration 5
déclarations d'alias
    alias de templates et 63
    contre `typedef` 62
    définition 62
    réduction de référence et 196
`decltype` 23
    expressions de retour 29
    paramètres `auto&&` dans les expressions lambda 225
    réduction de référence et 197
    traitement des noms et traitement des expressions 28
`decltype(auto)` 26
déduction de type *Voir aussi* templates, déduction de type, 30
    `emplace_back` et 162
    pour `auto` 18
    références universelles et 161
`=default` 109, 149, 252
définition 5
définition des termes
    alias de template 63
    argument d'une fonction 4
    bloc de contrôle 125
    C++03 2
    C++11 2
    C++14 2
    C++98 2
    capture généralisée 220, 221
    chargements redondants 271
classe de fermeture 212
classe RAII 248
classe template 5
cohérence lâche 268
cohérence séquentielle 268
comportement indéfini 6
compteur de références 123
compteur faible 141
contrats étendus 93
contrats restreints 93
conversions restrictives 51
copie d'un objet 4
copie profonde 150
copie superficielle 150
CRTP (*Curiously Recurring Template Pattern*) 129
déclaration 5
déclaration d'alias 62
définition 5
demande excédentaire 239
durée de stockage statique 218
élision de copie 170
entrées-sorties mappées en mémoire 270
`enum` délimités 67
`enum` non délimités 67
état partagé 254
expression constante entière 95
expression lambda 5, 211
expression lambda générique 225
fermeture 5, 212
fonction `make` 136
fonction membre spéciale 106
fonction supprimée 74
fonction template 5
fonctionnalité obsolète 6
garantie forte 4
garantie minimale 4
idiome Pimpl 144
initialisation à accolades 50
initialisation uniforme 50
`lhs` 3

© Dunod – Toute reproduction non autorisée est un délit.
Copyright © 2016 Dunod.

lvalue 2
mauvaise odeur du code 258
mémoire locale de thread 243
most vexing parse 51
mot clé contextuel 81
neutralité envers les exceptions 92
NRVO (*named return value optimization*) 170
objet fonction 4
objet invocable 4
objet RAII 248
opération de copie 3
opération de déplacement 3
optimisation de la valeur de retour 170
optimisation de la valeur de retour nommée 170
optimisation des petites chaînes 199
`override` 78
paramètre d'une fonction 4
pointeur brut 6
pointeur intelligent 6
pointeur pendouillant 131
programmation multitâche 238
programmation multithread 237
propagation de `const` 204
propriétaire d'une ressource 115
propriété exclusive 117
propriété partagée 123
qualificatif de référence 79
réduction de référence 193
Règle des trois 108
réveils intempestifs 259
`rhs` 3
rvalue 2
RVO (*return value optimization*) 170
sémantique du déplacement 153
signature d'une fonction 6
small string optimization (SSO) 199
`std::thread` joignable 246
`std::thread` non joignable 246
`std::weak_ptr` périmé 132
stockages morts 271
supprimeur personnalisé 118
sûreté vis-à-vis des exceptions 4
tableaux, dégradation 15
tag dispatching 183
template actif 184
template de classe 5
template de fonction 5
template inactif 184
thread interruptible 251
thread logiciel 238
thread matériel 238
thread système 238
TLS (*thread local storage*) 243
traduction 95
transmission de paramètres 201
transmission parfaite 4, 153, 202
type de retour arrière 25
type dépendant 64
type incomplet 145
type littéral 98
type non dépendant 64
type réservé au déplacement 103, 117
`Widget` 3
`=delete` *Voir* fonctions supprimées
demande excédentaire, définition 239
déplacement, sémantique
Voir sémantique du déplacement
destructeurs
lien avec les opérations de copie et la gestion de ressources 108
par défaut 109, 148
virtuels par défaut 109
drapeaux booléens et communication d'un événement 259
durée de stockage statique, définition 218

E

Einstein, théorie de relativité générale 164

Copyright © 2016 Dunod.

élision de copie, définition 170
entrées-sorties mappées en mémoire, définition 270
enum
  conversions implicites et 67
  déclaration anticipée 68
  délimités contre enum non délimités 67
  délimités, définition 67
  dépendances de compilation et 69
  non délimités contre enum délimités 67
  non délimités, définition 67
  std::get et 71
  std::tuples et 71
  type sous-jacent 68
énumérations
  classes enum *Voir* délimitées
  délimitées contre non délimitées 67
  délimitées, définition 67
  non délimitées, définition 67
errata 7
état partagé
  compteur de références dans 254
  définition 254
  futur, comportement du destructeur 254
exceptions
  spécifications 88
  sûreté vis-à-vis des, définition 4
exemples de classes/templates *Voir aussi* std::, 9
  Base 78, 110
  Bond 117
  Derived 78, 80
  Investment 117, 120
  IPv4Header 208
  IsValAndArch 221
  MyAllocList 63
  MyAllocList<Wine> 64
  Password 282, 283
  Person 175, 178, 180, 184, 186, 188, 190
  Point 24, 98, 100, 103
  Polynomial 101
  PolyWidget 234
  RealEstate 117
  ReallyBigType 142
  SomeCompilerGeneratedClassName 225
  SpecialPerson 178, 186
  SpecialWidget 284
  std::add_lvalue_reference 65
  std::basic_ios 74
  std::get 252
  std::pair 91
  std::remove_const 65
  std::remove_reference 65
  std::string 156
  std::vector 24, 162, 286
  std::vector<bool> 46
  Stock 117
  StringTable 110
  struct Point 24
  TD 30
  ThreadRAII 249, 252
  Warning 82
  Widget 3, 5, 50, 52, 63, 77, 79, 82, 104, 105, 107, 109, 112, 128, 145, 151, 158, 164, 166, 196, 204, 215, 220, 255, 275, 284
  Widget::Impl 146, 149
  Widget::processPointer 77
  Wine 64
exemples de fonctions/templates *Voir aussi* std::, 214
  addDivisorFilter 213, 218
  arraySize 16
  authAndAccess 24, 26, 27
  Base::Base 110
  Base::~Base 110
  Base::doWork 78
  Base::mf1 80
  Base::mf2 80
  Base::mf3 80
  Base::mf4 80

© Dunod – Toute reproduction non autorisée est un délit.
Copyright © 2016 Dunod.

Base::operator= 110
calcEpsilon 47
calcValue 256
cbegin 87
cleanup 94
compress 233
computerPriority 137
continueProcessing 70
createInitList 23
createVec 32, 34
cusDel 143
delInvmt2 121
Derived::mf1 80
Derived::mf2 80
Derived::mf3 80
Derived::mf4 80
detect 263, 264
doAsyncWork 237
doSomething 82
doSomeWork 57, 217
doWork 94, 247, 251
dwim 37
f 10, 16, 18, 22, 32, 34, 59, 88, 93, 160, 193, 203, 206, 243
f1 17, 29, 60
f2 17, 29, 60
f3 60
fastLoadWidget 133
features 43
findAndInsert 86
func 5, 39, 192, 195
func_for_cx 19
func_for_rx 19
func_for_x 19
fwd 202
Investment::~Investment 120
isLucky 75
IsValAndArch::IsValAndArch 221
IsValAndArch::operator() 221
killWidget 290
loadWidget 133
lockAndCall 61
logAndAdd 173, 174, 181, 182
logAndAddImpl 182
logAndProcess 157
makeInvestment 117, 120
makeStringDeque 27
makeWidget 79, 83, 169–172
midpoint 99
myFunc 16
nameFromIdx 174
operator+ 3, 168
Password::changeTo 282, 283
Password::Password 282
Person::Person 175, 178, 180, 184, 186, 187, 190
Point::distanceFromOrigin 103
Point::Point 98
Point::setX 98
Point::setY 98
Point::xValue 98
Point::yValue 98
Polynomial::roots 101
PolyWidget::operator() 234
pow 97, 98
primeFactors 67
process 128, 129, 157
processPointer 76, 77
processPointer<char> 76
processPointer<const char> 76
processPointer<const void> 76
processPointer<void> 77
processVal 206
processVals 3
processWidget 143
react 263
reallyAsync 245
reduceAndCopy 169
reflection 100
setAlarm 228, 231
setSignText 168
setup 94

Copyright © 2016 Dunod.

SomeCompilerGeneratedClassName::operator() 225
someFunc 4, 17, 20, 163
SpecialPerson::SpecialPerson 178, 186
SpecialWidget::processWidget 284
std::add_lvalue_reference 65
std::all_of 214
std::basic_ios::basic_ios 74, 156
std::basic_ios::operator= 74, 156
std::forward 194, 195, 226
std::get 252
std::make_shared 136, 167
std::make_unique 136, 167
std::move 154
std::pair::swap 91
std::remove_const 65
std::remove_reference 65
std::swap 91
std::vector<bool>::operator[] 46
std::vector::emplace_back 162
std::vector::operator[] 24
std::vector::push_back 162, 286
StringTable::StringTable 110
StringTable::~StringTable 111
ThreadRAII::get 249, 252
ThreadRAII::ThreadRAII 249
ThreadRAII::~ThreadRAII 249, 252
toUType 72
Warning::override 82
Widget::addFilter 215, 218
Widget::addName 275
Widget::create 129
Widget::data 82
Widget::doWork 79
widgetFactory 195
Widget::isArchived 220
Widget::isProcessed 220
Widget::isValidated 220
Widget::magicValue 104, 105
Widget::operator float 53
Widget::operator= 107, 109, 112, 149, 150
Widget::process 128, 129
Widget::processPointer<char> 76
Widget::processPointer<void> 76
Widget::processWidget 137
Widget::setName 165, 166
Widget::setPtr 280
Widget::Widget 3, 52, 107, 109, 112, 145, 151, 158, 164
Widget::~Widget 109, 145, 148
workOnVal 207
workWithContainer 214
expressions constantes entières, définition 95
expressions lambda
  arguments liés et déliés 233
  capture avec une durée de stockage statique 218
  capture généralisée 219
  capture implicite du pointeur this 216
  capture, modes par défaut 212
  capture par déplacement 234
  capture par référence 213
  capture par valeur, inconvénients 215
  capture par valeur, pointeurs et 215
  code inline et 231
  contre std::bind 228
  définition 5, 211
  fermetures, créer 212
  objets fonctions polymorphiques et 234
  paramètres auto&& et decltype 225
  puissance d'expression 211
  références dans le vide et 213
  surcharge et 230
  variadiques 227
expressions lambda contre std::bind
  arguments déliés, traitement 234
  arguments liés, traitement 233

© Dunod – Toute reproduction non autorisée est un délit.
Copyright © 2016 Dunod.

capture par déplacement 234
code inline et 231
lisibilité 228
objets fonctions polymorphiques et 234
surcharge et 230
expressions lambda génériques
définition 225
`operator()` 225

F

fermetures
classe de fermeture, définition 212
copies 212
définition 5, 212
`final`, mot clé 81
fonctionnalités obsolètes
définition 6
génération automatique d'une opération de copie 109
spécifications d'exceptions en C++98 88
`std::auto_ptr` 116
fonctions
arguments, définition 4
déduction de type de retour 25
dégradation 17
fabriques avec cache 133
gourmandes en C++ 175
`noexcept` conditionnel 91
noms surchargés 206
objets, définition 4
paramètres, définition 4
privées et indéfinies, contre fonctions supprimées 74
références universelles et 175
signature, définition 6
syntaxes des paramètres de type pointeur 206
virtuelles, `override` et 78
fonctions `make`
définition 136
duplication du code et 137
parenthèses contre accolades 140
supprimeurs personnalisés 139
sûreté vis-à-vis des exceptions 137, 291
fonctions membres 86
par défaut 109
qualificatifs de référence et 82
templates 112
fonctions membres spéciales
définition 106
génération implicite 106
template de fonction membre 112
fonctions non membres 87
suppression 75
fonctions supprimées 73
contre fonctions privées et indéfinies 74
définition 74
forwarding, références 160
futurs
comportement indéterminé du destructeur 255
`void` 261

G

garantie forte
définition 4
`noexcept` et 90
opérations de déplacement et 200
garantie minimale, définition 4
gestion de ressources
opérations de copie et destructeur 108
suppression et 124

I

inférence de type *Voir* déduction de type

Copyright © 2016 Dunod.

initialisation
ordre avec des données membres
`std::thread` 250
syntaxes 49
uniforme 50
initialiseurs à accolades 50
`auto` et 21
déduction de type de retour 23
définition 50
`std::initializer_lists` et 53, 54
transmission parfaite et 203, 204
initialiseurs au type explicite 43
insertion
constructeurs `explicit` et 293
contre placement 285
interface, conception
`constexpr` et 100
contrats étendus contre restreints 93
spécifications d'exceptions 88
interrogation, coût/efficacité 260

J

joignabilité, test sur `std::thread` 250

L

lecture-modification-écriture, opérations 266
`std::atomic` et 266
`volatile` et 266
`lhs`, définition 3
lvalues, définition 2

M

mauvaise odeur du code 258
mémoire
entrées-sorties mappées en mémoire 270
locale de thread, définition 243
modèles de cohérence 268
spéciale 269
messages d'erreur, références universelles et 190
mise en exergue 3
modes de capture par défaut 212
most vexing parse, définition 51
mots clés contextuels 81
définition 81

N

named return value optimization (NRVO) 170
neutralité envers les exceptions, définition 92
Newton, lois du mouvement 164
`noexcept` 88
avertissements du compilateur et 94
conditionnel 91
destructeurs et 92
fonctions de désallocation et 92
fonctions `swap` et 90
garantie forte et 90
interface de fonction et 91
opérations de déplacement et 89
`operator delete` et 92
optimisation et 88
NRVO (*named return value optimization*)
Voir optimisation de la valeur de retour nommée
`NULL`
surcharge et 58
templates et 60
`nullptr`
contre `0` et `NULL` 58
surcharge et 59
templates et 60
type 59

O

objets
copie, définition 4
création avec `()` et `{}` 49

© Dunod – Toute reproduction non autorisée est un délit.
Copyright © 2016 Dunod.

fonctions polymorphiques 234
invocables, définition 4
opérations de copie
définition 3
génération automatique 109
idiome Pimpl et 150
implicites dans les classes qui déclarent des opérations de déplacement 108
lien avec le destructeur et la gestion de ressources 108
par construction et par affectation, comparaison 281
par défaut 110
pour des classes qui déclarent des opérations de copie ou un destructeur 109
pour `std::atomic` 272
opérations de déplacement
code générique et 201
définition 3
garantie forte et 200
génération implicite 107
idiome Pimpl et 149
par défaut 110
`std::array` et 199
`std::shared_ptr` et 124
`std::string` et 199
templates et 201
types anciens et 198
`operator()`, dans les expressions lambda génériques 225
`operator[]`, type de retour 24, 46
optimisation
de la valeur de retour, définition 170
des petites chaînes (SSO) 199, 283
`override` 78
comme mot clé 81
conditions de redéfinition 78
fonctions virtuelles 78

P

paramètres
de type référence rvalue 2
par valeur, `std::move` 277
transmission, définition 201
passage par valeur 275
efficacité 277
problème de *slicing* 284
Pimpl (idiome) 144
définition 144
opérations de copie et 150
opérations de déplacement et 149
`std::shared_ptr` et 151
`std::unique_ptr` et 146
temps de compilation et 145
placement
constructeurs `explicit` et 292
construction contre affectation 288
contre insertion 285
fonctions 287
heuristique pour l'utilisation 288
sûreté vis-à-vis des exceptions 290
transmission parfaite et 287
pointeurs
dans le vide *Voir* pointeurs pendouillant
de retour 135
pendouillant, définition 131
pointeurs bruts
comme pointeurs de retour 135
définition 6
inconvénients 115
pointeurs intelligents 115
contre pointeurs bruts 115
définition 6, 116
gestion d'une ressource à propriété exclusive 116
pointeurs pendouillant et 131
points de suspension
étroits 3
larges 3

Copyright © 2016 Dunod.

programmation
multitâche, définition 238
multithread, définition 237
propriétaire d'une ressource, définition 115
propriété
exclusive, définition 117
partagée, définition 123

Q

qualificatifs de référence
définition 79
sur des fonctions membres 82

R

RAII
classes, définition 248
objets, définition 248
réduction de référence 191
`auto` et 196
contextes 195
déclarations d'alias et 196
`decltype` et 197
règles 193
`typedef` et 196
références
à des références, illégalité 192
aux tableaux 16
dans le code binaire 205
dans le vide 213
forwarding 160
réduction 191
référence lvalues, définition 2
références rvalue
contre références universelles 160
définition 2
dernière utilisation 167
paramètres 2
passer à `std::forward` 226
références universelles *Voir aussi* transmission parfaite, 160
alternatives à la surcharge sur 179
`auto` et 163
avantages par rapport à la surcharge 166
constructeurs et 175, 184
contre références rvalue 160
déduction de type et 161
dernière utilisation 167
efficacité 174
encodage de lvalue/rvalue 192
fonctions gournandes et 175
forme syntaxique 161
initialiseurs et 161
messages d'erreur et 190
noms 163
signification réelle 196
`std::move` et 165
surcharge et 173
Règle des trois, définition 108
relais *Voir* transmission parfaite
répartition de charge 240
Resource Acquisition is Initialization *Voir* RAII
ressources, gestion *Voir* gestion de ressources
return value optimization (RVO) 170
réveils intempestifs, définition 259
`rhs`, définition 3
RMW (*read-modify-write*), opérations *Voir* lecture-modification-écriture, opérations
`rvalue_cast` 155
rvalues, définition 2
RVO (*return value optimization*) *Voir* optimisation de la valeur de retour

© Dunod – Toute reproduction non autorisée est un délit.
Copyright © 2016 Dunod.

S

sémantique du déplacement, définition 153
séparateur de milliers, apostrophes 247
SFINAE, technologie 185
shared_from_this 129
signe égal (=), affectation contre initialisation 50
slicing, problème 284
SSO (*small string optimization*)
    Voir optimisation des petites chaînes
static_assert 147, 191
std::add_lvalue_reference 65
std::add_lvalue_reference_t 66
std::allocate_shared
    classes avec gestion personnalisée de la mémoire 141
    efficacité 139
std::all_of 214
std::array, opérations de déplacement et 199
std::async 240
    destructeur des futurs créés 254
    std::packaged_task et 256
    stratégie de démarrage 242
    stratégie de démarrage, boucles temporisées 243
    stratégie de démarrage, mémoire locale de thread 243
    stratégie de démarrage par défaut 242
std::atomic 265
    chargements redondants et 271
    contre volatile 265
    opérations de copie et 272
    opérations de lecture-modification-écriture 266
    réorganisation du code et 267
    stockages morts et 271
    utilisation avec volatile 273
    variables multiple et transactions 104
std::auto_ptr 116
std::basic_ios 74
std::basic_ios::basic_ios 74
std::basic_ios::operator= 74
std::bind
    arguments liés et déliés 233
    captures par déplacement 234
    captures par déplacement, émulation 222
    code inline et 231
    contre expressions lambda 228
    lisibilité 228
    objets fonctions polymorphiques et 234
    surcharge et 230
    transmission parfaite et 234
std::cbegin 87
std::cend 87
std::crbegin 87
std::crend 87
std::decay 185
std::enable_if 184
std::enable_shared_from_this 128
std::false_type 182
std::forward 157, 193, 195
    conversion de type et 154
    références rvalue, passer à 226
    références universelles et 164
    remplacer std::move par 158
    retour par valeur et 168
std::function 39
std::future<void> 261
std::initializer_lists, initialiseurs à accolades et 53
std::is_base_of 187
std::is_constructible 190
std::is_nothrow_move_constructible 91
std::is_same 185
std::launch::async 242
    utilisation automatique en stratégie de démarrage 245

Copyright © 2016 Dunod.

std::launch::deferred 242
boucles temporisées et 243
std::literals 229
std::make_shared *Voir aussi* fonctions make, 136, 167
alternatives à 291
classes avec gestion personnalisée de la mémoire 141
efficacité 139
objets volumineux et 141
std::make_unique *Voir aussi* fonctions make, 136, 167
std::move 154
conversion de type et 154
objets const et 155
paramètres par valeur et 277
références rvalue et 164
références universelles et 165
remplacé par std::forward 158
retour par valeur et 168
std::move_if_noexcept 91
std::nullptr_t 59
std::operator 156
std::operator= 74
std::operator[] 24, 46
std::packaged_task 255
std::async et 256
std::pair 91
std::pair::swap 91
std::plus 230
std::promise 253
valeur, fixer 261
std::promise<void> 261
std::rbegin 87
std::ref 233
std::remove_const 65
std::remove_const_t 66
std::remove_reference 65
std::remove_reference_t 66
std::rend 87
std::result_of 245
std::shared_future<void> 261
std::shared_ptr 122
blocs de contrôle multiples et 127
construction à partir de this 127
construction à partir d'un pointeur brut 126
contre std::weak_ptr 132
conversion depuis std::unique_ptr 122
création à partir d'un std::weak_ptr 132
cycles et 134
efficacité 123, 130
opérations de déplacement et 124
supprimeurs 124
supprimeurs std::unique_ptr 152
tableaux et 131
taille 123
std::string, opérations de déplacement et 199
std::swap 91
std::system_error 239
std::thread
classe RAII pour 249, 263
comme données membres, ordre d'initialisation des membres 250
join ou detach implicite 248
non joignable, définition 246
test de joignabilité 250
std::thread joignable
contre non joignable 246
définition 246
destruction 246
test de joignabilité 250
std::true_type 182
std::unique_ptr 116
conversion vers std::shared_ptr 122
efficacité 116
fonctions fabriques et 117
pour les tableaux 121
supprimeurs 118, 124
supprimeurs std::shared_ptr 152

Copyright © 2016 Dunod.
© Dunod – Toute reproduction non autorisée est un délit.

taille de 121
std::vector 24, 162, 286
constructeurs 56
std::vector<bool> 43, 46
std::vector<bool>::operator[] 46
std::vector<bool>::reference 43
std::vector::emplace_back 162
std::vector::push_back 162, 286
std::weak_ptr
construction d'un std::shared_ptr avec 132
contre std::shared_ptr 132
cycles et 134
design pattern Observateur et 134
efficacité 135
interception d'exception et 133
périmé 132
stockages morts, définition 271
stratégie de démarrage par défaut 242
mémoire locale de thread 243
structures, exemples *Voir* exemples de classes/templates
suffixes de temps 229
suppression de fonctions *Voir* fonctions supprimées
supprimeurs
personnalisés 118, 139
std::unique_ptr contre std::shared_ptr 124, 152
surcharge
alternatives à 179
évolutivité 167
expression lambda et 230
références universelles et 166, 173
types pointeur et types entiers 59
sûreté vis-à-vis des exceptions
alternatives à std::make_shared 142, 291
définition 4
fonctions make et 137, 291
placement 290
système, threads 238

T

T&&, signification 160
tableaux
arguments 15
dégradation, définition 15
paramètres 16
référence à 16
taille, déduire 16
tâches
contre threads 237
état de report, demander 244
répartition de charge 240
tag dispatching 181
templates
actifs contre inactifs 184
actifs, définition 184
alias 63
classes, définition 5
déduction de type 10
déduction de type auto 18
déduction de type, fonctions en arguments 17
déduction de type pour le passage par valeur 14
déduction de type pour les références universelles 13
déduction de type pour les types pointeur et référence 11
déduction de type, tableaux en arguments 15
fonctions, définition 5
inactifs, définition 184
indéfinis pour imposer les messages d'erreur du compilateur 30
instanciations, suppression 76
noms, transmission parfaite et 206
opérateurs standard et arguments de type 230
opérations de déplacement et 201
parenthèses contre accolades 57
The View from Aristeia (blog) IX, 264
thread_local, variables 243

Copyright © 2016 Dunod.

threads
contre tâches 237
descripteur, comportement du destructeur 253
destruction 248
épuisement 239
interruptibles, définition 251
`join` ou `detach` implicite 248
logiciels, définition 238
matériels, définition 238
priorité/affinité, fixer 241, 247, 262
suspension 262
système, définition 238
test de joignabilité 250
valeur de retour d'une fonction et 238
TLS (*thread local storage*) *Voir* mémoire locale de thread
traduction, définition 95
traits de type 65
transformations de type 65
transmission parfaite *Voir aussi* références universelles, 234
cas d'échec 201
constructeurs 175, 184
constructeurs et copie d'objets 176
constructeurs et héritage 178, 186
définition 4, 153, 202
placement et 287
`std::bind` et 234
transmission parfaite (cas d'échec)
champs de bits 208
données membres `static const` intégrales uniquement déclarées 204
initialiseurs à accolades 203
noms de fonctions surchargées et noms de templates 206
`typedef`, réduction de référence et 196
`typeid` et affichage des types déduits 31
`typename`
`class` pour les paramètres de template 3
type dépendant et 64
type non dépendant et 64
types
anciens et opérations de déplacement 198
compatibles avec le déplacement 107
conversion *Voir* conversion de type
déduction *Voir* déduction de type
déduits, affichage 30
dépendants, définition 64
incomplets, définition 145
inférence *Voir* déduction de type
littéraux, définition 98
non dépendants, définition 64
réservés au déplacement, définition 103, 117
test d'égalité 185
traits 65
transformations 65

V

variables de condition
communication d'un événement 257
dépendances temporelles 259
réveils intempestifs et 259
variables locales, retour par valeur et 169
View from Aristeia, The (blog) IX
`volatile` 265
chargements redondants et 271
contre `std::atomic` 265
mémoire spéciale et 269
opérations de lecture-modification-écriture 266
réorganisation du code et 269
stockages morts et 271
utilisation avec `std::atomic` 273

W

`Widget`, définition 3

© Dunod – Toute reproduction non autorisée est un délit.