PROGRAMMER EFFICACEMENT EN C++

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

Scott Meyers

DUNOD

Authorized French translation of *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

© Dunod, 2016
5, rue Laromiguière, 75005 Paris
www.dunod.com
ISBN 978-2-10-074391-9

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

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

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

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

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

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.

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.

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

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 :

```
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

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.

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.
```

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.

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*).

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.

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&`.

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&.
```

```
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.
```

```
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

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` :

```
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 ?
```