Elogios para o C++ Moderno e Eficaz

Então, ainda interessado em C++? Deveria! O C++ Moderno (C++11/C++14) é muito mais do que uma cara nova. A respeito dos recursos novos, parece mais com uma reinvenção. Procurando por um guia e ajuda? Então este livro é certamento o que você está procurando. Quando se fala de C++, Scott Meyers foi e ainda é sinônimo de precisão, qualidade e satisfação.

— Gerhard Kreuzer
Pesquisador e engenheiro de desenvolvimento da Siemens AG

Encontrar o maior especialista é muito difícil. Encontrar um perfeccionismo com o ensino — a obsessão de um autor com a criação de estratégias e a otimização de explicações — também é difícil. E você sabe que vem coisa boa quando encontra ambos personificados em uma só pessoa. *C++ Moderno e Eficaz* é o imponente resultado de um escritor técnico excepcional. Ele acumula esclarecimentos lúcidos, significativos e bem sequenciados sobre tópicos complexos e interconectados, tudo em um estilo literário bem claro. É muito improvável que você encontre um erro técnico uma passagem sem sentido ou uma frase preguiçosa em *C++ Moderno e Eficaz*.

— Andrei Alexandrescu
Pesquisador científico no Facebook e autor do *Modern C++ Design*

Como alguém com mais de duas décadas de experiência em C++, para conseguir o máximo do C++ moderno (tanto a melhor prática quanto os perigos a se evitar) é altamente recomendável comprar este livro, lê-lo com atenção e voltar a ele com frequência! Certamente aprendi coisas novas!

— Nevim Liber
Engenheiro de software sênior no DRW Trading Group

Bjarne Stroustrup — o criador do C++ — disse: "O C++11 parece uma nova linguagem". *C++ Moderno e Eficaz* nos faz compartilhar do mesmo sentimento ao explicar de forma clara como os programadores do dia a dia podem se beneficiar com as novas funções e termos do C++11 e C++14. Outro ótimo livro de Scott Meyers.

— Cassio Neri
Analista quantitativo FX no Lloyds Banking Group

Scott tem um dom de cozinhar a complexidade técnica e servir um kernel compreensível. Seus livros *Effective C++* ajudaram a elevar o estilo de codificação de uma geração passada de programadores C++; o livro mais recente parece fazer o mesmo para os que usam o C++ moderno.

— Roger Orr
Membro do comitê de padronização ISO C++

C++ Moderno e Eficaz é uma grande ferramenta para aprimorar suas habilidades em C++ moderno. Ele não apenas ensina como, quando e onde usar o C++ moderno e fazê-lo de forma eficaz, mas também explica o *porquê*. Sem dúvidas, a escrita clara e perspicaz do Scott, distribuída pelos 42 itens bem escolhidos, permite aos programadores uma compreensão bem melhor da linguagem.

— Bart Vandewoestyne
Engenheiro de pesquisa e desenvolvimento e entusiasta do C++

Amo o C++, ele já foi minha ferramenta de trabalho por muitas décadas. E com a nova leva de funções, ele está agora mais poderoso e expressivo do que sequer imaginei. Mas com todas essas escolhas vem uma questão: "Quando e como usar esses recursos?", Como sempre foi o caso, os livros *Effective C++* de Scott são a resposta definitiva para essa questão.

— Damien Watkins
Líder da equipe de engenharia de software e computação no CSIRO

Uma boa leitura para quem faz a transição para o C++ moderno — as novas funções da linguagem C++11/14 são descritas junto da C++98, e os itens principais são fáceis de reconhecer. Há sempre um conselho ao final de cada seção. Útil e divertido tanto para desenvolvedores profissionais quanto casuais de C++.

— Rachel Cheng
Engenheira de software na F5 Networks

Se você está migrando do C++88/03 para o C++11/14, você precisa das informações práticas e claras que Scott oferece no *C++ Moderno e Eficaz*. Se você já está programando em C++, provavelmente descobrirá os usos dos novos recursos por meio da discussão aprofundada dos novos recursos da linguagem. De qualquer forma, este livro vale cada minuto gasto.

— Rob Stewart
Membro do Boost Steering CommitTee (*boost.org*)

C++ Moderno e Eficaz

Scott Meyers

ALTA BOOKS
EDITORA
Rio de Janeiro, 2016

Para Darla,

uma extraordinária labradora retriever preta

Sumário

Avisos

Comecei a investigar o que agora se conhece como C++0x (a nascente do C++11) em 2009. Postei várias perguntas no grupo de notícias Usenet `comp.std.c++`, e agradeço aos membros da comunidade (especialmente ao Daniel Krügler) por suas postagens, que me foram muito úteis. Nos anos mais recentes eu ia para o Stack Overflow quando tinha perguntas sobre o C++11 e C++14, e estou igualmente em débito com essa comunidade por sua ajuda para compreender os pontos mais específicos do C++.

Em 2010, preparei materiais para um curso de treinamento em C++0x (que acabou sendo publicado como *Overview of the New C++*, pela Artima Publishing, 2010). Ambos os materiais e meu conhecimento foram amplamente beneficiados pela verificação técnica feita por Stephan T. Lavavej, Bernhard Merkle, Stanley Friesen, Leor Zolman, Hendrik Schober e Anthony Williams. Sem a ajuda deles, provavelmente eu não estaria à altura de encarar o *C++ Moderno e Eficaz*. Esse título, a propósito, foi sugerido ou pedido por diversos leitores que responderam à minha postagem no blog em 18 de fevereiro de 2014, "Me ajudem a dar um nome ao livro", e Andrei Alexandrescu (autor do *Modern C++ Design*, Addison-Wesley, 2001) foi gentil o suficiente para abençoar o título ao entender que não era uma invasão ao seu território quanto à terminologia.

Não sou capaz de identificar as origens de todas as informações deste livro, mas algumas fontes tiveram um impacto direto nele. O uso de um *template* indefinido no **Item 4** para convencer a informação de tipo de compiladores foi sugerida por Stephan T. Lavavej, e Matt P. Dziubinski me chamou a atenção para o Boost.TypeIndex. No **Item 5**, o exemplo `unsigned-std::vector<int>::size_type` é do artigo do Andrey Karpov de 28 de fevereiro de 2010, "In what way can C++0x standard help you eliminate 64-bit errors". O exemplo `std::pair<std::string, int>/std::pair<const std::string, int>,` no mesmo item, é da fala de Stephan T. Lavavej, do *Going Native 2012*, "STL11: Magic&& Secrets". O **Item 6** foi inspirado pelo artigo de Herb Sutter em 12 de agosto de 2013, "GotW #94

Solution: AAA Style (Almost Always Auto)". O **Item 9** foi motivado pela postagem no blog do Martinho Fernandes em 27 de maio de 2012, "Handling dependent names". O exemplo do **Item 12** demonstrando a sobrecarga nos qualificadores de referência foi baseado na resposta de Casey para a pergunta "What's a use case for overloading member functions on reference qualifiers?", postada no Stack Overflow em 14 de janeiro de 2014. Meu tratamento do **Item 15** do C++14 na expansão do suporte para as funções `constexpr` incorpora informações que recebi de Rein Halbersma. O **Item 16** é baseado na apresentação de 2102 no C++ *and Beyond*, de Herb Suter, "You don't know `const` and `mutable`". O conselho no **Item 18** de funções factory voltarem a `std::unique_ptrs` é baseado no artigo de 30 de maio de 2013 de Herb Sutter, "*GotW# 90 Solution: Factories*". No **Item 19**, o `fastLoadWidget` vem da apresentação do Herb Sutter de 2013 no *Going Native*, "*My Favorite C++ 10-Liner*". Meu tratamento do `std::unique_ptr` e de tipos incompletos no **Item 22** vem do artigo de Herb Sutter de 27 de novembro de 2011, "GotW #100: Compilation Firewalls", e também da resposta de Howard Hinnant dada em 22 de maio de 2011 a uma pergunta no Stack Overflow, "Is std::unique_ptr<T> required to know the full definition of T?". A adição do `Matrix` no exemplo do **Item 25** é baseada em escritos de David Abrahams. O comentário de JoeArgonne em 8 de dezembro de 2012 à postagem no blog de 30 de novembro de 2012, "Another alternative to lambda move capture", foi a fonte da abordagem no **Item 32** usando o `std::bind` para emular a captura inicial em C++11. A explicação do **Item 37** do problema com um destaque implícito no destrutor do `std::thread` foi tirada do artigo de dezembro de 2008 de Hans-J. Boehm "N2802: A plea to reconsider detach-on-destruction for thread objects". O **Item 41** foi originalmente motivado por discussões da postagem no blog de David Abrahams, "Want speed? Pass by value", em 15 de agosto de 2009. A ideia de que tipos *move-only* merecem tratamento especial veio graças a Matthew Fioravante, enquanto a análise das cópias com base em atribuição se origina de comentários de Howard Hinnant. No **Item 42**, Stephan T. Lavavej e Howard Hinnant me ajudaram a compreender os perfis de performance relativa de funções de colocação e inserção, e Michael Winterberg chamou minha atenção para como a colocação pode resultar em vazamento de recursos (e Michael credita a apresentação "C++ Seasoning" no *Going Native 2013* como sua fonte). Michael também apontou como as funções de colocação usam inicialização direta, enquanto as funções de inserção usam a inicialização por cópia.

Revisar os rascunhos de um livro técnico é uma tarefa exigente que consome tempo e é extremamente importante. Tenho sorte de ter tanta gente disposta a fazer isso por mim.

Rascunhos totais ou parciais do *C++ Moderno e Eficaz* foram revisados oficialmente por 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, e por Tomasz Kamiński. Tive também o feedback de diversos leitores através do e-book em pré-lançamento da O'Reilly e dos livros no prelo da Safari Books Online, dos comentários no meu blog (*The View from Aristeia*) e por e-mail. Sou grato a cada uma das pessoas envolvidas. O livro está *bem* melhor do que seria sem a ajuda dessa gente. Estou em débito especialmente com Stephan T. Lavavej e Rob Stewart, cujas observações extremamente detalhadas e compreensivas me levaram a crer que eles passaram quase tanto tempo quanto eu neste livro. Um agradecimento especial vai para Leor Zolman, que, além de revisar o manuscrito, revisou os exemplos de códigos.

Revisões dedicadas das versões digitais deste livro foram feitas por Gerhard Kreuzer, Emyr Williams e por Bradley E. Needham.

Minha decisão de limitar a extensão do código de linha a 64 caracteres (o máximo possível de ser exibido adequadamente em forma impressa e em diversos dispositivos digitais, orientações de dispositivos e com diversas configurações de fonte) foi baseado nos dados fornecidos por Michael Maher.

Desde a publicação inicial, incorporei os ajustes de bugs e outras melhorias sugeridas por Kostas Vlahavas, Daniel Alonso Alemany, Takatoshi Kondo, Bartek Szurgot, Tyler Brock, Jay Zipnick, Barry Revzin, Robert McCabe, Oliver Bruns, Fabrice

Ferino, Dainis Jonitis, Petr Valasek e Bart Vandewoestyne. Muito obrigado a todos por me ajudarem a melhorar a precisão e clareza do *C++ Moderno e Eficaz.*

Ashley Morgan Williams tornou um jantar no lago Oswego Pizzicato em uma noite especialmente divertida. Quando se trata de uma Caesars extragrande, ela sempre está pronta.

Mais de 20 anos após a primeira experiência de conviver com um autor em produção, minha esposa, Nancy L. Urbano, mais uma vez tolerou vários meses de conversas distraídas com uma mistura de resignação, exasperação e gotas pontuais de compreensão e

ajuda. Durante o mesmo período, nossa cadela, Darla, ficava bem contente em cochilar durante as horas que gastei na frente do computador, mas ela nunca me deixou esquecer de que existe uma vida além do teclado.

Introdução

Se você é um programador C++ e se parece de alguma forma comigo, certamente já pensou em algo do tipo quando o assunto é C++11: "Sim, entendi. É o C++, só que com coisas a mais". Mas ao aprender mais sobre ele, você ficou surpreso com a amplitude das mudanças. Declarações `auto`, loops com base no escopo (*range-based*), expressões lambda e referências rvalue mudam a cara do C++, isso sem mencionar os novos recursos de simultaneidade (*concurrency*). E existem ainda as mudanças de expressões. `0` e `typedef`s saíram, e `nullptr` e declarações do tipo alias entraram. As Enums devem agora ter um escopo. E os ponteiros inteligentes (*smart pointers*) são preferíveis, em vez dos ponteiros embutidos (*built-in*). Mover objetos é, geralmente, melhor do que copiá-los.

Há muito a se aprender sobre o C++11, isso sem falar do C++14.

Ainda mais importante, há muito a se aprender sobre fazer um uso *eficaz* das novas possibilidades. Se você precisa das informações básicas sobre os recursos do C++ "moderno", incluindo as funções, mas está procurando por ajuda para saber como usar os recursos para criar softwares que sejam corretos, eficientes, sustentáveis e portáteis, a busca é mais desafiadora. E é aí que este livro entra. Ele é dedicado não apenas a descrever os recursos do C++11 e C++14, mas a descrever seu uso eficaz.

As informações aqui contidas estão divididas em guias chamados de *Itens*. Quer entender as várias formas de dedução de tipo? Ou saber quando (e quando não) usar as declarações `auto`? Está interessado no porquê de as funções membro `const` serem *thread safe*, como implementar o Idioma Pimpl usando o `std:unique_ptr`, por que se deve evitar modelos de captura padrão em expressões lambda ou as diferenças entre o `std::atomic` e `volatile`? Todas as respostas estão aqui. Além disso, são respostas que independem de plataforma e que estão de acordo com os padrões. Este livro é sobre C++ *portátil*.

Os Itens neste livro são guias, não regras, pois guias têm exceções. A parte mais importante de cada Item não é o conselho que ele oferece, mas a lógica por trás do conselho. Uma vez tendo lido isso, você estará apto a determinar se as circunstâncias de seu projeto justificam a violação do guia do Item. O objetivo real deste livro não é dizer o que fazer ou o que evitar fazer, mas passar um conhecimento mais aprofundado de como as coisas funcionam em C++11 e C++14.

Terminologia e Convenções

Para certificar-nos de que estamos nos entendemos, é importante concordar em algum tipo de terminologia, começando, ironicamente, com "C++". Existem quatro versões oficiais do C++, cada uma nomeada de acordo com o ano no qual o Padrão ISO correspondente foi adotado: *C++98*, *C++03*, *C++11* e *C++14*. C++98 e C++03 são diferentes apenas em detalhes técnicos, então, neste livro, eu me refiro a ambos como C++98. Quando me refiro a C++11, falo de ambos, C++11 e C++14, devido ao C++14 ser, de fato, um super conjunto do C++11. Quando escrevo C++14, falo especificamente do C++14. E se mencionar simplesmente C++, estou fazendo uma afirmação geral que abarca todas as versões da linguagem.

Termo que Uso	Versão da Linguagem
C++	Todas
C++98	C++98 e C++03
C++11	C++11 e C++14
C++14	C++14

Como resultado, posso dizer que o C++ preza por eficácia (verdade para todas as versões), que o C++98 não tem suporte para simultaneidade (verdadeiro apenas paraC++98 e C++03), que o C++11 tem suporte para as expressões lambda (verdadeiro para C++11 e C++14) e que o C++14 oferece funções generalizadas de dedução do tipo de retorno (verdadeiro apenas para o C++14).

Em C++ 11, a funcionalidade mais difundida é, provavelmente, a semântica do movimento, e a raiz da semântica do movimento está na distinção do que é um *rvalue* e um *lvalue*.

E isso acontece porque os rvalue indicam objetos aptos para operações de movimento, enquanto que lvalue geralmente não fazem isso. Na teoria (apesar de nem sempre na prática), os rvalue correspondem aos objetos temporários que retornam de funções, enquanto que os lvalue correspondem a objetos para os quais você pode se referir, seja pelo nome ou por seguir um ponteiro ou referência lvalue.

Um método heurístico para determinar se uma expressão é um lvalue é perguntar se você pode anotar seu endereço. Se puder, então geralmente é. Caso não consiga, é geralmente um rvalue. Um bom recurso desse heurístico é que ele o ajuda a se lembrar de que o tipo de uma expressão é independente da expressão ser um lvalue ou rvalue. Ou seja, dado um tipo `T`, você pode ter lvalue do tipo `T`, bem como rvalues do tipo `T`. Isso é especialmente importante de se lembrar ao lidar com um parâmetro de tipo de referência a rvalue, pois o parâmetro por si só é um lvalue:

```
class Widget {
public:
  Widget(Widget&& rhs);    // rhs é um lvalue, apesar de ter
  …                        // um tipo de referência rvalue
};
```

Aqui seria perfeitamente válido tirar o endereço do `rhs` de dentro do construtor de movimento do `Widget`, então o `rhs` é um lvalue, mesmo que seu tipo seja uma referência rvalue (por lógica similar, todos os parâmetros são lvalue).

Esse pedaço de código demonstra várias convenções que eu costumo seguir:

- O nome da classe é `Widget`. Eu uso `Widget` sempre que quero falar de um tipo arbitrário definido pelo usuário. A menos que precise mostrar dados específicos da classe, eu uso `Widget` sem declará-lo.
- Eu uso o nome de parâmetro `rhs` (*right-hand side*, ou lado da mão direita). É meu nome de parâmetro preferido para as *operações de movimento* (ou seja, construtor de movimento e operador de atribuição de movimento) e para as *operações de cópia* (ou seja, o construtor de cópia e o operador de atribuição de cópia). Eu também o uso para o parâmetro do lado direito de operadores binários:

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

  Espero que não seja surpresa que *`lhs`* signifique *left-hand side*, ou lado da mão esquerda.

- Eu aplico formatação especial a partes do código ou partes de comentários para chamar sua atenção a eles. No construtor de movimento `Widget` anterior, eu destaquei a declaração do `rhs` e a parte do comentário notando que o `rhs` seja um lvalue. O código destacado não é inerentemente bom ou mau. Ele é simplesmente parte do código à qual você deve prestar atenção.
- Eu uso "..." para indicar "outros códigos caem aqui". Essa pequena omissão é diferente da elipse ("..."), que é usada no código-fonte para os templates "variádicos" do C++11. O termo variádico se refere a um template ou função que pode receber um número variável de argumentos. Isso parece ser confuso, mas não é. Por exemplo:

```
template<typename... Ts>               // essas são
void processVals(const Ts&... params)  // elipses de
{                                      // código-fonte C++

  …                                    // isso que dizer que
                                       // "alguns códigos
}                                      // cabem aqui"
```

  A declaração dos `processVals` mostra que uso o `typename` ao declarar os parâmetros de tipos em templates, mas isso é meramente uma preferência pessoal. A palavra-chave `class` funcionaria tão bem quanto. Nessas ocasiões em que mostro excertos de código de um Padrão C++, declaro os parâmetros de tipo usando `class`, pois é isso que os Padrões fazem.

Quando um objeto é inicializado com outro objeto do mesmo tipo, o novo objeto é visto como uma *cópia* do objeto inicial, mesmo que a cópia tenha sido criada via construtor de movimento. Infelizmente, não há uma terminologia em C++ que distingue entre um objeto que é cópia construída por *copy* e uma cópia que tenha sido construída por *move*:

```
void someFunc(Widget w);        // o parâmetro w do
                                // é passado pelo valor

Widget wid;                     // wid é algum Widget

someFunc(wid);                  // nessa chamada a someFunc,
                                // w é uma cópia de wid criada
                                // via construção copy
```

```
someFunc(std::move(wid));       // nessa chamada a SomeFunc,
                                // w é uma cópia de wid criada
                                // via construção move
```

Cópias de rvalue são geralmente construídas por movimento (*move*), enquanto cópias de lvalue são geralmente construídas por cópia (*copy*). Uma implicação disso é que, caso você saiba que um objeto é cópia de outro objeto, não é possível dizer quão trabalhoso foi construir a cópia. No código anterior, por exemplo, não há como dizer quão trabalhoso foi criar o parâmetro `w` sem saber se os rvalue ou lvalue são passados para `someFunc`. Você também teria de saber o custo de mover e copiar `Widgets`.

Em uma função de chamada (*call*), as expressões passadas no campo de chamada são os *argumentos* da função. Os argumentos são usados para inicializar os *parâmetros* da função. Na primeira chamada para a `someFunc` anterior, o argumento é `wid`. Na segunda chamada, o argumento é `std::move(wid)`. Em ambas o parâmetro é `w`. A distinção entre parâmetros e argumentos é importante, pois os parâmetros são lvalue, mas os argumentos com os quais eles são inicializados podem ser tanto rvalue quanto lvalue. Isso é especialmente relevante durante o processo de *encaminhamento perfeito*, por onde um argumento passado para uma função passa por uma segunda função para que o estado de rvalue ou lvalue do argumento original seja preservado. (O encaminhamento perfeito é abordado com detalhes no **Item 30**.)

Funções bem planejadas são *à prova de exceções*, e isso quer dizer que elas oferecem, pelo menos, a garantia de segurança à exceção (ou seja, a *garantia básica*). Tais funções asseguram os chamadores (*callers*) mesmo que uma exceção seja lançada, as invariantes do programa ficam intactas (ou seja, nenhuma estrutura de dados se corrompe) e não há vazamento de recursos. As funções que oferecem a garantia forte de segurança à exceção (ou seja, a *garantia forte*) asseguram aos chamadores que se uma exceção surgir, o estado do programa continuará do jeito que era antes da chamada.

Quando me refiro a uma *função-objeto*, geralmente estou falando de um objeto de um tipo que suporta uma função membro `operator()`. Em outras palavras, um objeto que atua como função. Ocasionalmente uso o termo de uma forma um pouco mais geral que significa qualquer coisa que pode ser invocada usando a sintaxe de uma *chamada* de função não membro (ou seja, "*`function Name(arguments)`*"). Essa definição mais ampla abrange não só os objetos com suporte ao `operator()`, mas também as funções e ponteiros de função do tipo C. A definição mais precisa vem do C++98, a mais ampla

vem do C++11. Generalizando mais ainda, adicionar os ponteiros de função membro nos leva ao que são chamados de *objetos chamáveis* (*callable objects*). Geralmente você pode ignorar as distinções mais específicas e simplesmente pensar em funções-objeto e objetos chamáveis como itens no C++ que podem ser invocados usando algum tipo de sintaxe de chamada de função.

Funções-objeto criados por meio de expressões lambda são conhecidos como *fechamentos (closure)*. Quase nunca é necessário distinguir entre as expressões lambda e os fechamentos que elas criam, então geralmente me refiro a ambas como *lambdas*. Da mesma forma, raramente faço a distinção entre *templates de função* (os templates que geram função) e *funções do template* (as funções geradas pelos templates de função). O mesmo vale para *classes de template* e *template de classe*.

Muitas coisas em C++ podem ser tanto declaradas quanto definidas. As *declarações* apresentam nomes e tipos sem dar detalhes, como onde o armazenamento está localizado ou como as coisas são implementadas:

```
extern int x;                          // declaração de objeto

class Widget;                          // declaração de classe

bool func(const Widget& w);            // declaração de função

enum class Color;                      // declaração de enum com escopo
                                       // (veja o Item 10)
```

As *definições* fornecem os locais de armazenamento ou detalhes de implementação:

```
int x;                                 // definição de objeto

class Widget {                         // definição de classe
  …
};

bool func(const Widget& w)
{ return w.size() < 10; }              // definição de função

enum class Color
{ Yellow, Red, Blue };                 // definição de enum com escopo
```

Uma definição também se qualifica como uma declaração, então, a menos que seja realmente importante que algo seja uma definição, prefiro falar de declarações.

Defino uma *assinatura* de função como a parte de sua declaração que especifica os tipos de parâmetro e de retorno. Nomes de funções e parâmetros não são parte da assinatura. No exemplo anterior, a assinatura do `func` é `bool` (`const Widget&`). Os outros elementos de uma declaração de função, além de seus tipos de parâmetro e retorno (por exemplo: `noexcept` ou `constexpr`, se presentes), são excluídos. (`noexcept` e `constexpr` são descritos nos itens **14** e **15**.) A definição oficial de "assinatura" é ligeiramente diferente da minha, mas para este livro a minha definição é mais útil. A definição oficial, por vezes, omite os tipos de retorno.

Os novos Padrões C++ geralmente preservam a validade do código escrito abaixo dos antigos, mas, por vezes, o Comitê de Padronização *desaprova* funções. Tais funções estão no corredor da morte da padronização e podem ser removidas dos padrões futuros. Os compiladores podem ou não informar sobre o uso de funções reprovadas, mas você deve fazer o seu melhor para evitá-las. Elas não só podem levar a dores de cabeça na hora de fazer o "port" no futuro, mas também são funções geralmente inferiores às que as substituem. Por exemplo, a `std::auto_ptr` foi reprovada no C++11 porque a `std::unique_ptr` faz o mesmo trabalho melhor.

De vez em quando um Padrão diz que o resultado de uma operação é um *comportamento indefinido*. Isso quer dizer que um comportamento do tempo de execução não é previsível, e é desnecessário dizer que você quer se livrar dessa incerteza. Exemplos de ações com comportamento indefinido incluem o uso de colchetes ("`[]`") para indexar além dos limites de um `std::vector`, derreferenciar um iterador não inicializado ou começar uma corrida de dados (por exemplo: ter duas ou mais threads, com, pelo menos, uma funcionando como escritor, acessando o mesmo local de memória simultaneamente).

Eu chamo de *ponteiros embutidos* (*built-in*), como os que retornaram do `new`, *ponteiros crus (raw)*. O oposto de um ponteiro cru é um *ponteiro inteligente (smart)*. Os ponteiros inteligentes geralmente sobrecarregam os operadores de derreferenciamento de ponteiros (`operator->` e `operator*`), apesar de o **Item 20** explicar que o `std::weak_ptr` é uma exceção.

Informando sobre Bugs e Sugerindo Melhorias

Fiz o meu melhor para preencher este livro com informações claras, precisas e úteis, mas certamente existe espaço para melhorias. Se você encontrar erros de qualquer tipo (técnicos, de exposição, gramaticais, tipográficos etc.), ou se tiver sugestões sobre como este livro poderia ser melhorado, visite a página da Alta Books e procure pelo título do livro. Você pode enviar sua opinião ou procurar por possíveis erratas. Novas impressões me dão a oportunidade de revisar o *C++ Moderno e Eficaz*, e eu não posso falar de assuntos dos quais desconheço.

CAPÍTULO 1

Deduzindo Tipos

O C++98 tinha um único conjunto de regras quanto a dedução de tipos: aquele para os templates de função. O C++ 11 modifica um pouco esse esquema de regras e adiciona mais dois, um para `auto` e outro para o `decltype`. O C++ 14, por sua vez, amplia os contextos de uso nos quais o `auto` e o `decltype` podem ser empregados. A aplicação cada vez mais ampla das deduções de tipo nos deixa livres da tirania de soletrar os tipos que são óbvios ou redundantes. Ele torna o programa C++ mais adaptável, pois mudar um tipo em um ponto no código-fonte se propaga automaticamente pelas deduções de tipo para outros locais. Entretanto, isso pode exigir um código mais complicado de se lidar, uma vez que os tipos deduzidos pelos compiladores podem não ser tão aparentes quanto se gostaria.

Sem um bom conhecimento de como a dedução de tipo opera, a programação efetiva do C++ moderno acaba sendo quase impossível. Existem muitos contextos em que a dedução de tipo acontece: em chamadas para templates de função, na maioria dos casos em que o `auto` aparece, nas expressões `decltype` e, para o C++ 14, onde o construtor misterioso `decltype(auto)` é utilizado.

Este capítulo fornece as informações sobre a dedução de tipo de que todos os desenvolvedores C++ precisam. Ele explica como a dedução de tipo de template funciona, como o `auto` constrói em cima disso e como o `decltype` segue seu curso. Ele explica, ainda, como você pode forçar os compiladores a tornar os resultados de suas deduções de tipo visíveis, permitindo assim que você garanta que os compiladores estejam deduzindo os tipos que você quer que eles deduzam.

Item 1: Entendendo a dedução de tipo de template

Quando os usuários de um sistema não sabem como ele funciona, mas estão contentes com o que ele faz, isso diz muito sobre o design de tal sistema. A partir dessa medida, a dedução por tipo de template em C++ é um sucesso tremendo. Milhões de programadores passaram argumentos para as funções do template com resultados completamente satisfatórios, mesmo que muitos desses programadores tivessem dificuldades em dar uma explicação além da mais vaga das descrições sobre como os tipos usados por tais funções foram deduzidos.

Se esse grupo inclui você, eu tenho boas e más notícias. A boa notícia é que a dedução de tipo por templates é a base de uma das funções mais atrativas do C++ moderno: a `auto`. Se você ficou feliz com a maneira com a qual o C++ 98 deduzia tipos para templates, vai ficar ainda mais feliz com a forma com a qual o C++ 11 deduz tipos para a `auto`. A notícia ruim é que quando as regras de dedução de tipo por template são aplicadas no contexto do `auto`, elas parecem de certa forma menos intuitivas que quando aplicadas a templates. Por isso, é realmente importante compreender os aspectos da dedução por tipo de template na qual a `auto` se baseia. Este Item aborda o que você precisa saber.

Se você está disposto a observar um pouco de pseudocódigo, podemos pensar em um template como algo assim:

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

Uma chamada (*call*) pode se parecer com isto:

```
f(expr);                    // chamada f com alguma expressão
```

Durante a compilação, os compiladores usam o *expr* para deduzir dois tipos: um para o *T* e outro para o *ParamType*. Esses tipos são frequentemente diferentes, pois o `ParamType` geralmente contém adornos, como o `const` ou qualificadores de referência. Por exemplo, se o template estiver declarado como a seguir,

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

e tivermos esta chamada,

```
int x = 0;

f(x);                          // chamada f com um int
```

o `T` é deduzido como `int`, mas o *`ParamType`* é deduzido como `const int&`.

É natural esperar que o tipo deduzido para o `T` seja o mesmo que o tipo de argumento passado para a função, por exemplo, que o `T` seja o tipo de *`expr`*. No exemplo anterior, é esse o caso: o `x` é um `int`, e o `T` é deduzido como `int`. Mas isso nem sempre funciona desse jeito. O tipo deduzido para o `T` depende não apenas do tipo de *`expr`*, mas também da forma do *`ParamType`*. Existem três casos:

- O *`ParamType`* é um ponteiro ou tipo de referência, mas não uma referência universal. (Referências universais são descritas no **Item 24**. Neste ponto, tudo o que você precisa saber é que existem e que não são iguais a referências lvalue ou rvalue.)
- O *`ParamType`* é uma referência universal.
- O *`ParamType`* não é nem um ponteiro e nem uma referência.

Portanto, temos três cenários de tipos de dedução a examinar. Cada um será baseado em nossa forma geral para templates e chamadas para ele:

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

f(expr);                  // deduz o T e o paramType do expr
```

Caso 1: O *`ParamType`* É um Ponteiro ou Referência, mas Não uma Referência Universal

A situação mais simples é quando o *`ParamType`* é um tipo de referência ou tipo de ponteiro, mas não uma referência universal. Nesse caso, as deduções por tipo funcionam assim:

1. Se o tipo do *`expr`* é uma referência, ignore a parte de referência.
2. Então compare os padrões do tipo do *`expr`* com os do *`ParamType`* para determinar o `T`.

Por exemplo, se este é nosso template,

```
template<typename T>
void f(T& param);       // param é uma referência
```

e temos estas declarações variáveis,

```
int x = 27;             // x é um int
const int cx = x;       // cx é um const int
const int& rx = x;      // rx é uma referência a x como um const int
```

os tipos deduzidos para `param` e `T` em várias chamadas são os seguintes:

```
f(x);                   // T é int, o tipo de param é int&

f(cx);                  // T é const int,
                        // o tipo do param é const int&

f(rx);                  // T é const int,
                        // o tipo do param é const int&
```

Nas segunda e terceira chamadas, perceba que devido ao `cx` e ao `rx` designarem valores `const`, o `T` é deduzido como `const int`, produzindo, então, um tipo de parâmetro de `const int&`. E isso é importante para os chamadores (*callers*). Quando passam um objeto `const` para um parâmetro de referência, eles esperam que o objeto continue imutável, por exemplo, que o parâmetro seja uma referência a `const`. É por isso que passar um objeto `const` para um template levando um parâmetro `T&` é seguro: a parte do `const` do objeto se torna parte do tipo deduzido para `T`.

No terceiro exemplo, perceba que mesmo que o tipo de `rx` seja uma referência, o `T` é deduzido como sendo não referência. Isso se deve ao fato de a parte de referência do `rx` ser ignorada durante o tipo de dedução.

Se mudarmos o tipo do parâmetro `f` de `T&` para `const T&`, as coisas mudam um pouco, mas não de forma surpreendente. A parte do `const` de `cx` e `rx` continua a ser respeitada, mas por agora assumirmos que o `param` é uma referência a `const`, não há mais necessidade para `const` ser deduzido como parte de `T`:

```
template<typename T>
void f(const T& param);  // param agora é ref-to-const

int x = 27;              // como antes
const int cx = x;        // como antes
```

```
const int& rx = x;       // como antes

f(x);                    // T é int, o tipo de param é const int&

f(cx);                   // T é int, o tipo de param é const int&

f(rx);                   // T é int, o tipo de param é const int&
```

Como antes, a parte de referência do `rx` é ignorada durante a dedução de tipo.

Se o `param` fosse um ponteiro (ou ponteiro para `const`), em vez de uma referência, as coisas funcionariam essencialmente da mesma maneira:

```
template<typename T>
void f(T* param);        // param agora é um ponteiro

int x = 27;              // como antes
const int *px = &x;      // px é um ptr para x como const int

f(&x);                   // T é int, o tipo de param é int*

f(px);                   // T é const int,
                         // o tipo de param é const int*
```

Por enquanto, você pode até bocejar e sentir sono, pois as regras de dedução de tipo do C++ funcionam de forma tão natural para referências e parâmetros de ponteiros, que vê-las na forma escrita parece ser realmente uma chatice. Tudo está tão óbvio! E é exatamente por isso que você quer um sistema de dedução de tipo.

Caso 2: O *ParamType* É uma Referência Universal

As coisas ficam menos óbvias para os templates usando parâmetros de referência universal. Tais parâmetros são declarados como referências de rvalue (por exemplo, em um template de função usando um parâmetro de tipo `T`, um tipo declarado de referência universal é `T&&`), mas eles se comportam de forma diferente quando os argumentos lvalue são passados para dentro. A história completa é contada no **Item 24**, mas aqui está a versão resumida:

- Se o *`expr`* é um lvalue, tanto o `T` quanto o *`ParamType`* são deduzidos como referências lvalue. Isso é duplamente incomum. Primeiro, é a única situação na dedução de tipo de template em que o `T` é deduzido como uma referência. Segun-

do, apesar de o *ParamType* ser declarado usando a sintaxe para uma referência rvalue, seu tipo deduzido é uma referência lvalue.

- Se o *expr* é um rvalue, as regras "normais" (por exemplo, do Caso 1) se aplicam.

Por exemplo:

```
template<typename T>
void f(T&& param);       // param é agora referência universal

int x = 27;              // como antes
const int cx = x;        // como antes
const int& rx = x;       // como antes

f(x);                    // x é lvalue, então T é int&,
                         // o tipo param é também int&

f(cx);                   // cx é lvalue, então T é const int&,
                         // o tipo param é também const int&

f(rx);                   // rx é lvalue, então T é const int&,
                         // o tipo param é também const int&

f(27);                   // 27 é rvalue, então T é int,
                         // o tipo param é portanto int&&
```

O **Item 24** explica exatamente o motivo de esses exemplos saírem do jeito que estão. O ponto principal aqui é que as regras de dedução de tipo para parâmetros de referência universal são diferentes daquelas para parâmetros que são referências lvalue ou referências rvalue. Em particular, quando referências universais estão em uso, a dedução de tipo distingue entre argumentos lvalue e argumentos rvalue. Isso nunca acontece para referências não universais.

Caso 3: O *ParamType* Não É Nem um Ponteiro e Nem uma Referência.

Quando o *ParamType* não é nem um ponteiro e nem uma referência, estamos lidando com uma passagem por valor:

```
template<typename T>
void f(T param);         // param é agora uma passagem por valor
```

Isso significa que o `param` será uma cópia do que quer que seja passado para dentro — um objeto completamente novo. O fato de tal `param` ser um novo objeto motiva as regras que governam sobre como o `T` é deduzido de *`expr`*:

1. Como antes, se o tipo de *`expr`* for uma referência, ignore a parte de referência.
2. Se depois de ignorar a parte de referência de *`expr`*, o *`expr`* for um `const`, ignore isso também. Se for `volatile`, também ignore. Objetos `volatile` são incomuns. São geralmente usados apenas para a implementação de drivers de dispositivo. Para detalhes, veja o **Item 40**.

Portanto:

```
int x = 27;           // como antes
const int cx = x;     // como antes
const int& rx = x;    // como antes

f(x);                 // tipos T e param são ambos int

f(cx);                // tipos T e param são novamente ambos int

f(rx);                // tipos T e param são ainda ambos int
```

Perceba que mesmo apesar de `cx` e `rx` representarem valores `const`, o `param` não é `const`. Isso faz sentido. O `param` é um objeto que é completamente independente do `cx` e `rx` — uma *cópia* de `cx` ou `rx`. O fato de `cx` e `rx` não poderem ser modificados não diz nada sobre a possibilidade do `param` poder ser. É por isso que a parte `const` do *`expr`* (e a parte `volatile`, se existir) é ignorada ao deduzir o tipo para o `param`: só porque o *`expr`* não pode ser modificado não significa que uma cópia deste não possa ser.

É importante reconhecer que o `const` (e o `volatile`) é ignorado apenas pelos parâmetros por valor. Como vimos, para os parâmetros que são referências para ou ponteiros para o `const`, a parte `const` do *`expr`* é preservada durante a dedução de tipo. Mas considere o caso em que o *`expr`* é um ponteiro para um `const`, e *`expr`* é passado por um `param` por valor:

```
template<typename T>
void f(T param);          // param ainda é passado por valor

const char* const ptr =  // ptr é ponteiro const para objeto const
  "Fun with pointers";
```