C++ Eficaz

terceira edição

55 maneiras de aprimorar seus programas e projetos

Scott Meyers

bookman

M612c Meyers, Scott.
C++ eficaz [recurso eletrônico] : 55 maneiras de aprimorar seus programas e projetos / Scott Meyers ; tradução técnica: Eduardo Kessler Piveta. – 3. ed. – Dados eletrônicos. – Porto Alegre : Bookman, 2011.

Editado também como livro impresso em 2011.
ISBN 978-85-7780-820-5

1. Computação – Linguagem de programação – C++.
I. Título.

CDU 004.438C++

Catalogação na publicação: Ana Paula M. Magnus – CRB 10/2052

Scott Meyers

C++ Eficaz

terceira edição

55 maneiras de aprimorar seus programas e projetos

Tradução técnica:

Eduardo Kessler Piveta
Doutor em Ciência da Computação – UFRGS
Professor Adjunto da Universidade Federal de Santa Maria – UFSM

Versão impressa
desta obra: 2011

bookman

2011

Obra originalmente publicada sob o título
Effective C++: 55 Specific Ways to Improve Your Programs and Designs, 3rd Edition.
ISBN 0321334876 / 978-032-133487-9

Arte da capa de Michio Hoshino, Minden Pictures.
Fotografia do autor de Timothy J. Park.

Capa: *Rogério Grilho*, arte sobre capa original

Preparação de original: *Daniel Grassi*

Leitura final: *Taís Bopp da Silva*

Editora Sênior – Bookman: *Arysinha Jacques Affonso*

Editora responsável por esta obra: *Elisa Etzberger Viali*

Projeto e editoração: *Techbooks*

SÃO PAULO
Av. Embaixador Macedo Soares, 10.735 - Pavilhão 5 - Cond. Espace Center
Vila Anastácio 05095-035 São Paulo SP
Fone (11) 3665-1100 Fax (11) 3667-1333

SAC 0800 703-3444

IMPRESSO NO BRASIL
PRINTED IN BRAZIL

AUTOR

Scott Meyers é um dos mais importantes especialistas em desenvolvimento de software C++ do mundo. Autor de diversos livros sobre o tema, também é consultor e membro de conselho editorial de editoras e revistas e já foi membro de comitês técnicos consultivos de várias empresas iniciantes. Recebeu o título de Ph.D em ciência da computação pela Brown University, Rhode Island, Estados Unidos, em 1993. O endereço de seu site é www.aristeia.com.

Para Nancy:
sem ela, nada
valeria muito a pena ser feito.

E em memória de Persephone,
1995–2004

AGRADECIMENTOS

C++ Eficaz existe há quinze anos*, e eu comecei a aprender C++ cerca de cinco anos antes de escrever este livro. O "projeto *C++ Eficaz*" tem estado em desenvolvimento por mais de duas décadas. Durante esse tempo, eu me beneficiei de ideias, sugestões, correções e, ocasionalmente, de conhecimento de centenas (milhares?) de pessoas. Cada uma delas ajudou a melhorar *C++ Eficaz*. Sou grato a todas elas.

Desisti de tentar acompanhar onde aprendi cada coisa, mas uma fonte geral de informação tem me ajudado desde o início: o grupo de notícias de C++ da Usenet, especialmente `comp.lang.c++.moderated` e `comp.std.c++`. Muitos dos Itens neste livro – talvez a maioria deles – se beneficiaram do fluxo de ideias técnicas nas quais os participantes desse grupo são especialistas.

Em relação ao novo material da terceira edição, Steve Dewhurst trabalhou comigo para chegarmos a um conjunto inicial de Itens candidatos. No Item 11, a ideia de implementar `operator=` por meio da técnica de copiar e trocar veio dos textos de Herb Sutter sobre o tópico, ou seja, do Item 13 de seu *Exceptional C++* (Addison-Wesley, 2000). RAII (veja o Item 13) é de Bjarne Stroustrup em *The C++ Programming Language* (Addison-Wesley, 2000). A ideia por trás do Item 17 veio da seção de "Melhores Práticas" da página Web de `shared_ptr` de Boost, `http://boost.org/libs/smart_ptr/shared_ptr.htm#Best-Practices`, e foi refinada pelo Item 21 do livro *More Exceptional C++, de* Herb Sutter (Addison-Wesley, 2002). O Item 29 foi muito influenciado pelos textos abrangentes de Herb Sutter sobre o tópico, ou seja, os Itens 8-19 de *Exceptional C++*, os Itens 17–23 de *More Exceptional C++* e os Itens 11–13 de *Exceptional C++ Style* (Addison-Wesley, 2005); David Abrahams me ajudou a entender melhor as três garantias de segurança de exceções. O idioma NVI no Item 35 é da coluna, "Virtuality", de Herb Sutter, de setembro de 2001, do *C/C++ Users Journal*. No mesmo item, os padrões de projeto Template, Método e Estratégia são do livro *Padrões de Projeto* (Bookman, 2000) de Erich Gamma, Richard Helm, Ralph Johnson e John Vlissides. A ideia de usar o idioma NVI no Item 37 é de Hendrik Schober. David Smallberg contribuiu com a motivação para escrever uma implementação personalizada de set no Item 38. A observação

*N. de T.: Em 2005, quando o original em inglês foi escrito.

no Item 39, de que o EBO geralmente não está disponível sob herança múltipla, é de David Vandevoorde e Nicolai M. Josuttis no livro *C++ Templates* (Addison-Wesley, 2003). No Item 42, meu entendimento inicial sobre typename veio da FAQ sobre C++ e C de Greg Comeau (http://www.comeaucomputing.com/techtalk/#typename), e Leor Zolman me ajudou a entender que meu entendimento estava incorreto (falha minha, não de Greg). A essência do Item 46 é da apresentação de Dan Saks chamada "Making New Friends". A ideia no final do Item 52, de que, se você declarar uma versão de `operator new`, você deve declarar todas elas, é do Item 22 do livro *Exceptional C++ Style* de Herb Sutter. Meu entendimento do processo de revisão de Boost (resumido no Item 55) foi refinado por David Abrahams.

Tudo acima corresponde a com quem ou onde *eu* aprendi algo, não necessariamente a quem inventou ou onde foi inventado ou publicado primeiro.

Minhas notas me dizem que eu também usei informações de Steve Clamage, Antoine Trux, Timothy Knox e Mike Kaelbling, embora, infelizmente, elas não me digam como ou onde as usei.

Os rascunhos da primeira edição foram revisados por Tom Cargill, Glenn Carroll, Tony Davis, Brian Kernighan, Jak Kirman, Doug Lea, Moises Lejter, Eugene Santos Jr., John Shewchuk, John Stasko, Bjarne Stroustrup, Barbara Tilly e Nancy L. Urbano. Recebi sugestões de melhorias, que consegui incorporar em impressões posteriores, de Nancy L. Urbano, Chris Treichel, David Corbin, Paul Gibson, Steve Vinoski, Tom Cargill, Neil Rhodes, David Bern, Russ Williams, Robert Brazile, Doug Morgan, Uwe Steinmüller, Mark Somer, Doug Moore, David Smallberg, Seth Meltzer, Oleg Shteynbuk, David Papurt, Tony Hansen, Peter McCluskey, Stefan Kuhlins, David Braunegg, Paul Chisholm, Adam Zell, Clovis Tondo, Mike Kaelbling, Natraj Kini, Lars Nyman, Greg Lutz, Tim Johnson, John Lakos, Roger Scott, Scott Frohman, Alan Rooks, Robert Poor, Eric Nagler, Antoine Trux, Cade Roux, Chandrika Gokul, Randy Mangoba e Glenn Teitelbaum.

Os rascunhos da segunda edição foram revisados por Derek Bosch, Tim Johnson, Brian Kernighan, Junichi Kimura, Scott Lewandowski, Laura Michaels, David Smallberg, Clovis Tondo, Chris Van Wyk e Oleg Zabluda. As impressões posteriores aproveitaram os comentários de Daniel Steinberg, Arunprasad Marathe, Doug Stapp, Robert Hall, Cheryl Ferguson, Gary Bartlett, Michael Tamm, Kendall Beaman, Eric Nagler, Max Hailperin, Joe Gottman, Richard Weeks, Valentin Bonnard, Jun He, Tim King, Don Maier, Ted Hill, Mark Harrison, Michael Rubenstein, Mark Rodgers, David Goh, Brenton Cooper, Andy Thomas-Cramer, Antoine Trux, John Wait, Brian Sharon, Liam Fitzpatrick, Bernd Mohr, Gary Yee, John O'Hanley, Brady Patterson, Christopher Peterson, Feliks Kluzniak, Isi Dunietz, Christopher Creutzi, Ian Cooper, Carl Harris, Mark Stickel, Clay Budin, Panayotis Matsinopoulos, David Smallberg, Herb Sutter, Pajo Misljencevic, Giulio Agostini, Fredrik Blomqvist, Jimmy Snyder, Byrial Jensen, Witold Kuzminski, Kazunobu Kuriyama, Michael Christensen, Jorge Yáñez Teruel, Mark Davis, Marty Rabinowitz, Ares Lagae e Alexander Medvedev.

O rascunho inicial e parcial desta edição foi revisado por Brian Kernighan, Angelika Langer, Jesse Laeuchli, Roger E. Pedersen, Chris Van Wyk, Nicholas Stroustrup e Hendrik Schober. Os revisores do manuscrito completo foram Leor Zolman, Mike Tsao, Eric Nagler, Gene Gutnik, David Abrahams, Gerhard Kreuzer, Drosos Kourounis, Brian Kernighan, Andrew Kirmse, Balog Pal, Emily Jagdhar, Eugene Kalenkovich, Mike Roze, Enrico Carrara, Benjamin Berck, Jack Reeves, Steve Schirripa, Martin Fallenstedt, Timothy Knox, Yun Bai, Michael Lanzetta, Philipp Janert, Guido Bartolucci, Michael Topic, Jeff Scherpelz, Chris Nauroth, Nishant Mittal, Jeff Somers, Hal Moroff, Vincent Manis, Brandon Chang, Greg Li, Jim Meehan, Alan Geller, Siddhartha Singh, Sam Lee, Sasan Dashtinezhad, Alex Marin, Steve Cai, Thomas Fruchterman, Cory Hicks, David Smallberg, Gunavardhan Kakulapati, Danny Rabbani, Jake Cohen, Hendrik Schober, Paco Viciana, Glenn Kennedy, Jeffrey D. Oldham, Nicholas Stroustrup, Matthew Wilson, Andrei Alexandrescu, Tim Johnson, Leon Matthews, Peter Dulimov e Kevlin Henney. Os rascunhos de alguns Itens individuais foram revisados por Herb Sutter e Attila F. Feher.

Revisar um manuscrito não lapidado (possivelmente incompleto) é um trabalho que exige bastante, e fazê-lo sob a pressão do tempo apenas dificulta a tarefa. Continuo sendo grato ao fato de que tantas pessoas estavam dispostas a fazê-lo para mim.

Revisar é mais difícil ainda se você não tem experiência com o material que está sendo discutido e quando se espera que você capture *todos* os problemas no manuscrito. Surpreende o fato de que algumas pessoas ainda escolham ser redatores/revisores. Chrysta Meadowbrooke foi a redatora/revisora deste livro, e seu trabalho muito cuidadoso expôs muitos problemas que haviam passado batidos por todo mundo.

Leor Zolman verificou todos os exemplos de código em vários compiladores na preparação para a revisão completa, e então fez isso novamente após eu ter revisado o manuscrito. Se ainda houver erros lá, sou responsável por eles, não Leor.

Karl Wiegers e, especialmente, Tim Johnson ofereceram um feedback rápido e prestativo sobre a contracapa do livro na edição norte-americana.

Desde a publicação da primeira edição, incorporei revisões sugeridas por Jason Ross, Robert Yokota, Bernhard Merkle, Attila Fehér, Gerhard Kreuzer, Marcin Sochacki, J. Daniel Smith, Idan Lupinsky, G. Wade Johnson, Clovis Tondo, Joshua Lehrer, T. David Hudson, Phillip Hellewell, Thomas Schell, Eldar Ronen, Ken Kobayashi, Cameron Mac Minn, John Hershberger, Alex Dumov, Vincent Stojanov, Andrew Henrick, Jiongxiong Chen, Balbir Singh, Fraser Ross, Niels Dekker, Harsh Gaurav Vangani, Vasily Poshehonov, Yukitoshi Fujimura, Alex Howlett, Ed Ji Xihuang. Mike Rizzi, Balog Pal, David Solomon, Tony Oliver, Martin Rottinger, Miaohua, Brian Johnson, Joe Suzow, Effeer Chen, Nate Kohl, Zachary Cohen, Owen Chu e Molly Sharp.

John Wait, meu editor das duas primeiras edições deste livro, tolamente se alistou mais uma vez nesse serviço. Sua assistente, Denise Mickelsen, lidou rápida e eficientemente com minhas irritações, sempre com um sorriso agradável. (Ao menos eu acho que ela estava sorrindo. Na verdade, nunca a vi.) Julie Nahil juntou-se ao grupo e logo se tornou minha gerente de produção. Com uma calma notável, ela lidou com a perda, do dia para a noite, de seis semanas no cronograma de produção. John Fuller (chefe dela) e Marty Rabinowitz (chefe dele) ajudaram com questões de produção também. O trabalho oficial de Vanessa Moore era ajudar nas questões do FrameMaker e na preparação dos arquivos PDF, mas ela também adicionou as entradas do Apêndice B e o formatou para a impressão na contracapa. Solveig Haugland me ajudou com a formatação do índice. Sandra Schroeder e Chuti Prasertsith foram responsáveis pelo design da capa, embora coubesse a Chuti retrabalhar a capa cada vez que eu dizia: "Mas e o que vocês acham *desta* foto com uma faixa *daquela* cor?". Chanda Leary-Coutu foi escalada para o trabalho pesado de marketing.

Durante os meses em que trabalhei no manuscrito, muitas vezes a série de TV *Buffy, a Caça-Vampiros* me ajudou a "desestressar" no fim do dia. Foi com muito esforço que consegui manter as falas de Buffy fora do livro.

Kathy Reed me ensinou a programar em 1971, e sou grato por continuarmos amigos até hoje. Donald French contratou a mim e a Moises Lejter para criar materiais de treinamento para C++ em 1989 (uma atitude que me levou a *realmente* conhecer C++), e em 1991 ele me estimulou a apresentá-los na *Stratus Computer*. Os alunos daquela turma me incentivaram a escrever o que se tornou a primeira edição deste livro. Don também me apresentou a John Wait, que concordou em publicá-lo.

Minha esposa, Nancy L. Urbano, continua me estimulando a escrever, mesmo após sete projetos de livros, uma adaptação em CD e uma dissertação. Ela tem uma paciência inacreditável. Eu não poderia fazer o que faço sem ela.

Do início ao fim, nossa cadela, Persephone, foi uma companhia sem igual. Infelizmente, em boa parte deste projeto, sua companhia tomou a forma de uma urna funerária no escritório. Realmente sentimos a sua falta.

A sabedoria e a beleza formam uma combinação muito rara.

— Petronius Arbiter
Satyricon, XCIV

PREFÁCIO

Escrevi a edição original de *C++ Eficaz* em 1991. Quando chegou a hora de uma segunda edição, em 1997, atualizei o material em questões importantes, mas, como não queria confundir os leitores familiarizados com a primeira edição, tentei ao máximo manter a estrutura existente: 48 dos 50 títulos de itens originais permaneceram essencialmente iguais. Se o livro fosse uma casa, a segunda edição seria o equivalente a trocar o carpete, fazer uma nova pintura e trocar algumas lâmpadas.

Para a terceira edição, derrubei tudo. (Houve momentos em que quis começar da fundação.) O mundo de C++ passou por enormes mudanças desde 1991, e o objetivo deste livro – identificar as recomendações de programação C++ mais importantes em pacotes pequenos e legíveis – não mais era atendido pelos itens que estabeleci cerca de 15 anos antes. Em 1991, era razoável imaginar que os programadores de C++ vinham de uma experiência com C. Hoje, os programadores que estão adotando C++ também vêm de Java ou de C#. Em 1991, a herança e a programação orientada a objetos eram novidade para a maioria dos programadores; hoje, são conceitos bem estabelecidos, e as exceções, os templates e a programação genérica são as áreas nas quais as pessoas precisam de mais ajuda. Em 1991, ninguém antes ouvira falar de padrões de projeto; hoje, é difícil discutir sistemas de software sem se referir a eles. Em 1991, o trabalho para a definição de um padrão para C++ estava recém começando; hoje, esse padrão já tem oito anos, e o trabalho para a próxima versão já começou.

Para lidar com essas mudanças, tentei começar o mais próximo possível do zero e me perguntei: "quais são os conselhos mais importantes para os programadores de C++ em 2005?". O resultado é o conjunto de itens nesta nova edição. O livro tem novos capítulos sobre gerenciamento de recursos e sobre programação com templates. Na verdade, interesses relacionados aos templates estão mesclados ao longo do texto, porque afetam praticamente tudo em C++. O livro também inclui novo material sobre programação na presença de exceções, sobre a aplicação de padrões de projeto e sobre o uso dos novos recursos da biblioteca do TR1 (TR1 é descrito no Item 54). O livro reconhece que as técnicas e abordagens que funcionam bem em sistemas com uma linha de execução apenas podem não ser apropriadas em sistemas com várias linhas de execução. Bem, mais ou menos metade do

material no livro é nova. Entretanto, a maioria das informações fundamentais da segunda edição continua sendo importante, então encontrei uma maneira de mantê-las de um jeito ou de outro.

Trabalhei duro para deixar este livro o melhor possível, mas não tenho ilusões que ele seja perfeito. Se você achar que alguns itens neste livro são inadequados como recomendação geral, que existe uma maneira melhor de realizar uma tarefa examinada no livro, ou que uma ou mais discussões técnicas não estão claras, estão incompletas, ou levam a entendimentos equivocados, por favor, me avise. Se você encontrar um erro de qualquer natureza – técnico, gramatical, tipográfico, *qualquer que seja* – por favor, me avise também. Adicionarei, com prazer, nos agradecimentos em impressões posteriores, o nome da primeira pessoa a chamar minha atenção para os problemas.

Mesmo com o número de itens tendo subido para 55, o conjunto de recomendações neste livro está longe de esgotar o tema. Mas chegar a regras boas – que servem para praticamente todas as aplicações o tempo todo – é mais difícil do que pode parecer. Se você tiver sugestões de novas recomendações, ficarei muito feliz de ouvir sobre elas.

Mantenho uma lista de mudanças deste livro desde sua primeira impressão, incluindo correções de erros, esclarecimentos e atualizações técnicas. A lista está disponível na página *Effective C++ Errata*, em `http://aristeia.com/BookErrata/ec++3e-errata.html`. Se você quiser ser notificado quando eu atualizá-la, cadastre-se em minha lista de e-mails. Eu a uso para fazer anúncios que possam interessar as pessoas que acompanham meu trabalho profissional. Para mais detalhes, consulte `http://aristeia.com/MailingList/`.

Scott Douglas Meyers
`http://aristeia.com/`

Stafford, Oregon
Abril de 2005

SUMÁRIO

INTRODUÇÃO

Aprender os fundamentos de uma linguagem de programação é uma coisa; aprender como projetar e implementar programas *eficientes* nessa linguagem é algo completamente diferente, principalmente em C++, uma linguagem poderosa e expressiva. Se usada de forma correta, C++ pode ser ótima de trabalhar; com ela, uma enorme variedade de projetos pode ser expressa diretamente e implementada de maneira eficiente. Um conjunto de classes, funções e templates, cuidadosamente escolhido e criado, pode deixar a programação de aplicações fácil, intuitiva, eficiente e praticamente livre de erros. Não é tão difícil escrever programas eficazes em C++ se você *sabe* como fazê-lo. Empregada sem disciplina, no entanto, C++ pode gerar códigos incompreensíveis, ineficientes, difíceis de serem mantidos e extendidos e, muitas vezes, errados.

O objetivo deste livro é mostrar como usar C++ de modo *eficaz*. Presumo que você já conheça C++ como *linguagem* e tenha alguma experiência em seu uso. Este é um guia para que a linguagem seja usada de forma que seus aplicativos de software sejam compreensíveis, fáceis de serem mantidos, portáveis, extensíveis, eficientes e tendam a se comportar como você espera.

Os conselhos que dou caem em duas categorias amplas: estratégias gerais de projeto e funcionalidades específicas da linguagem e de seus recursos. As discussões de projeto concentram-se em como escolher entre diferentes abordagens para realizar algo em C++. Como você escolhe entre a herança e os *templates*? Entre a herança pública e a privada? Entre a herança privada e a composição? Entre as funções membro e as não membro? Entre a passagem por valor e a passagem por referência? É importante tomar essas decisões corretamente no início, pois uma má escolha pode não ser aparente até muito tarde no processo de desenvolvimento, em um ponto no qual corrigir essa escolha normalmente é difícil, demanda tempo e é caro.

Mesmo quando você sabe exatamente o que quer fazer, pode ser complicado conseguir as coisas da maneira certa. Qual é o tipo de retorno apropriado para os operadores de atribuição? Quando um destrutor deve ser virtual? Como `operator new` deve se comportar quando não consegue encontrar memória suficiente? É crucial tratar de detalhes como esses, pois não fazer

isso quase sempre leva a um comportamento de programa inesperado, muitas vezes obscuro. Este livro o ajudará a evitar isso.

Esta não é uma referência completa sobre C++, mas uma coleção de 55 sugestões específicas (chamadas de *Itens*) para melhorar seus programas e projetos. Cada item é independente, mas a maioria contém referências a outros Itens. Uma maneira de ler o livro é começar com um item de interesse e seguir suas referências para onde elas o levarem.

Este livro também não é uma introdução a C++. O Capítulo 2, por exemplo, discute tudo sobre as implementações adequadas de construtores, destrutores e operadores de atribuição, mas presumo que você já conheça ou possa consultar outra referência para descobrir o que essas funções fazem e como elas são declaradas. Diversas obras sobre C++ contêm informações como essas.

O propósito *deste* livro é destacar aqueles aspectos da programação em C++ que muitas vezes não são vistos com o cuidado necessário; outros livros descrevem as diferentes partes da linguagem. Este diz como combinar essas partes de forma que você tenha programas eficazes; outros ensinam como fazer seus programas serem compilados. Esta obra discute como evitar problemas que os compiladores não dirão a você.

Ao mesmo tempo, este livro limita-se a C++ *padrão*: apenas recursos no padrão oficial da linguagem foram usados. A portabilidade é uma preocupação muito importante aqui; se você está procurando truques que dependem de plataforma, este não é o lugar para encontrá-los.

Outra coisa que você não encontrará neste livro é a bíblia de C++, o único caminho verdadeiro para o desenvolvimento de software perfeito em C++. Todos os itens deste livro fornecem guias sobre como desenvolver projetos melhores, como evitar problemas comuns, ou como ter maior eficiência, mas nenhum é universalmente aplicável. O projeto e a implementação de software são tarefas complexas, dificultadas ainda mais pelas restrições de hardware, de sistema operacional e de aplicativo, então o melhor que posso fazer é fornecer *recomendações* para criar programas melhores.

Se você seguir todas as recomendações o tempo todo, provavelmente não cairá nas armadilhas mais comuns em C++, mas as recomendações, por sua natureza, têm exceções. É por isso que cada item tem uma explicação. As explicações são a parte mais importante do livro. Só entendendo o raciocínio por trás de um item é que você pode determinar se ele se aplica ao sistema de software que você está desenvolvendo e às restrições que precisa atender.

O melhor uso deste livro é para saber como C++ se comporta, por que se comporta dessa maneira e como usar esse comportamento em seu benefí-

cio. A aplicação cega dos itens deste livro é obviamente inadequada, mas, ao mesmo tempo, você provavelmente não deve violar as recomendações sem uma boa razão.

Terminologia

Existe um pequeno vocabulário C++ que todo programador deve entender. Os termos a seguir são suficientemente importantes para que concordemos quanto ao seu significado.

Uma **declaração** diz aos compiladores o nome e o tipo de algo, mas omite certos detalhes. Estes são exemplos de declarações:

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

std::size_t numDigits(int number);             // declaração de função

class Widget;                                  // declaração de classe

template<typename T>                           // declaração de template
class GraphNode;                               // (veja o Item 42 para mais informações sobre
                                               // o uso de "typename")
```

Observe que me refiro ao inteiro `x` como um "objeto", mesmo que ele seja de um tipo primitivo. Algumas pessoas reservam o nome "objeto" para variáveis de tipos definidos pelo usuário, mas não sou uma delas. Observe também que o tipo de retorno da função `numDigits` é `std::size_t`, ou seja, o tipo `size_t` no espaço de nomes `std`. Esse espaço de nomes é onde se localiza praticamente tudo da biblioteca padrão de C++. Entretanto, como a biblioteca padrão de C (aquela de C89, para ser mais preciso) também pode ser usada em C++, os símbolos herdados de C (como `size_t`) podem existir em escopo global, dentro de `std`, ou em ambos, dependendo de quais cabeçalhos foram incluídos (através de `#include`). Neste livro, considero que os cabeçalhos C++ foram incluídos, e é por isso que me refiro a `std::size_t` em vez de `size_t`. Quando me refiro aos componentes da biblioteca padrão, em geral omito referências a `std`, pois entendo que você reconhece que coisas como `size_t`, `vector` e `cout` estão em `std`. Em código de exemplos, sempre incluo `std`, porque o código real não será compilado sem isso.

A propósito, `size_t` é apenas uma definição de tipo para alguns tipos sem sinal que C++ usa quando está contando coisas (como o número de caracteres em uma cadeia baseada em `char*`, o número de elementos em um contêiner STL, etc). Também é o tipo usado pelas funções `operator[]` em `vector`, `deque` e `string`, uma convenção que seguiremos quando estivermos definindo nossas próprias funções `operator[]` no Item 3.

A declaração de cada função revela sua **assinatura**, ou seja, seus tipos de parâmetros e de retorno. A assinatura de uma função é o mesmo que seu tipo. No caso de `numDigits`, a assinatura é `std::size_t(int)`, ou seja, "uma função que recebe um `int` e retorna um `std::size_t`". A definição oficial C++ de "assinatura" exclui o tipo de retorno da função,

mas, neste livro, é mais útil que o tipo de retorno seja considerado parte da assinatura.

Uma ***definição*** fornece aos compiladores os detalhes que uma declaração omite. Para um objeto, a definição é onde os compiladores reservam memória para o objeto. Para uma função ou para um template de função, a definição fornece o corpo de código. Para uma classe ou para um template de classe, a definição lista os membros da classe ou do template:

```
int x;                                          // definição de objeto

std::size_t numDigits(int number)               // definição de função
{                                               // (Esta função retorna
   std::size_t digitsSoFar = 1;                 // o número de dígitos
                                                // em seu parâmetro)
   while ((number /= 10) != 0) ++digitsSoFar;

   return digitsSoFar;
}

class Widget {                                  // definição de classe
public:
   Widget( );
   ~Widget( );
   ...
};

template<typename T>                            // definição de template
class GraphNode {
public:
   GraphNode( );
   ~GraphNode( );
   ...
};
```

A ***inicialização*** é o processo de dar a um objeto seu primeiro valor. Para objetos de tipos definidos pelo usuário, a inicialização é realizada pelos construtores. Um ***construtor padrão*** é aquele que pode ser chamado sem argumentos. Esse construtor não tem parâmetros ou possui um valor padrão para cada um dos parâmetros:

```
class A {
public:
   A( );                                        // construtor padrão
};

class B {
public:
   explicit B(int x = 0, bool b = true);        // construtor padrão; veja abaixo
};                                              // para mais informações sobre "explicit"

class C {
public:
   explicit C(int x);                           // não é um construtor padrão
};
```

Os construtores para as classes B e C são declarados como explícitos (`explicit`) aqui. Isso impede que sejam usados para realizar conversões

de tipo implícitas, apesar de ainda poderem ser usados para conversões de tipo explícitas:

```
void doSomething(B bObject);          // uma função que recebe um objeto do
                                      // tipo B
B bObj1;                              // um objeto do tipo B

doSomething(bObj1);                   // ok, passa um B para doSomething

B bObj2(28);                          // ok, cria um B a partir do inteiro 28
                                      // (o valor de bool é padronizado como verdadeiro)

doSomething(28);                      // erro! doSomething recebe um B
                                      // não é um int e não existe uma
                                      // conversão implícita de int para B

doSomething(B(28));                   // ok, usa o construtor de B para
                                      // converter explicitamente (cast) o
                                      // int para um B para esta chamada. (Veja
                                      // o Item 27 para obter mais informações
                                      // sobre conversões explícitas)
```

Os construtores declarados como explícitos (`explicit`) normalmente são preferíveis aos não explícitos, porque impedem que os compiladores realizem conversões de tipo inesperadas (frequentemente não desejadas). A menos que tenha uma boa razão para permitir que um construtor seja usado para conversões de tipo implícitas, o declaro como explícito. Incentivo o leitor a seguir a mesma política.

Observe como destaquei a conversão explícita no exemplo acima. Ao longo deste livro, uso esses destaques para chamar a sua atenção para o material que é digno de nota. (Também destaco os números de capítulos, mas apenas porque acho que fica bonito.)

O ***construtor de cópia*** é usado para inicializar um objeto com um objeto diferente do mesmo tipo, e o ***operador de atribuição por cópia*** é usado para copiar um valor de um objeto para outro do mesmo tipo:

```
class Widget {
public:
  Widget( );                                // construtor padrão
  Widget(const Widget& rhs);                // construtor de cópia
  Widget& operator=(const Widget& rhs);     // construtor de atribuição por cópia
  ...
};

Widget w1;                                  // invoca o construtor padrão

Widget w2(w1);                              // invoca o construtor de cópia

w1 = w2;                                    // invoca o construtor de atribuição por cópia
```

Quando enxergar o que aparenta ser uma atribuição, leia com muita atenção, porque a sintaxe "=" pode ser usada para chamar o construtor de cópia:

```
Widget w3 = w2;                             // invoca o construtor de cópia!
```

Felizmente, a construção de cópia é fácil de distinguir da atribuição por cópia. Se um novo objeto está sendo definido (tal como `w3` na sentença acima), é necessário chamar um construtor; ele não pode ser atribuído. Se nenhum objeto novo estiver sendo definido (como a sentença "`w1 = w2`" acima), nenhum construtor pode ser envolvido, então é uma atribuição.

O construtor de cópia é uma função especialmente importante, porque define como um objeto é passado por valor. Por exemplo, considere o seguinte:

```
bool hasAcceptableQuality(Widget w);
...
Widget aWidget;
if (hasAcceptableQuality(aWidget)) ...
```

O parâmetro `w` é passado para `hasAcceptableQuality` por valor, então, na chamada acima, `Widget` é copiada para `w`. A cópia é feita pelo construtor de cópia de `Widget`. A passagem por valor *significa* "chame o construtor de cópia". (Entretanto, passar tipos definidos pelo usuário por valor costuma ser uma má ideia. Passar por referência a `const` geralmente é uma escolha melhor. Para obter mais detalhes, veja o Item 20.)

A ***STL*** é a Biblioteca de Templates Padrão – *Standard Template Library* –, a parte da biblioteca padrão a de C++ dedicada aos contêineres (por exemplo, `vector`, `list`, `set`, `map`, etc), iteradores (por exemplo, `vector<int>::iterator`, `set<string>::iterator`, etc), algoritmos (`for_each`, `find`, `sort`, etc) e funcionalidades relacionadas. Muitas dessas funcionalidades estão ligadas a ***objetos função***: objetos que agem como funções. Esses objetos vêm de classes que sobrecarregam `operator()`, o operador de chamada da função. Se você não está familiarizado com a STL, é interessante ter uma referência decente disponível à medida que lê este livro, pois a STL é útil demais para que eu não tire proveito dela. Depois de usá-la um pouco, você pensará da mesma forma.

Os programadores que vêm de linguagens como Java ou C# podem se surpreender com a noção de ***comportamento indefinido***. Por uma série de razões, o comportamento de algumas construções em C++ é literalmente indefinido: você não pode prever com certeza o que acontecerá em tempo de execução. Veja dois exemplos de código com comportamento indefinido:

```
int *p = 0;                          // p é um ponteiro nulo
std::cout << *p;                     // desreferenciar um ponteiro nulo
                                     // leva a comportamento indefinido
char name[ ] = "Darla";              // nome é um vetor de tamanho 6 (não
                                     // se esqueça do nulo no final!)
char c = name[10];                   // referencia um índice inválido de vetor
                                     // leva a comportamento indefinido
```

Para enfatizar que os resultados de comportamentos indefinidos não são previsíveis e podem ser muito desagradáveis, programadores experientes

frequentemente dizem que os programas com comportamento indefinido podem apagar seu disco rígido. É verdade: um programa com comportamento indefinido *pode* apagar seu disco rígido. Mas isso é pouco provável. É mais provável que o programa se comporte de maneira errática, algumas vezes rodando normalmente, outras travando, e ainda outras produzindo resultados incorretos. Programadores de C++ eficazes fazem o melhor possível para se livrar de comportamentos indefinidos. Neste livro, destaco diversos locais nos quais você precisa tomar cuidado com isso.

Outro termo que pode causar confusão para os programadores que vêm de outra linguagem é o termo **interface**. As linguagens Java e .NET fornecem interfaces como elemento de linguagem, mas isso não existe em C++, embora o Item 31 discuta como abordá-las. Quando uso o termo "interface", geralmente estou falando da assinatura de uma função, sobre os elementos acessíveis de uma classe (por exemplo, a "interface pública", a "interface protegida" ou a "interface privada" de uma classe) ou sobre expressões que devem ser válidas para um parâmetro de tipo de um template (veja o Item 41). Ou seja, estou falando de interfaces como uma ideia bem geral de projeto.

Um **cliente** é alguém ou algo que usa o código (em geral, as interfaces) que você escreve. Os clientes de uma função, por exemplo, são seus usuários: as partes do código que chamam a função (ou que recebem seu endereço), bem como os seres humanos que escrevem e mantêm tal código. Os clientes de uma classe ou de um template são as partes de software que usam a classe ou o template, bem como os programadores que escrevem e mantêm esse código. Quando discuto clientes, normalmente me foco nos programadores, pois eles podem ser confundidos, enganados ou incomodados por interfaces ruins. O código que eles escrevem, não.

Você pode não estar acostumado a pensar nos clientes, mas passarei boa parte do tempo tentando convencê-lo a tornar a vida deles o mais fácil possível. Você mesmo é um cliente dos sistemas de software que outras pessoas desenvolvem – não gostaria que essas pessoas tornassem as coisas fáceis para você? Além disso, em algum momento, você certamente se encontrará na posição de seu próprio cliente (ou seja, usando código que escreveu) e, nesse ponto, ficará feliz por ter se preocupado com os clientes quando estava desenvolvendo suas interfaces.

Neste livro, frequentemente ignoro as distinções entre funções e templates de funções e entre classes e templates de classes. Isso porque o que é verdade para um normalmente é verdade para o outro. Nas situações em que esse não for o caso, faço distinção entre classes, funções e templates que fazem surgir classes e funções.

Ao me referir a construtores e destrutores em comentários de código, às vezes uso as abreviações **ctor** e **dtor**.

Convenções de nomenclatura

Tentei selecionar nomes significativos para objetos, classes, funções, templates, etc., mas o significado por trás de alguns nomes pode não ser imediatamente visível. Dois dos meus nomes de parâmetros favoritos, por exemplo, são `lhs` e `rhs`. Eles significam "lado esquerdo" (*left-hand side*) e "lado direito" (*right-hand side*), respectivamente. É comum eu usá-los como nomes de parâmetros para funções que implementam operadores binários, como `operator==` e `operator*`. Por exemplo, se `a` e `b` são objetos que representam números racionais, e se os objetos racionais (`Rational`) puderem ser multiplicados por uma função não membro `operator*` (como explica o Item 24, esse provavelmente é o caso), a expressão

```
a * b
```

é equivalente à chamada a função

```
operator*(a, b)
```

No Item 24, declaro `operator*` assim:

```
const Rational operator*(const Rational& lhs, const Rational& rhs);
```

Como você pode ver, o operando da esquerda, `a`, é conhecido como `lhs` dentro da função, e o operando da direita, `b`, é conhecido como `rhs`.

Para as funções membro, o argumento da esquerda é representado pelo ponteiro `this`, assim, às vezes uso o nome do parâmetro `rhs` por si mesmo. Talvez você tenha observado isso nas declarações para algumas funções membro de `Widget` na página 5. Frequentemente, uso a classe `Widget` nos exemplos; "`Widget`" não tem significado específico, é apenas um nome que uso quando preciso de um nome de classe de exemplo. Nada tem a ver com os *widgets* dos kits de ferramentas de interface gráfica com o usuário (GUIs).

Também nomeio bastante os ponteiros seguindo a regra em que um ponteiro para um objeto do tipo `T` é chamado de `pt`, "ponteiro para `T`". Veja alguns exemplos:

```
Widget *pw;                                  // pw = ptr para Widget

class Airplane;
Airplane *pa;                                // pa = ptr para Airplane

class GameCharacter;
GameCharacter *pgc;                          // pgc = ponteiro para GameCharacter
```

Uso uma convenção parecida para referências: `rw` pode ser uma referência para um `Widget` e `ra` uma referência para uma aeronave (`Airplane`).

Às vezes, uso o nome `mf` quando estou falando sobre funções membro.

Considerações sobre linhas de execução (*threads*)

Como linguagem, C++ não tem a noção de linhas de execução – nenhuma noção de concorrência de qualquer tipo, na verdade. O mesmo ocorre com

a biblioteca padrão de C++. No que diz respito a C++, não existem programas com linhas de execução múltiplas.

Ainda assim elas existem. Meu foco, neste livro, é C++ padrão, portável, mas não posso ignorar o fato de que a segurança das linhas de execução é uma questão que deve ser tratada por muitos programadores. Minha abordagem para lidar com essa diferença entre o padrão C++ e a realidade é apontar os locais em que as construções de C++ examinadas podem causar problemas em um ambiente com linhas de execução múltiplas. No entanto, este livro não é sobre programação com linhas de execução múltiplas em C++. Longe disso. Esta obra se foca em programação C++ e, embora em sua maior parte se limite a considerações que levam em conta uma linha de execução única, discute a existência de linhas de execução múltiplas. Ela também tenta apontar os locais específicos em que os programadores que precisam estar cientes das linhas de execução de um programa têm que tomar cuidado ao avaliar o conselho que ofereço.

Se você não está familiarizado com linhas de execução múltiplas ou não precisa se preocupar com isso, pode ignorar meus comentários sobre elas. Se estiver programando um aplicativo ou biblioteca com linhas de execução múltiplas, entretanto, lembre que meus comentários são um pouco mais do que um ponto de partida para as questões com as quais precisará lidar quando estiver usando C++.

TR1 e Boost

Você encontrará referências a TR1 e a Boost ao longo do livro. Cada um tem um item que o descreve com algum detalhamento (Item 54 para TR1 e Item 55 para Boost), mas, infelizmente, esses itens estão no final do livro. (Eles estão lá porque funciona melhor assim. Verdade. Tentei colocá-los em vários outros lugares, mas não deu.) Se quiser, você pode ir para esses itens e lê-los agora, mas se preferir começar o livro do início e não do fim, este resumo vai ajudá-lo:

- TR1 (Relatório Técnico 1 – *Technical Report 1*) é uma especificação para novas funcionalidades adicionadas à biblioteca padrão de C++. Essas funcionalidades tomam a forma de classes e templates de funções novas para coisas como tabelas de dispersão, ponteiros espertos de contagem de referência, expressões regulares, e muito mais. Todos os componentes TR1 estão no espaço de nomes `tr1` que está aninhado dentro do espaço de nomes `std`.

- Boost é uma organização e um site (`http://boost.org`) que oferece bibliotecas C++ portáveis, revisadas por pares e de código aberto. A maior parte da funcionalidade de TR1 é baseada no trabalho feito em Boost, e, até que os fornecedores de compiladores incluam TR1 em suas distribuições da biblioteca de C++, o site Boost provavelmente continuará sendo a primeira parada para os desenvolvedores em busca de implementações do TR1. Boost oferece mais do que está disponível em TR1, entretanto; assim, vale a pena conhecê-lo.

CAPÍTULO 1

ACOSTUMANDO-SE COM A LINGUAGEM C++

Seja qual for a sua experiência em programação, é provável que você demore um pouco para se acostumar com a linguagem C++. Trata-se de uma linguagem poderosa com uma enorme variedade de ferramentas, mas, para aproveitar todo esse poder e utilizar seus recursos de modo eficaz, é preciso adaptar-se ao modo C++ de fazer as coisas. Esse é o tema deste livro, e este capítulo aborda os aspectos mais importantes da linguagem.

Item 1: Pense em C++ como um conjunto de linguagens

No início, C++ era apenas C com alguns recursos de orientação a objetos incorporados. O próprio nome original da linguagem, "C com Classes", refletia essa herança.

À medida que C++ amadurecia, a linguagem se tornava mais ousada e aventureira, adotando ideias, recursos e estratégias de programação diferentes das de C com Classes. As exceções exigiram abordagens diferentes para estruturar as funções (veja o Item 29). Os templates fizeram surgir novas maneiras de pensar sobre o projeto (veja o Item 41), e a STL definiu uma visão para a extensibilidade diferente de tudo o que a maioria das pessoas já havia visto.

Hoje, C++ é uma *linguagem de programação de múltiplos paradigmas*, que suporta uma combinação de recursos procedurais orientados a objetos, funcionais, genéricos e de metaprogramação. Esse poder e flexibilidade tornam C++ uma ferramenta ímpar, mas também podem confundir – todas as regras de "uso apropriado" parecem ter exceções. Como vamos entender essa linguagem?

A maneira mais fácil é ver C++ não como uma linguagem, mas como um conjunto de linguagens relacionadas. Dentro de uma sublinguagem específica, as regras tendem a ser simples, diretas e fáceis de lembrar. Quando você se move de uma sublinguagem para outra, entretanto, as regras podem mudar. Para entender C++, você precisa reconhecer suas sublinguagens principais. Felizmente, existem apenas quatro:

- **C.** No fundo, C++ ainda é uma linguagem baseada em C. Os blocos, as sentenças, o pré-processador, os tipos de dados predefinidos, os vetores, os ponteiros, etc., são oriundos de C. Em muitos casos, C++ fornece abordagens para resolver problemas que são superiores às correspondentes em C – por exemplo, veja os Itens 2 (alternativas ao pré-processador) e 13 (usando objetos para gerenciar recursos). No entanto, quando você programa com a parte C de C++, as regras para programação eficaz refletem o escopo mais limitado de C: não há *templates*, não há exceções, não há sobrecarga, etc.

- **C++ Orientada a Objetos.** É a parte que C com Classes representava: classes (incluindo construtores e destrutores), encapsulamento, herança, polimorfismo, funções virtuais (vinculação dinâmica), etc. As regras clássicas de projeto orientado a objetos aplicam-se mais diretamente nessa sublinguagem.

- **C++ com Templates.** É a parte de programação genérica de C++, aquela em que a maioria dos programadores tem menos experiência de uso. As considerações de templates permeiam C++, e é comum que as regras de boa programação incluam cláusulas especiais que se aplicam somente aos templates (por exemplo, veja o Item 46 sobre como facilitar conversões de tipos em chamadas a funções templates). Na verdade, os templates são tão poderosos que deram origem a um paradigma de programação completamente novo, a metaprogramação por templates (TMP – template *metaprogramming*). O Item 48 oferece uma visão geral da TMP, mas, a menos que você seja viciado em templates, não precisa se preocupar com isso. As regras para TMP raramente interagem com a programação mais utilizada em C++.

- **STL.** A STL é uma biblioteca de templates muito especial. Suas convenções sobre contêineres, iteradores, algoritmos e objetos função entrelaçam-se com perfeição, mas os templates e as bibliotecas podem ser criados também em torno de outras ideias. A STL tem uma maneira específica de trabalhar: certifique-se de seguir suas convenções ao usar essa sublinguagem.

Mantenha essas quatro sublinguagens em mente e não se surpreenda ao encontrar situações em que uma programação eficaz exigir uma mudança de estratégia ao trocar de uma sublinguagem para outra. Por exemplo, a passagem por valor, em geral, é mais eficiente do que a passagem por referência para tipos predefinidos (por exemplo, tipos de C), mas, quando você se move da parte C de C++ para C++ Orientada a Objetos, a existência de construtores e destrutores definidos pelos usuários normalmente torna a "passagem por referência para constante" uma opção melhor. Isso ocorre principalmente ao trabalhar com C++ com Templates, porque, nesse caso, você nem mesmo sabe o tipo de objeto de que estará tratando. Ao lidar com a STL, entretanto, você sabe que os iteradores e os objetos função são modelados como ponteiro em C; portanto, a regra antiga de C sobre a passagem por valor se aplica novamente (para ver mais detalhes sobre como escolher dentre as opções de passagem de parâmetros, veja o Item 20).

C++, portanto, não é uma linguagem unificada com um conjunto único de regras: é uma união de quatro sublinguagens, cada uma com suas próprias convenções. Com essas sublinguagens em mente, C++ será muito mais fácil de entender.

Lembrete

» As regras de uma programação eficaz em C++ variam de acordo com a parte da linguagem que você está usando.

Item 2: Prefira constantes, enumerações e internalizações a definições

Este item poderia ser intitulado "prefira o compilador ao pré-processador", pois `#define` pode ser tratado como se não fizesse parte da linguagem propriamente dita. Esse é um de seus problemas. Quando você faz algo como

```
#define ASPECT_RATIO 1.653
```

o nome simbólico `ASPECT_RATIO` (proporção de tela*) talvez nunca seja visto pelos compiladores; ele pode ser removido pelo pré-processador antes que o código-fonte chegue a um compilador. Como resultado, o nome `ASPECT_RATIO` talvez não entre na tabela de símbolos, o que pode ser confuso se você receber um erro durante a compilação envolvendo o uso da constante, pois a mensagem de erro talvez se referencie a `1.653`, mas não a `ASPECT_RATIO`. Se `ASPECT_RATIO` fosse definido em um arquivo de cabeçalho que você não escreveu, você não teria ideia de onde veio esse `1.653`, e perderia tempo tentando rastreá-lo. Esse problema também pode aparecer em um depurador simbólico, porque, mais uma vez, o nome com o qual você está programando pode não estar na tabela de símbolos.

A solução é substituir a macro por uma constante:

```
const double AspectRatio = 1.653;          // nomes em maiúsculo em geral são para
                                           // macros, logo a troca de nomes
```

Como uma constante de linguagem, `AspectRatio` é definitivamente visto pelos compiladores e, certamente, é inserido em suas tabelas de símbolos. Além disso, no caso de uma constante de ponto flutuante (como nesse exemplo), o uso da constante pode levar a um código menor do que com um `#define`. Isso ocorre porque a substituição cega do pré-processador do nome de macro `ASPECT_RATIO` por 1.653 pode resultar em cópias múltiplas de 1.653 em seu código objeto, enquanto que o uso da constante `AspectRatio` nunca deve resultar em mais de uma cópia.

* N. de T.: "*Aspect ratio*" é uma constante numérica muito usada para representar a proporção de tela de uma imagem bidimensional.

Quanto à substituição de `#define`s por constantes, vale a pena mencionar dois casos especiais. O primeiro é a definição de ponteiros constantes. Como as definições constantes em geral são colocadas em arquivos de cabeçalho (em que muitos arquivos-fonte diferentes vão incluí-los), é importante que o *ponteiro* seja declarado como constante (`const`), normalmente em acréscimo àquilo para o qual o ponteiro aponta. Para definir uma cadeia de caracteres (string) constante baseada em `char*` em um arquivo de cabeçalho, por exemplo, você precisa escrever `const` *duas vezes*:

```
const char * const authorName = "Scott Meyers";
```

Para ver uma discussão completa dos significados e usos de `const`, especialmente em conjunto com ponteiros, veja o Item 3. Entretanto, vale a pena lembrar que, em geral, objetos `string` são preferíveis aos seus progenitores baseados em `char*`; por isso, o nome do autor (`authorName`) costuma ser melhor definido da seguinte forma:

```
const std::string authorName("Scott Meyers");
```

O segundo caso especial diz respeito às constantes específicas de uma determinada classe. Para limitar o escopo de uma constante a uma classe, você deve torná-la um membro, e, para garantir que existirá no máximo uma cópia da constante, você deve torná-la um membro *estático*:

```
class GamePlayer {
private:
  static const int NumTurns = 5;          // declaração da constante
  int scores[NumTurns];                    // uso da constante
  ...
};
```

O que você vê acima é uma *declaração* para o número de turnos (`NumTurns`), não uma definição. Normalmente, C++ requer que você forneça uma definição para qualquer coisa que usar, mas constantes específicas de uma classe que são estáticas e de um tipo integral (por exemplo, inteiros, caracteres, tipos booleanos) são uma exceção. Desde que não pegue os endereços delas, você pode declará-las e usá-las sem fornecer uma definição. Se você obtiver o endereço de uma constante de classe, ou se o compilador insistir, incorretamente, em uma definição mesmo que você não obtenha o endereço, você deve fornecer uma definição separada, como esta:

```
const int GamePlayer::NumTurns;          // definição de NumTurns; veja
                                         // abaixo porque não é dado nenhum valor
```

Você coloca isso em um arquivo de implementação, não em um arquivo de cabeçalho. Como o valor inicial de constantes de classes é fornecido quando a constante é declarada (por exemplo, `NumTurns` é inicializada como `5` quando ela é declarada), nenhum valor inicial é permitido no momento da definição.

Observe, a propósito, que não é possível criar uma constante específica de classe usando `#define`, porque `#define`s não respeitam escopo. Quando se define uma macro, ela aparece para todo o resto da com-

pilação (a menos que exista um `#undef` em algum lugar no caminho). Isso significa que `#define` não só não pode ser usado para constantes específicas de classe, como também não pode ser usado para fornecer algum tipo de encapsulamento, ou seja, não existe um `#define` "privado". Obviamente, os membros de dados constantes (`const`) podem ser encapsulados; `NumTurns` é encapsulado.

Os compiladores mais antigos podem não aceitar a sintaxe citada, porque costumava ser ilegal fornecer um valor inicial para um membro de classe estático no momento de sua declaração. Além disso, as inicializações na classe são permitidas apenas para tipos inteiros e constantes. Nos casos em que a sintaxe não pode ser usada, você coloca o valor inicial no momento da definição:

```
class CostEstimate {
private:
   static const double FudgeFactor;          // a declaração de constante de classe estática
   ...                                       // constant; vai no arquivo de cabeçalho
};
const double                                 // definação de classe estática
   CostEstimate::FudgeFactor = 1.35;         // constant; vai no arquivo de implementação
```

Isso é tudo o que você precisa na maior parte do tempo. A única exceção é quando você precisa do valor de uma constante de classe durante a compilação da classe, tal como na declaração do vetor `GamePlayer:scores` (que representa os pontos do jogador, e em que os compiladores insistem em saber o tamanho do vetor em tempo de compilação). Então, a maneira aceita para compensar os erros dos compiladores que (incorretamente) proíbem a especificação na classe de valores iniciais para constantes de classe inteiras estáticas é usar o que carinhosamente (e não pejorativamente) é conhecido como o "hack de enumeração". Essa técnica aproveita-se do fato de os valores de um tipo enumerado poderem ser usados nos locais em que se esperam valores inteiros, então `GamePlayer` (classe que representa um jogador) poderia ser muito bem definida assim:

```
class GamePlayer {
private:
   enum { NumTurns = 5 };                    // "hack de enumeração" — torna
                                             // NumTurns um nome simbólico para 5
   int scores[NumTurns];                     // OK
   ...
};
```

Vale a pena saber sobre o hack de enumeração, por diversas razões. Primeiro, de certa maneira o hack de enumeração se comporta mais como `#define` do que com uma constante (`const`), e, algumas vezes, é isso o que você quer. Por exemplo, é permitido pegar o endereço de uma constante, mas não é permitido pegar o endereço de uma enumeração, e, em geral, não é permitido pegar o endereço de um `#define`. Se você não quer deixar que as pessoas obtenham um ponteiro ou uma referência para uma de suas constantes inteiras, a enumeração é uma boa maneira de garantir essa restrição. (Para saber mais sobre como garantir restri-

ções de projeto por meio de decisões de codificação, consulte o Item 18.) Além disso, embora os bons compiladores não reservem espaço para objetos constantes de tipos inteiros (a menos que você crie um ponteiro ou uma referência para eles), compiladores medíocres podem fazê-lo, e talvez você não queira reservar memória para esses objetos. Como os `#define`s, as enumerações nunca resultam nesse tipo desnecessário de alocação de memória.

Uma segunda razão para conhecer o hack de enumeração é puramente pragmática. Diversos códigos o empregam, então, você precisa reconhecê-lo quando o vir. Na verdade, o hack de enumeração é uma técnica fundamental para a metaprogramação por templates (veja Item 48).

Voltando ao pré-processador, outro (des)uso comum da diretiva `#define` é usá-la para implementar macros que se parecem com funções, mas que não incorrem na sobrecarga de uma chamada de função. Eis uma macro que chama uma função `f` com o maior dos argumentos da macro:

```
// chama f com o máximo entre a e b
#define CALL_WITH_MAX(a, b) f((a) > (b) ? (a) : (b))
```

Macros como essa possuem tantas desvantagens que só pensar nelas já é doloroso.

Sempre que você escrever esse tipo de macro, tem que se lembrar de colocar todos os argumentos no corpo da macro entre parênteses. Caso contrário, você pode ter problemas quando alguém chamar a macro com uma expressão. Mas, mesmo que você acerte isso, veja as coisas estranhas que podem acontecer:

```
int a = 5, b = 0;
CALL_WITH_MAX(++a, b);          // a é incrementado duas vezes
CALL_WITH_MAX(++a, b+10);       // a é incrementado uma vez
```

Aqui, o número de vezes que `a` é incrementado antes de chamar `f` depende do que está sendo comparado a ele!

Felizmente, você não precisa continuar com essa macro sem sentido. Você pode obter toda a eficiência de uma macro, além de todo o comportamento previsível e a segurança de tipos de uma função regular, com o uso de um template para uma função internalizada (*inline function*) (veja o Item 30):

```
template<typename T>                        // como não sabemos o que é T,
inline void callWithMax(const T& a, const T& b)   // passamos de uma
{                                           // referência para uma
   f(a > b ? a : b);                        // constante – veja o Item 20
}
```

Esse template gera uma família completa de funções, e cada uma delas recebe dois objetos do mesmo tipo e chama `f` com o maior dos dois objetos. Não há necessidade de usar parênteses para os parâmetros dentro do corpo da função, nem de se preocupar em avaliar parâmetros várias vezes, etc. Além disso, como `callWithMax` (chama com o máximo) é uma

função real, ela obedece às regras de escopo e de acesso. Por exemplo, faz todo o sentido falar sobre uma função internalizada que é privada a uma classe. Em geral, não é possível fazer isso com uma macro.

Dada a disponibilidade de constantes, enumerações e internalizações, sua necessidade de usar o pré-processador (especialmente `#define`) é reduzida, mas não desaparece. `#include` permanece essencial, e `#ifdef/#ifndef` continuam a ter papéis importantes no controle da compilação. Ainda não é hora de aposentar o pré-processador, mas, definitivamente, você deve dar férias longas e frequentes a ele.

Lembretes

» Para as constantes simples, prefira objetos constantes ou enumerações a `#define`s.

» Para macros parecidas com funções, prefira funções internalizadas a `#define`s.

Item 3: Use `const` sempre que possível

Uma coisa incrível sobre o uso de `const` é que ele permite especificar uma restrição semântica – um objeto em particular *não* deve ser modificado –, e os compiladores garantem essa restrição. Ele permite que você comunique, tanto para os compiladores quanto para outros programadores, que um valor deve permanecer invariante. Sempre que for verdadeiro, você deve garantir que isso seja dito, porque, dessa forma, seu compilador ajuda a garantir que a restrição não seja violada.

A palavra-chave `const` é, evidentemente, versátil. Fora das classes, você pode usá-la para constantes de escopo global ou de escopo de espaço de nomes (veja o Item 2), bem como para objetos declarados estáticos (`static`) no escopo de um arquivo, de uma função ou de um bloco. Dentro das classes, você pode usá-la para membros de dados estáticos e para não estáticos. Para os ponteiros, você pode especificar se o ponteiro propriamente dito é constante, se o dado para o qual ele aponta é constante, ambos os casos ou nenhum deles:

```
char greeting[ ] = "Hello";
char *p = greeting;                          // ponteiro não constante,
                                             // dado não constante

const char *p = greeting;                    // ponteiro não constante,
                                             // dado constante

char * const p = greeting;                   // ponteiro constante,
                                             // dado não constante

const char * const p = greeting;             // ponteiro constante,
                                             // dado constante
```

Essa sintaxe não é tão instável quanto possa parecer. Se a palavra `const` aparecer na esquerda do asterisco, o que *é apontado* é constante; se a palavra `const` aparecer na direita do asterisco, o *ponteiro propriamente dito* é constante; se `const` aparecer em ambos os lados, ambos são constantes.*

Quando o que é apontado é constante, alguns programadores listam `const` antes do tipo; outros a listam após o tipo, mas antes do asterisco. Não existe diferença no significado, então as seguintes funções recebem o mesmo tipo de parâmetro:

```
void f1(const Widget *pw);          // f1 recebe um ponteiro para
                                    // um objeto Widget constante

void f2(Widget const *pw);          // f2 também
```

Já que as duas formas existem em código real, você deve se acostumar a ambas.

Os iteradores da STL são modelados como ponteiros, então um iterador (`iterator`) age de forma bastante parecia com um ponteiro `T*`. Declarar um iterador como constante (`iterator const`) é como declarar um ponteiro constante (ou seja, declarar um ponteiro `T* const`): o iterador não tem permissão para apontar para algo diferente, mas o item para o qual está apontando pode ser modificado. Se você quer um iterador que aponte para algo que não pode ser modificado (ou seja, o análogo em STL para um ponteiro `const T*`), você quer um iterador constante (`const_iterator`):

```
std::vector<int> vec;
...
const std::vector<int>::iterator iter =     // iter age como um T* const
   vec.begin( );
*iter = 10;                                 // OK, modifica o item para o
                                            // qual o ponteiro está apontando
++iter;                                     // erro! iter é constante

std::vector<int>::const_iterator cIter =    // cIter age como T* const
   vec.begin( );
*cIter = 10;                                // erro! cIter é constante
++cIter;                                    // ótimo, modifica cIter
```

Alguns dos usos mais poderosos de `const` vêm de sua aplicação às declarações de funções. Dentro de uma declaração de função, `const` pode se referir ao valor de retorno da função, aos parâmetros individuais e, para funções membro, à função como um todo.

Fazer uma função retornar um valor constante é, geralmente, inapropriado, mas, algumas vezes, fazer isso reduz incidência de erros do cliente sem abrir mão da segurança ou da eficiência. Por exemplo, considere a declaração da função `operator*` para números racionais que é explorada no Item 24:

```
class Rational { ... };
const Rational operator*(const Rational& lhs, const Rational& rhs);
```

*Algumas pessoas acham útil ler declarações de ponteiros da direita para a esquerda, por exemplo, para ler `const char * const p` como "`p` é um ponteiro constante para `char`s constantes."

Muitos programadores desconfiam quando veem isso pela primeira vez. Por que o resultado de `operator*` seria um objeto constante? Porque, se não fosse, os clientes conseguiriam cometer atrocidades como esta:

```
Rational a, b, c;
...
(a * b) = c;                    // invocar operator= no
                                // resultado de a*b!
```

Não sei por que um programador iria querer fazer uma atribuição ao produto de dois números, mas sei que muitos programadores tentaram fazer isso sem querer. Basta apenas um erro de digitação (e um tipo que possa ser convertido implicitamente em um `bool`):

```
if (a * b = c) ...              // ups, queríamos fazer uma comparação!
```

Esse código seria ilegal desde o início se `a` e `b` fossem um tipo predefinido. Uma das principais características de bons tipos definidos pelo usuário é que eles evitam incompatibilidades desnecessárias com os tipos predefinidos (veja o Item 18), e permitir atribuições ao produto de dois números parece ser bastante desnecessário para mim. Declarar o valor de retorno de `operator*` como constante (`const`) impede isso, e é por esse motivo que é "a coisa certa a fazer" nesse caso.

Não existe nada especialmente novo sobre parâmetros constantes – eles agem como objetos constantes locais, e você deve usar ambos sempre que puder. A menos que você precise poder modificar um parâmetro ou um objeto local, cuide para declará-lo usando `const`. Digitando apenas seis caracteres, você evita de encontrar erros irritantes como "eu queria digitar '==', mas, acidentalmente, digitei '='", que vimos antes.

Funções membro constantes

O objetivo de `const` nas funções membro é identificar quais funções membro podem ser invocadas em objetos constantes. Essas funções membro são importantes por duas razões. Primeiro, elas facilitam a compreensão da interface de uma classe. É importante saber quais funções podem modificar um objeto e quais não podem. Segundo, elas permitem trabalhar com objetos constantes. Esse é um aspecto crucial da escrita de código eficiente, pois, como o Item 20 explica, uma das maneiras fundamentais de melhorar o desempenho de um programa C++ é passar objetos por referência à constante. Essa técnica é viável apenas se existirem funções membro constantes com as quais podemos manipular os objetos resultantes qualificados com `const`.

Muitas pessoas subestimam o fato de que as funções membro que se diferem *apenas* por serem constantes podem ser sobrecarregadas, mas esse é um recurso importante de C++. Considere uma classe para representar um bloco de texto:

```
class TextBlock {
public:
...
   const char& operator[ ](std::size_t position) const      // operator[ ] para
   { return text[position]; }                               // objetos constantes

   char& operator[ ](std::size_t position)                  // operator[ ] para
   { return text[position]; }                               // objetos não constantes
private:
   std::string text;
};
```

As funções `operator[]` de `TextBlock` (bloco de texto) podem ser usadas assim:

```
TextBlock tb("Hello");
std::cout << tb[0];                                         // chama TextBlock::operator[ ]
                                                            // não constante
const TextBlock ctb("World");
std::cout << ctb[0];                                        // chama TextBlock::operator[ ]
                                                            // constante
```

Casualmente, os objetos constantes aparecem com mais frequência em programas reais por terem sido passados por ponteiros ou por referências para constantes. O exemplo de `ctb` (bloco de texto constante) acima é artificial. Este é mais realista:

```
void print(const TextBlock& ctb)                            // nesta função, ctb é constante
{
   std::cout << ctb[0];                                     // chama TextBlock:: operator[ ] constante
   ...
}
```

Ao sobrecarregar `operator[]` e ao dar às diferentes versões diferentes tipos de retorno, você pode ter `TextBlock`s constantes e não constantes tratados diferentemente:

```
std::cout << tb[0];                                         // ok – lendo um
                                                            // TextBlock não constante

tb[0] = 'x';                                                // fine — escrevendo um
                                                            // TextBlock não constante

std::cout << ctb[0];                                        // ok – lendo um
                                                            // TextBlock constante

ctb[0] = 'x';                                               // erro! gravando um
                                                            // TextBlock constante
```

Observe que o erro aqui só tem a ver com o *tipo de retorno* do `operator[]` que é chamado; as chamadas a `operator[]` propriamente ditas estão certas. O erro aparece da tentativa de fazer uma atribuição a um `const char&`, porque esse é o tipo de retorno da versão constante de `operator[]`.

Observe também que o tipo de retorno do `operator[]` não constante é uma *referência* a um `char` – um `char` propriamente dito não poderia ser

usado. Se `operator[]` realmente retornasse um `char` simples, sentenças como esta não compilariam:

```
tb[0] = 'x';
```

Isso ocorre porque nunca é permitido modificar o valor de retorno de uma função que retorna um tipo predefinido. Mesmo que fosse permitido, o fato de C++ retornar objetos por valor (veja o Item 20) significaria que uma *cópia* de `tb.text[0]` seria modificada, e não `tb.text[0]` propriamente dito, e esse não é o comportamento que você quer.

Paremos um momento para filosofar. O que significa, para uma função membro, ser constante (`const`)? Existem duas noções prevalentes: *constância bit a bit* (também conhecida como constância física) e *constância lógica*.

O grupo da constância bit a bit acredita que uma função membro é `const` se e apenas se ela não modificar nenhum dos membros de dados do objeto (excluindo aqueles que são estáticos), ou seja, se ela não modificar qualquer dos bits dentro do objeto. O bom dessa abordagem é que é fácil detectar violações: os compiladores só buscam atribuições a membros de dados. Na verdade, a constância bit a bit é a definição de C++ de constância, e uma função membro definida como `const` não pode modificar nenhum dos membros de dados não estáticos do objeto no qual ela é invocada.

Infelizmente, muitas funções membro que não agem de forma muito constante passam no teste bit a bit. Em particular, uma função que modifica o item para o qual um ponteiro *aponta* frequentemente não age de maneira constante. Mas, se apenas o *ponteiro* está no objeto, a função é constante bit a bit, e os compiladores não reclamarão. Isso pode levar a um comportamento inesperado. Por exemplo, suponhamos que temos uma classe parecida com `TextBlock` que armazena seus dados como um `char*` e não como `string`, porque ela precisa se comunicar por meio de uma API C que não entende objetos `string`.

```
class CTextBlock {
public:
  ...
  char& operator[ ](std::size_t position) const       // declaração inadequada (mas
  { return pText[position]; }                         // constante bit a bit)
                                                      // de operator[ ]
private:
  char *pText;
};
```

Essa classe declara (inadequadamente) `operator[]` como função membro constante, embora essa função retorne uma referência aos dados internos dos objetos (um tópico tratado em profundidade no Item 28). Coloque isso de lado e observe que a implementação de `operator[]` não modifica `pText` (um ponteiro para o texto) de modo algum. Como resultado, os compiladores geram códigos para `operator[]` com satisfação; trata-se, no fim das contas, de uma função constante bit a bit, e isso é

tudo o que os compiladores verificam. Mas veja o que isso permite que aconteça:

```
const CTextBlock cctb("Hello");              // declara objeto constante
char *pc = &cctb[0];                         // chama const operator[ ] para
                                             // obter um ponteiro para os dados de cctb
*pc = 'J';                                   // cctb agora tem um valor "Jello"
```

Certamente, há algo de errado quando você cria um objeto constante com um valor específico e invoca apenas funções membro constantes nele, mas mesmo assim modifica o seu valor!

Isso nos leva à noção de constância lógica. Os adeptos dessa filosofia – e você deveria estar entre eles – argumentam que uma função membro constante pode modificar alguns bits no objeto no qual é invocada, mas apenas de maneira que os clientes não possam detectar. Por exemplo, sua classe `CTextBlock` (bloco de texto) pode querer armazenar em *cache* o tamanho do bloco de texto sempre que ele for requisitado:

```
class CTextBlock {
public:
  ...
  std::size_t length( ) const;
private:
  char *pText;
  std::size_t textLength;                    // último tamanho calculado do bloco de texto
  bool lengthIsValid;                        // se o tamanho é atualmente válido
};
std::size_t CTextBlock::length( ) const
{
  if (!lengthIsValid) {
    textLength = std::strlen(pText);         // erro! impossível atribuir a textLength,
    lengthIsValid = true;                    // e lengthIsValid é uma função
  }                                          // membro constante
  return textLength;
}
```

Essa implementação de `length` com certeza não é constante bit a bit – tanto `textLength` (tamanho do texto) quanto `lengthIsValid` (tamanho é válido) podem ser modificados – embora ainda pareça que ela precisa ser válida para objetos `const CTextBlock`. Os compiladores discordam. Eles insistem na constância bit a bit. O que fazer?

A solução é simples: aproveite a flexibilidade de C++ relacionada ao uso de `const` conhecida como `mutable`. `mutable` libera membros de dados não estáticos das restrições da constância bit a bit:

```
class CTextBlock {
public:
   ...
   std::size_t length( ) const;
private:
   char *pText;
   mutable std::size_t textLength;                  // esses membros de dados podem
   mutable bool lengthIsValid;                      // ser modificados sempre, mesmo
};                                                  // em funções membro constantes
std::size_t CTextBlock::length( ) const
{
   if (!lengthIsValid) {
      textLength = std::strlen(pText);              // agora está ok
      lengthIsValid = true;                         // também ok
   }
   return textLength;
}
```

Evitando duplicação em funcões membro constantes e não constantes

O uso de `mutable` é uma boa solução para o problema "constância bit a bit não era o que eu tinha em mente", mas não resolve todas as dificuldades relacionadas às constantes. Por exemplo, suponhamos que `operator[]` em `TextBlock` (e em `CTextBlock`) não apenas retornasse uma referência ao caractere apropriado, mas também realizasse verificações de limites, fizesse *log* de informações de acesso, talvez até mesmo fizesse validação de integridade de dados. Colocar tudo isso em ambas as funções `operator[]` (constante e não constante) – e não se preocupar que agora temos funções internas implícitas de tamanho não trivial (veja o Item 30) – gera esse tipo de monstruosidade:

```
class TextBlock {
public:
   ...
   const char& operator[ ](std::size_t position) const
   {
      ...                                           // faz verificação de limites
      ...                                           // armazena em log os dados de acesso
      ...                                           // verifica a integridade de dados
      return text[position];
   }
   char& operator[ ](std::size_t position)
   {
      ...                                           // faz verificação de limites
      ...                                           // armazena em log os dados de acesso
      ...                                           // verifica a integridade de dados
      return text[position];
   }
private:
   std::string text;
};
```

Ai! Você consegue lidar com a duplicação de código e com as dores de cabeça envolvidas no tempo de compilação, manutenção e inchaço de código? Claro, é possível mover todo o código para verificação de limites, etc., em uma função membro separada (privada, naturalmente) que ambas as versões de `operator[]` chamam, mas você ainda tem as chamadas duplicadas para essa função e a sentença com o código de retorno (`return`) duplicada.

O que você quer fazer é implementar a funcionalidade de `operator[]` uma vez só e usá-la duas vezes. Ou seja, você quer ter uma versão de `operator[]` chamando a outra. Isso nos leva a converter explicitamente.

Em geral, as conversões explícitas (casts) não são recomendáveis. Dediquei um item inteiro para relatar isso (Item 27), mas a duplicação de código também não é uma maravilha. Nesse caso, a versão constante de `operator[]` faz exatamente o que a versão não constante faz; ela só tem um tipo de retorno qualificado com `const`. Converter explicitamente a constante no valor de retorno é seguro, nesse caso, porque qualquer um que tenha chamado a versão não constante de `operator[]` deve ter tido um objeto não constante em primeiro lugar. Caso contrário, ele não poderia ter chamado uma função não constante. Assim, fazer o `operator[]` não constante chamar a versão constante é uma maneira segura de evitar duplicação de código, mesmo que isso exija uma conversão explícita. Aqui está o código, que ficará mais claro depois que você ler a explicação a seguir:

```
class TextBlock {
public:
  ...
  const char& operator[ ](std::size_t position) const          // o mesmo de antes
  {
    ...
    ...
    ...
    return text[position];
  }
  char& operator[ ](std::size_t position)                      // agora simplesmente chama
                                                               // a constante op[ ]
  {
    return
      const_cast<char&>(                                       // converta explicitamente a constante
                                                               // para o tipo de retorno de op[ ];
        static_cast<const TextBlock&>(*this)                   // adiciona const ao tipo de *this
          [position]                                           // chama a versão constante de op[ ]
      );
  }
...
};
```

Como você pode ver, o código tem duas conversões explícitas, e não uma. Queremos que a versão não constante de `operator[]` chame a versão constante, mas, se dentro de `operator[]` não constante chamássemos apenas `operator[]`, estaríamos recursivamente chamando a nós mesmos. Isso é divertido apenas no primeiro milhão de vezes. Para evitar a recursão infinita, precisamos especificar que queremos chamar o `operator[]` constante; porém, não existe uma maneira direta para tanto. Em vez disso, convertemos explicitamente `*this` de seu tipo nativo de `TextBlock&` para `const TextBlock&`. Sim, usamos uma conversão explícita para *adicionar* `const`. Portanto, temos duas conversões: uma para adicionar `const` em `*this` (de forma que nossa chamada a `operator[]` chame a versão constante), e outra para remover `const` do valor de retorno de `const operator[]`.

A conversão explícita que adiciona `const` está apenas forçando uma conversão segura (de um objeto não constante em um objeto constante), assim usamos uma conversão explícita estática (`static_cast`) para isso. Aquela que remove `const` só pode ser realizada através de uma conversão explícita constante (`const_cast`) – não temos escolha aqui. (Tecnicamente, temos. Uma conversão no estilo de C também funcionaria, mas, como eu explico no Item 27, essas conversões explícitas raramente são a escolha correta. Se você não conhece `static_cast` ou `const_cast`, o Item 27 oferece uma visão geral.)

Além de tudo isso, a sintaxe é um pouco estranha porque estamos chamando um operador nesse exemplo. O resultado pode não ser bonito, mas tem o efeito desejado de evitar a duplicação de código ao implementar a versão não constante de `operator[]` em termos da versão constante. Só você consegue determinar se atingir esse objetivo vale a pena em termos de inclusão na complexidade da sintaxe, mas, com certeza, é importante conhecer a técnica de implementar uma função membro não constante em termos de sua função gêmea constante.

Mais importante ainda é saber que tentar fazer as coisas do modo contrário – fazer a versão constante chamar a versão não constante para evitar a duplicação – *não* é o melhor caminho. Uma função membro constante promete nunca modificar o estado lógico de seu objeto, mas uma função membro não constante não oferece essa garantia. Se você chamasse uma função não constante a partir de uma função constante, correria o risco de que o objeto que você prometeu não modificar fosse modificado. Por isso, fazer uma função membro constante chamar uma função membro não constante é algo errado: o objeto pode ser modificado. Na verdade, para que o código seja compilado, você precisaria ter usado um `const_cast` a fim de se livrar de `const` em `*this`, um sinal evidente de problemas. A sequência inversa de chamadas, aquela que usamos anteriormente, é segura – a função membro não constante pode fazer o que quiser com um objeto, portanto chamar uma função membro constante não impõe risco algum. É por isso que um `static_cast` funciona em `*this` nesse caso: não existem perigos relacionados às constantes.

Como apontei no início deste Item, `const` é uma coisa ótima. Em ponteiros e iteradores; em objetos referenciados por ponteiros, iteradores e referências;

em parâmetros de função e tipos de retorno; em locais variáveis; e em funções membro, `const` é um aliado poderoso. Use-o sempre que puder – você não vai se arrepender.

Lembretes

» Declarar algo como constante (usando `const`) ajuda os compiladores a detectar erros de uso. `const` pode ser aplicado a objetos em qualquer escopo, para parâmetros de funções e tipos de retorno e para funções membro como um todo.

» Os compiladores verificam a constância bit a bit, mas você deve programar usando a constância lógica.

» Quando as funções membro constantes e não constantes possuem implementações essencialmente idênticas, a duplicação de código pode ser evitada, fazendo a versão não constante chamar a versão constante.

Item 4: Certifique-se de que os objetos sejam inicializados antes do uso

C++ pode parecer um tanto errático sobre a inicialização de valores de objetos. Por exemplo, se você disser isto,

```
int x;
```

em alguns contextos é garantido que `x` será inicializado (para zero), mas em outros, não é. Se você disser isto,

```
class Point {
   int x, y;
};
...
Point p;
```

às vezes é garantido que os membros de dados de `p` serão inicializados (para zero), mas em outras vezes, não é. Se você está vindo de uma linguagem na qual os objetos não inicializados não podem existir, preste atenção, porque isso é importante.

Ler valores não inicializados leva a um comportamento indefinido. Em algumas plataformas, o mero ato de ler um valor não inicializado pode travar seu programa. Em geral, o resultado da leitura será de bits pseudoaleatórios, que poluirão o objeto para o qual você está enviando os bits, o que, no final das contas, leva a comportamentos de programas crípticos e muita depuração desagradável.

No entanto, existem regras que descrevem quando uma inicialização de objeto é garantida e quando não é. Infelizmente, as regras são complica-

das – na minha opinião muito complicadas para valer a pena memorizá-las. Em geral, se você está na parte C de C++ (Item 1) e a inicialização provavelmente incorrerá em custo de tempo de execução, não há garantias de que as inicializações ocorram. Se você se mover para as partes não C de C++, as coisas, algumas vezes, mudam. Isso explica por que não se garante que o conteúdo de um vetor (da parte C de C++) não seja necessariamente inicializado, mas um `vector` (da parte STL de C++) é.

A melhor maneira de lidar com esse estado aparentemente indeterminado de interesses é *sempre* inicializar seus objetos antes de usá-los. Para objetos não membros de tipos predefinidos, você precisará fazer isso manualmente. Por exemplo:

```
int x = 0;                                          // inicialização manual de um int

const char * text = "A C-style string";             // inicialização manual de um ponteiro
                                                    // (veja também o Item 3)

double d;                                           // "inicialização" pela leitura de dados
std::cin >> d;                                      // de um fluxo de entrada
```

Para quase todo o resto, a responsabilidade pela inicialização cai sobre os construtores. A regra aqui é simples: certifique-se de que todos os construtores inicializem tudo no objeto.

A regra é fácil de ser seguida, mas é importante não confundir atribuição com inicialização. Considere um construtor para uma classe que representa entradas em um livro de endereços:

```
class PhoneNumber { ... };
class ABEntry {                                     // ABEntry = "Address Book Entry"
public:
  ABEntry(const std::string& name, const std::string& address,
          const std::list<PhoneNumber>& phones);
private:
  std::string theName;
  std::string theAddress;
  std::list<PhoneNumber> thePhones;
  int numTimesConsulted;
};
ABEntry::ABEntry(const std::string& name, const std::string& address,
                 const std::list<PhoneNumber>& phones)
{
  theName = name;                                   // essas são todas atribuições
  theAddress = address;                             // não inicializações
  thePhones = phones;
  numTimesConsulted = 0;
}
```

Isso faz os objetos `ABEntry` (entradas no livro de endereços) receberem os valores que você espera, mas ainda assim não é a melhor abordagem. As regras de C++ estipulam que os membros de dados de um objeto sejam inicializados *antes* de entrar no corpo de um construtor. Dentro do construtor `ABEntry`, `theName` (o nome), `theAddress` (o endereço) e `thePhones` (os telefones) não estão sendo inicializados, estão sendo *atribuídos*. A inicialização ocorre antes – quando seus construtores padrão foram automaticamente chamados antes de entrar no corpo do construtor `ABEntry`. O mesmo não se aplica a `numTimesConsulted` (número de vezes consultado), porque é de um tipo primitivo. Para esse membro de dados, não existem garantias de que ele tenha sido inicializado antes da atribuição.

Uma maneira melhor de escrever o construtor `ABEntry` é usando a lista de inicialização de membros em vez de atribuições:

```
ABEntry::ABEntry(const std::string& name, const std::string& address,
                 const std::list<PhoneNumber>& phones)
: theName(name),
  theAddress(address),                    // essas são inicializações agora
  thePhones(phones),
  numTimesConsulted(0)
{ }                                       // o corpo do construtor está vazio agora
```

Esse construtor leva ao mesmo resultado final daquele descrito anteriormente, mas, em geral, é mais eficiente. A versão baseada primeiro em atribuição chamava os construtores padrão para inicializar `theName`, `theAddress` e `thePhones`, então, logo a seguir, atribuía novos valores sobre aqueles valores construídos por padrão. Todo o trabalho realizado nesses construtores padrão foi, dessa forma, perdido. A abordagem de lista de inicialização de membros evita esse problema, porque os argumentos na lista de inicialização são usados como argumentos de construção para os vários membros de dados. Nesse caso, `theName` é construído por cópia a partir de `name` (nome), `theAddress` é construído por cópia a partir de `address` (endereço) e `thePhones` é construído por cópia a partir de `phones` (telefones). Para a maioria dos tipos, uma única chamada para um construtor de cópia é mais eficiente – algumas vezes *muito* mais eficiente – do que uma chamada para o construtor padrão, seguido por uma chamada ao operador de cópia de atribuição.

Para objetos de tipos predefinidos como `numTimesConsulted`, não existe diferença no custo entre a inicialização e a atribuição, mas, por consistência, é frequentemente melhor inicializar tudo pela inicialização de membros. De modo parecido, você pode usar a lista de inicialização de membros mesmo que queira construir por padrão um membro de dados – só não especifique nada como argumento de inicialização. Por exemplo, se `ABEntry` possui um construtor sem parâmetros, poderia ser implementado assim:

```
ABEntry::ABEntry( )
: theName( ),                             // chama o construtor padrão de theName;
  theAddress( ),                          // faça o mesmo para theAddress;
  thePhones( ),                           // e para thePhones;
  numTimesConsulted(0)                    // mas inicialize explicitamente
{ }                                       // numTimesConsulted para zero
```

Como os compiladores chamarão automaticamente os construtores padrão para os membros de dados de tipos definidos pelo usuário quando esses membros de dados não possuem inicializadores na lista de inicialização de membros, alguns programadores consideram a abordagem acima um exagero. Isso é compreensível, mas ter uma política de sempre listar cada um dos membros de dados na lista de inicialização evita que você tenha que se lembrar quais membros de dados podem não ter sido inicializados se forem omitidos. Como `numTimesConsulted` é um tipo predefinido, por exemplo, deixá-lo de fora da lista de inicialização de membros poderia abrir as portas para um comportamento indefinido.

Algumas vezes, a lista de inicialização *deve* ser usada, mesmo para tipos predefinidos. Por exemplo, os membros de dados que são constantes ou referências devem ser inicializados; não podem receber valores por atribuição (veja também o Item 5). Para não precisar memorizar quando os membros de dados devem ser inicializados na lista de inicialização de membros e quando é opcional, a escolha mais fácil é *sempre* usar a lista de inicialização. Algumas vezes, isso é obrigatório, e frequentemente é mais eficiente do que as atribuições.

Muitas classes possuem construtores múltiplos, e cada um dos construtores possui sua própria lista de inicialização. Se existirem muitos membros de dados e/ou classes-base, a existência de listas múltiplas de inicialização introduz uma repetição indesejável (nas listas) e tédio (para os programadores). Nesses casos, não é irracional omitir entradas nas listas para os membros de dados em que as atribuições funcionem tão bem quanto as inicializações verdadeiras, movendo as atribuições para uma só função (geralmente privada) que todos os construtores chamam. Essa abordagem pode ser especialmente útil se os valores iniciais verdadeiros para os membros de dados forem lidos de um arquivo ou buscados de uma base de dados. Em geral, entretanto, é preferível a inicialização real de membros (através de uma lista de inicialização) à pseudoinicialização através da atribuição.

Um aspecto de C++ que não é errático é a ordem pela qual os dados de um objeto são inicializados. Essa ordem é sempre igual: as classes-base são inicializadas antes das classes derivadas (veja também o Item 12) e, dentro de uma classe, os membros de dados são inicializados na ordem em que foram declarados. Em `ABEntry`, por exemplo, `theName` será sempre inicializado primeiro, `theAddress` em seguida, `thePhones` então e `numTimesConsulted` por último. Isso é verdade mesmo que sejam listados em uma ordem diferente na lista de inicialização de membros (algo que infelizmente é permitido). Para evitar confusão dos leitores, bem como a possibilidade de alguns *bugs* realmente obscuros, sempre liste os membros na lista de inicialização na mesma ordem em que foram declarados na classe.

Tendo tomado o cuidado de inicializar explicitamente os objetos não membros de tipos predefinidos, e se assegurado que seus construtores vão inicializar suas classes-base e membros de dados usando a lista de inicialização de membros, existe apenas mais uma coisa para você se preocupar. Essa coisa é – respire fundo – a ordem de inicialização de objetos estáticos não locais definidos em unidades de tradução diferentes.

Vamos dividir essa sentença bit por bit.

Um *objeto estático* é um objeto que existe do momento em que é construído até o final do programa. Os objetos baseados em pilha e monte (*heap*) são, então, excluídos. Estão incluídos os objetos globais, os objetos definidos no escopo do espaço de nomes, os objetos declarados como estáticos (`static`) dentro de classes, os objetos declarados como estáticos (`static`) dentro de funções e os objetos declarados como estáticos (`static`) no escopo de arquivo. Os objetos estáticos dentro de funções são conhecidos como *objetos estáticos locais* (porque são locais a uma função), e os outros tipos de objetos estáticos são conhecidos como *objetos estáticos não locais.* Os objetos estáticos são destruídos quando o programa termina, ou seja, seus destrutores são chamados quando `main` (a função principal) termina sua execução.

Uma *unidade de tradução* é o código-fonte que gera um único arquivo objeto. É basicamente um arquivo-fonte, juntamente com todos os seus arquivos incluídos (através de `#include`).

O problema com o qual nos preocupamos aqui envolve ao menos dois arquivos-fonte compilados separadamente, cada um contendo ao menos um objeto estático não local (ou seja, um objeto que é global, no escopo do espaço de nomes, ou estático em uma classe ou no escopo do arquivo). E o problema real é que, se a inicialização de um objeto estático não local em uma unidade de tradução usa um objeto estático não local em uma unidade de tradução diferente, o objeto que ela usa pode não estar inicializado, porque *a ordem relativa de inicialização de objetos estáticos não locais definidos em diferentes unidades de tradução é indefinida.*

Um exemplo vai nos ajudar. Suponhamos que você tenha uma classe chamada `FileSystem` (sistema de arquivos) que faz os arquivos na Internet parecerem locais. Como sua classe faz o mundo parecer com um sistema único de arquivos, você pode criar um objeto especial estático ou no escopo do espaço de nomes representando o sistema de arquivos único:

```
class FileSystem {                          // do arquivo de cabeçalho de sua biblioteca
public:
  ...
  std::size_t numDisks( ) const;            // uma de muitas funções membro
  ...
};
extern FileSystem tfs;                      // declara objetos para os clientes usarem;
                                            // "tfs" = "o sistema de arquivos"; a
                                            // definição está em algum.cpp em sua biblioteca
```

Um objeto `FileSystem` decididamente é incomum, então o uso do objeto `tfs` (que representa o sistema de arquivos) antes de ser construído seria desastroso.

Agora, suponhamos que algum cliente crie uma classe para diretórios em um sistema de arquivos. Naturalmente, sua classe usa o objeto `tfs`:

```
class Directory {                                   // criado por cliente da biblioteca
public:
   Directory( params );
   ...
};
Directory::Directory( params )
{
   ...
   std::size_t disks = tfs.numDisks( );             // usa o objeto tfs
   ...
}
```

Além disso, suponhamos que esse cliente decida criar um único objeto `Directory` para arquivos temporários:

```
Directory tempDir( params );                        // diretório para arquivos temporários
```

Agora a importância da ordem de inicialização torna-se aparente: a menos que `tfs` seja inicializado antes de `tempDir` (diretório temporário), o construtor de `tempDir` tentará usar `tfs` antes que tenha sido inicializado. Mas `tfs` e `tempDir` foram criados por diferentes pessoas em diferentes momentos em diferentes arquivos-fonte – são objetos estáticos não locais, definidos em unidades de tradução diferentes. Como você pode se certificar de que `tfs` seja inicializado antes de `tempDir`?

Você não pode. Mais uma vez, *a ordem relativa da inicialização de objetos estáticos não locais definidos em diferentes unidades de tradução é indefinida.* Existe uma razão para isso. É difícil determinar a ordem "apropriada" na qual os objetos estáticos não locais precisam ser inicializados. Muito difícil. Praticamente impossível. Em sua forma mais geral – com múltiplas unidades de tradução e objetos estáticos não locais gerados por instanciações implícitas baseadas em templates (os quais também podem aparecer por meio de instanciações implícitas baseadas em templates) –, não só é impossível determinar a ordem correta de inicialização, como, em geral, não vale nem a pena buscar por casos especiais em que *é* possível determinar a ordem correta.

Felizmente, uma pequena mudança de projeto elimina o problema por completo. Tudo o que precisa ser feito é mover cada objeto estático não local para a sua própria função, onde é declarado estático (`static`). Essas funções retornam referências aos objetos que elas contêm. Os clientes chamam as funções em vez de se referirem aos objetos; em outras palavras, os objetos estáticos não locais são substituídos por objetos estáticos *locais*. (Os aficionados por padrões de projeto reconhecerão essa solução como uma implementação comum do padrão *Singleton**.)

Essa abordagem se baseia na garantia de C++ de que os objetos estáticos locais sejam inicializados quando a definição dos objetos é encontrada pela primeira vez durante uma chamada a essa função. Então, se você

* Na verdade, é apenas uma *parte* da implementação de um *Singleton*. Uma parte essencial de um *Singleton* que ignorei neste item é prevenir a criação de múltiplos objetos de um tipo em especial.

substituir acessos diretos a objetos estáticos não locais por chamadas a funções que retornem referências a objetos estáticos locais, você estará garantindo que as referências que obtiver se refiram a objetos inicializados. Como bônus, se você nunca chamar uma função emulando um objeto estático não local, nunca incorrerá no custo de construir e destruir o objeto, algo que não pode ser dito como verdadeiro para objetos estáticos não locais.

Aqui está a técnica aplicada tanto a `tfs` quanto a `tempDir`:

```
class FileSystem { ... };                    // como antes

FileSystem& tfs( )                           // isso substitui o objeto tfs; pode ser
{                                            // estática na classe FileSystem

   static FileSystem fs;                     // define e inicializa um objeto estático local
   return fs;                                // retorna uma referência a ele
}
class Directory { ... };                     // como antes
Directory::Directory( params )               // como antes, exceto que as referências
{                                            // a tfs são agora para tfs( )
   ...
   std::size_t disks = tfs( ).numDisks( );
   ...
}
Directory& tempDir( )                        // isso substitui o objeto tempDir; pode
{                                            // ser estática na classe Directory

   static Directory td;                      // define/inicializa objeto estático local
   return td;                                // retorna uma referência a ele
}
```

Os clientes desse sistema modificado programam exatamente como estavam acostumados, com a diferença de que agora se referem à função `tfs()` e à função `tempDir()`, e não ao membro de dados `tfs` e ao membro `tempDir`. Ou seja, usam funções que retornam referências aos objetos em vez de usar os objetos propriamente ditos.

As funções que retornam referências ditadas por esse esquema são sempre simples: definem e inicializam um objeto estático local na linha 1, retornam-no na linha dois. Essa simplicidade as torna excelentes candidatas para internalização, especialmente se forem chamadas frequentemente (Item 30). Por outro lado, por conterem objetos estáticos, são problemáticas em sistemas com múltiplas linhas de execução (*multithreaded*). Então, mais uma vez, qualquer tipo de objeto estático não constante – local ou não local – é um problema que espera a oportunidade de acontecer na presença de múltiplas linhas de execução. Uma maneira de lidar com esse problema é invocando manualmente todas as funções que retornam referências durante a parte de linha de execução única de inicialização do programa. Isso elimina condições de corrida relacionadas à inicialização.

É claro que a ideia de usar funções que retornam referências para prevenir problemas na ordem de inicialização depende de se ter, em primeiro

lugar, uma ordem de inicialização razoável para seus objetos. Se você tem um sistema no qual o objeto `A` deve ser inicializado antes do objeto `B`, mas a inicialização de `A` depende de `B` já ter sido inicializado, terá problemas e, francamente, você os merece. Se fugir desses cenários patológicos, entretanto, a abordagem descrita aqui servirá bem, pelo menos em aplicações com uma só linha de execução.

Para evitar o uso de objetos antes de eles serem inicializados, você precisa fazer apenas três coisas. Primeiro, inicializar manualmente os objetos não membros de tipos predefinidos. Segundo, usar as listas de inicialização de membros para inicializar todas as partes de um objeto. Por fim, projetar considerando incerteza da ordem de inicialização que aflige os objetos estáticos não locais definidos em unidades de tradução separadas.

Lembretes

» Inicialize manualmente os objetos de tipos predefinidos, porque C++ os inicializa apenas às vezes.

» Em um construtor, prefira usar a lista de inicialização de membros em vez de atribuições dentro do corpo do construtor. Liste os membros de dados na lista de inicialização na mesma ordem em que foram declarados na classe.

» Evite problemas com a ordem de inicialização entre as unidades de tradução substituindo os objetos estáticos não locais por objetos estáticos locais.

CAPÍTULO 2

CONSTRUTORES, DESTRUTORES E OPERADORES DE ATRIBUIÇÃO

Quase todas as classes que você escreve terão um ou mais construtores, um destrutor e um operador de atribuição por cópia. Pouca novidade aqui. Essas são suas funções "arroz com feijão", aquelas que controlam as operações fundamentais de tornar um novo objeto existente e garantir que ele seja inicializado, livrar-se de um objeto e certificar-se de que seja liberado apropriadamente, além de dar a ele um novo valor. Introduzir erros nessas funções levará a grandes (e desagradáveis) repercussões ao longo de suas classes; assim, é vital que você as crie corretamente. Neste capítulo, ofereço recomendações sobre como reunir as funções que formam a espinha dorsal das classes bem formadas.

Item 5: Saiba quais funções C++ escreve e chama silenciosamente

Quando uma classe vazia não é vazia? Quando C++ a alcança. Se você não os declarar, os compiladores declararão as suas próprias versões de um construtor de cópia, de um operador de atribuição por cópia e de um destrutor. Além disso, se você não declarar nenhum construtor, os compiladores também declararão um construtor padrão para você. Todas essas funções serão públicas e internalizadas (`inline`) – veja o Item 30. Como resultado, se você escrever

```
class Empty{ };
```

é essencialmente o mesmo que se você tivesse escrito o seguinte:

```
class Empty {
public:
  Empty( ) { ... }                                  // construtor padrão
  Empty(const Empty& rhs) { ... }                   // construtor por cópia

  ~Empty( ) { ... }                                 // destrutor – veja abaixo
                                                    // se ele é virtual

  Empty& operator=(const Empty& rhs) { ... }        // operador de atribuição por cópia
};
```

Essas funções são geradas apenas se forem necessárias, mas não precisa muito para precisar delas. O código a seguir fará cada uma das funções ser gerada:

```
Empty e1;                          // construtor padrão;
                                   // destrutor
Empty e2(e1);                      // construtor por cópia
e2 = e1;                           // operador de atribuição por cópia
```

Como os compiladores estão escrevendo funções para você, o que as funções fazem? Bem, o construtor padrão e o destrutor principalmente dão aos compiladores um lugar para colocar o código "nos bastidores", tais como a invocação de construtores e de destrutores de classes-base e membro de dados não estáticos. Observe que o destrutor gerado é não virtual (veja o Item 7), a menos que seja para uma classe que está herdando de uma classe-base que ela própria declare um destrutor virtual (nesse caso, a propriedade de ser virtual advém da classe-base).

Em relação ao construtor por cópia e ao operador de atribuição por cópia, as versões geradas pelo compilador simplesmente copiam cada membro de dados não estático do objeto-fonte para o objeto-alvo. Por exemplo, considere um *template* chamado `NamedObject` (objeto nomeado) que lhe permite associar nomes com objetos de um tipo T:

```
template<typename T>
class NamedObject {
public:
   NamedObject(const char *name, const T& value);
   NamedObject(const std::string& name, const T& value);
   ...

private:
   std::string nameValue;
   T objectValue;
};
```

Como um construtor é declarado em `NamedObject`, os compiladores não gerarão um construtor padrão. Isso é importante. Significa que, se você projetou cuidadosamente uma classe para que ela precise de argumentos para o construtor, você não precisa se preocupar com os compiladores sobrescrevendo sua decisão ao adicionarem cegamente um construtor que não possui argumentos.

`NamedObject` não declara nem um construtor por cópia nem um operador de atribuição por cópia, então os compiladores gerarão essas funções (se forem necessárias). Veja, então, o seguinte uso do construtor por cópia:

```
NamedObject<int> no1("Smallest Prime Number", 2);

NamedObject<int> no2(no1);         // chama um construtor por cópia
```

O construtor por cópia gerado pelos compiladores deve inicializar `no2.nameValue` e `no2.objectValue` (valor do nome e valor do objeto de `no2`) usando `no1.nameValue` e `no1.objectValue` (valor do nome e valor do objeto de `no1`), respectivamente. O tipo de `nameValue` é `string` (cadeia de caracteres), e o tipo padrão `string` possui um construtor por cópia, então `no2.nameValue` será inicializado através de uma chamada ao construtor

por cópia de `string` que passa `no1.nameValue` como seu argumento. Por outro lado, o tipo de `NamedObject<int>::objectValue` é `int` (inteiro – pois T é `int` para essa instanciação do template), e int é um tipo primitivo; portanto, `no2.objectValue` será inicializado por meio da cópia dos bits em `no1.objectValue`.

O operador de atribuição por cópia gerado pelo compilador para `NamedObject<int>` se comportaria essencialmente da mesma forma, mas, em geral, os operadores de atribuição por cópia gerados pelo compilador comportam-se como descrevi apenas quando o código resultante é legal e se possuir uma chance razoável de fazer sentido. Se houve falha em qualquer um desses testes, os compiladores se recusarão a gerar um `operator=` para a sua classe.

Por exemplo, suponhamos que `NamedObject` fosse definida como segue, em que `nameValue` é uma *referência* a uma cadeia, e `objectValue` é uma constante definida como `const T`:

```
template<ty pename>
class NamedObject {
public:
  // esse construtor não mais recebe um nome constante, porque nameValue
  // é agora uma referência a uma cadeia não constante. O construtor char* foi abandonado,
  // porque precisamos de uma string à qual fazemos referência.
  NamedObject(std::string& name, const T& value);

  ...                                   // como antes, assumir
                                        // que nenhum operato = é declarado
private:
  std::string& nameValue;               // esta agora é uma referência
  const T objectValue;                  // esta agora é constante
};
```

Agora, considere o que deveria acontecer aqui:

```
std::string newDog("Persephone");
std::string oldDog("Satch");

NamedObject<int> p(newDog, 2);          // quando originalmente escrevi este código,
                                        // nossa cadela Persephone estava quase
                                        // completando seu segundo aniversário

NamedObject<int> s(oldDog, 36);         // a cadela da família Satch (de minha
                                        // infância) teria 36 se
                                        // estivesse viva

p = s;                                  // o que deveria acontecer aos
                                        // membros de dados em p?
```

Antes da atribuição, `p.nameValue` (valor do nome de `p`) e `s.nameValue` (valor do nome de `s`) referem-se a objetos do tipo `string`, mas não aos mesmos objetos. Como a atribuição deveria afetar `p.nameValue`? Depois da atribuição, `p.nameValue` deveria fazer referência à string referenciada por `s.nameValue`, ou seja, a referência propriamente dita deve ser modificada? Caso deva, trata-se de algo novo, porque C++ não fornece uma maneira de

fazer uma referência a um objeto diferente. Por outro lado, será que o objeto `string` ao qual `p.nameValue` se refere deveria ser modificado, afetando outros objetos que mantêm ponteiros ou referências a essa `string`, ou seja, objetos que não estão envolvidos na atribuição? É isso o que o operador de atribuição por cópia gerado pelo compilador deve fazer?

Frente a esse problema de difícil solução, C++ recusa-se a compilar o código. Se você quer suportar a atribuição por cópia em uma classe que contém um membro que é uma referência, você mesmo deve definir o operador de atribuição por cópia. Os compiladores comportam-se de maneira similar para as classes que contêm membros constantes (como `objectValue` na classe modificada acima). Não é legal modificar os membros constantes, então os compiladores ficam inseguros a respeito de como tratá-los durante uma função de atribuição gerada implicitamente. Por fim, os compiladores rejeitam os operadores de atribuição por cópia em classes derivadas que herdam de classes-bases que declaram o operador de atribuição por cópia como privado (`private`). Afinal, os operadores de atribuição por cópia para classes derivadas supostamente devem tratar as partes da classe-base também (veja o Item 12), mas, ao fazer isso, eles certamente não podem invocar funções membros aos quais não tenham acesso ou direito de chamarem.

Lembrete

» Os compiladores podem gerar implicitamente o construtor padrão, o construtor de cópia, o operador de atribuição por cópia e o destrutor de uma classe.

Item 6: Desabilite explicitamente o uso de funções geradas pelo compilador que você não queira

Os corretores de imóveis vendem casas, e um sistema de software que ofereça suporte a esses corretores naturalmente teria uma classe para representar casas à venda:

```
class HomeForSale { ... };
```

Como todo corretor rapidamente dirá, cada propriedade é única – nenhuma é exatamente igual a outra. Sendo esse o caso, a ideia de fazer uma *cópia* de um objeto `HomeForSale` (casa à venda) faz pouco sentido. Como você pode copiar algo que é inerentemente único? Você provavelmente gostaria que as tentativas de copiar objetos `HomeForSale` não fossem compiladas:

```
HomeForSale h1;
HomeForSale h2;

HomeForSale h3(h1);                 // tentativa de copiar h1
                                    // – não deve ser compilada!

h1 = h2;                            // tentativa de copiar h2
                                    // – não deve ser compilada!
```

Infelizmente, impedir essa compilação não se faz diretamente. Normalmente, se você não quer que uma classe suporte um tipo em particular de funcionalidade, simplesmente não declara a função que a forneceria. Essa estratégia não funciona para o construtor de cópia e para o operador de atribuição por cópia, porque, conforme mencionado no Item 5, se você não os declara e alguém os tenta chamar, os compiladores os declararão para você.

Isso o põe em débito. Se você não declarar um construtor de cópia ou um operador de atribuição por cópia, os compiladores podem gerá-los para você. Logo, sua classe suporta cópias. Se, por outro lado, você declarar essas funções, sua classe ainda assim suportará cópias. Mas o objetivo aqui é *impedir* que as cópias sejam feitas!

A chave para a solução é que todas as funções geradas pelo compilador sejam públicas. Para impedir que essas funções sejam geradas, você deve declará-las você mesmo, mas não existe nada que force *você* a declará-las públicas. Em vez disso, declare o construtor de cópia e o operador de atribuição por cópia como *privados*. Ao declarar uma função membro explicitamente, você impede que os compiladores gerem sua própria versão, e, ao tornar a função privada, você impede que as pessoas a chamem.

Isso funciona na maioria dos casos. Esse esquema não é à prova de falhas, porque os membros e as funções amigas ainda podem chamar suas funções privadas. *A menos que* você seja esperto o suficiente para não *defini-las*. Então, se alguém inadvertidamente chamar uma delas, obterá um erro em tempo de ligação. Esse truque – declarar as funções membro privadas e, deliberadamente, não as implementar – é tão bem estabelecido que é usado para impedir cópias em diversas classes da biblioteca `iostream` de C++. Dê uma olhada, por exemplo, nas definições de `ios_base`, `basic_ios` e `sentry` em sua implementação da biblioteca padrão. Você descobrirá que, em cada caso, tanto o construtor de cópia quanto o operador de atribuição por cópia são declarados como privados e não são definidos.

É fácil aplicar esse truque para `HomeForSale`:

```
class HomeForSale {
public:
   ...

private:
   ...
   HomeForSale(const HomeForSale&);                    // apenas declarações
   HomeForSale& operator=(const HomeForSale&);
};
```

Você observará que eu omiti os nomes dos parâmetros das funções. Isso não é obrigatório, é apenas uma convenção comum. Afinal, as funções nunca serão implementadas, quanto mais utilizadas, então qual é o objetivo de especificar nomes de parâmetros?

Com a definição de classe acima, os compiladores vão impedir as tentativas dos clientes de copiar os objetos da classe `HomeForSale`; se você, inadver-

tidamente, tentar fazer isso em um membro ou em uma função amiga, o ligador reclamará.

É possível mover o erro em tempo de ligação para que ele ocorra em tempo de compilação (sempre uma coisa boa – a detecção precoce de erros é melhor do que a tardia) declarando o construtor de cópia e o operador de atribuição por cópia como privados não em `HomeForSale` propriamente dita, mas em uma classe-base especificamente projetada para impedir as cópias. A classe-base é a simplicidade em pessoa:

```
class Uncopyable {
protected:                                          // permite a construção
   Uncopyable( ) { }                                // e a destruição de
   ~Uncopyable( ) { }                               // objetos derivados...
private:
   Uncopyable(const Uncopyable&);                   // ...mas impede cópias
   Uncopyable& operator=(const Uncopyable&);
};
```

Para evitar que os objetos da classe `HomeForSale` sejam copiados, tudo o que precisamos fazer é herdar de `Uncopyable` (não copiável):

```
class HomeForSale: private Uncopyable {             // a classe não declara mais um
   ...                                              // construtor de cópia, nem um
};                                                  // operador de atribuição por cópia
```

Isso funciona porque os compiladores tentarão gerar um construtor de cópia e um operador de atribuição por cópia se alguém – mesmo um membro ou uma função amiga – tentar copiar um objeto `HomeForSale`. Como o Item 12 explica, as versões geradas pelo compilador dessas funções tentarão chamar suas funções respectivas na classe-base, e essas chamadas serão rejeitadas, porque as operações de cópia são privadas na classe-base.

A implementação e o uso de `Uncopyable` possuem algumas sutilezas, como o fato de a herança de `Uncopyable` não precisar ser pública (ver os Itens 32 e 39) e que o destrutor de `Uncopyable` não precisa ser virtual (veja o Item 7). Como `Uncopyable` não possui nenhum dado, ela se qualifica para a otimização da classe-base vazia descrita no Item 39, mas, como é uma classe-base, o uso dessa técnica pode levar à herança múltipla (Item 40). A herança múltipla, por sua vez, pode, algumas vezes, desabilitar a otimização da classe-base vazia (mais uma vez, veja o Item 39). Em geral, você pode ignorar essas sutilezas e só usar `Uncopyable` como mostrado, porque ela funciona precisamente como propagandeada. Você também pode usar a versão disponível em Boost (veja o Item 55). Aquela classe é chamada de `noncopyable`. É uma boa classe, eu só acho que o nome é um pouco *não* natural.

Lembrete

» Para desabilitar as funcionalidades fornecidas automaticamente pelos compiladores, declare as funções membro correspondentes como privadas e não forneça nenhuma implementação. Usar uma classe-base tal como `Uncopyable` é uma maneira de fazer isso.

Item 7: Declare os construtores como virtuais em classes-base polimórficas

Existem diversas maneiras de acompanhar o tempo, então seria razoável criar uma classe-base `TimeKeeper` (contador de tempo) juntamente com classes derivadas para abordagens diferentes para a contagem de tempo:

```
class TimeKeeper {
public:
   TimeKeeper( );
   ~TimeKeeper( );
   ...
};

class AtomicClock: public TimeKeeper { ... };

class WaterClock: public TimeKeeper { ... };

class WristWatch: public TimeKeeper { ... };
```

Muitos clientes vão querer ter acesso ao tempo sem se preocupar com os detalhes e como ele é calculado; então, uma *função fábrica* – que retorna um ponteiro da classe-base para um objeto recém-criado de classe derivada – pode ser usada para retornar um ponteiro para um objeto de acompanhamento de tempo:

```
TimeKeeper* getTimeKeeper( );          // retorna um ponteiro para um objeto
                                       // alocado dinamicamente de uma
                                       // classe derivada de TimeKeeper
```

Para manter as convenções das funções fábrica, os objetos retornados por `getTimeKeeper` (obtém o contador de tempo) estão no monte; então, para evitar vazamento de memória e de outros recursos, é importante que cada objeto retornado seja liberado de maneira apropriada:

```
TimeKeeper *ptk = getTimeKeeper( );    // obtém dinamicamente o objeto alocado
                                       // da hierarquia de TimeKeeper
...                                    // use-o
delete ptk;                            // libere-o para evitar vazamento de recursos
```

O Item 13 explica que esperar do cliente a realização da exclusão pode acarretar erros, e o Item 18 explica como a interface para a função fábrica pode ser modificada para impedir erros comuns dos clientes, mas essas preocupações são secundárias aqui, porque, neste item, tratamos de uma fraqueza mais fundamental do código acima: mesmo que os clientes façam tudo certo, não existe uma maneira de saber como o programa se comportará.

O problema é que `getTimeKeeper` retorna um ponteiro para um objeto de uma classe derivada (por exemplo, `AtomicClock` – relógio atômico); esse objeto está sendo apagado com um ponteiro da classe-base (ou seja, um ponteiro `TimeKeeper*`), e a classe-base (`TimeKeeper`) possui um *destrutor não virtual*. Essa é uma receita para o desastre, porque C++ especifica que, quando um objeto de uma classe derivada é liberado por

meio de um ponteiro para uma classe-base com um destrutor não virtual, os resultados são indefinidos. O que geralmente acontece em tempo de execução é que a parte derivada do objeto nunca é destruída. Se uma chamada a `getTimeKeeper` retornasse um ponteiro para um objeto `AtomicClock`, a parte `AtomicClock` do objeto (ou seja, os membros de dados declarados na classe `AtomicClock`) provavelmente não seria destruída, nem o destrutor de `AtomicClock` seria executado. Entretanto, a parte da classe-base (ou seja, a parte `TimeKeeper`) em geral seria destruída, levando a um curioso objeto "parcialmente destruído". Esta é uma maneira excelente de vazar recursos, corromper estruturas de dados e gastar um bocado de tempo com um depurador.

Eliminar o problema é simples: dê à classe-base um destrutor virtual. Então, apagar um objeto da classe derivada fará exatamente o que você quer. O objeto inteiro será destruído, incluindo todas as suas partes derivadas:

```
class TimeKeeper {
public:
   TimeKeeper( );
   virtual ~TimeKeeper( );
   ...
};

TimeKeeper *ptk = getTimeKeeper( );

...
delete ptk;                                    // agora se comporta corretamente
```

Classes-base como `TimeKeeper` geralmente contêm funções virtuais além de seu destrutor, pois o propósito das funções virtuais é permitir a personalização das implementações das classes derivadas (veja o Item 34). Por exemplo, `TimeKeeper` poderia ter uma função virtual, `getCurrentTime` (obtém o tempo atual), que seria implementada diferentemente nas várias classes derivadas. Qualquer classe com funções virtuais deve, quase certamente, ter um destrutor virtual.

Se uma classe *não* contiver funções virtuais, isso frequentemente indica que ela não deve ser usada como classe-base. Quando se pretende que uma classe não seja estendida, tornar seu destrutor virtual costuma ser uma má ideia. Considere uma classe para representar pontos em um espaço bidimensional:

```
class Point {                                  // um ponto 2D
public:
   Point(int xCoord, int yCoord);
   ~Point( );

private:
   int x, y;
};
```

Se um `int` ocupa 32 bits, um objeto `Point` (ponto) pode caber em um registrador de 64 bits. Além disso, esse objeto `Point` pode ser passado como uma quantidade de 64 bits para as funções escritas em outras lin-

guagens, como C ou FORTRAN. Se o destrutor de `Point` torna-se virtual, entretanto, a situação muda.

A implementação de funções virtuais requer que os objetos carreguem informações que podem ser usadas em tempo de execução para determinar quais funções virtuais devem ser invocadas no objeto. Essa informação, em geral, tem a forma de um ponteiro chamado vptr ("ponteiro para a tabela virtual, ou "virtual table pointer"). O vptr aponta para um vetor de ponteiros de funções chamado de vtbl ("tabela virtual, ou "virtual table"); cada classe com funções virtuais possui uma vtbl associada. Quando uma função virtual é invocada em um objeto, a função realmente chamada é determinada pelo vptr de um objeto até a vtbl, buscando o ponteiro para a função apropriada na vtbl.

Os detalhes de como as funções virtuais são implementadas não são importantes. O que *é* importante é que, se a classe `Point` contém uma função virtual, os objetos desse tipo aumentarão de tamanho. Em uma arquitetura de 32 bits, eles irão de 64 bits (para os dois inteiros) até 96 bits (para os dois inteiros mais o vptr); em uma arquitetura de 64 bits, podem ir de 64 a 128 bits, porque os ponteiros nessas arquiteturas possuem tamanho de 64 bits. A inclusão de um vptr a `Point` aumentará, então, seu tamanho em 50 a 100%! Os objetos `Point` não mais cabem em um registrador de 64 bits. Além disso, os objetos `Point` em C++ não podem mais se parecer com a mesma estrutura declarada em outra linguagem tal como C, porque sua linguagem externa correspondente não terá o vptr. Como resultado, não é mais possível passar pontos para e a partir de funções escritas em outras linguagens, a menos que você compense explicitamente pelo vptr, o que é, na verdade, um detalhe de implementação e dessa forma não é portável.

A moral da história, aqui, é que declarar desnecessariamente todos os destrutores como virtuais é tão errado quanto nunca declará-los como virtuais. Na verdade, muitas pessoas resumem a situação da seguinte maneira: declare um destrutor virtual em uma classe se e somente se essa classe contiver ao menos uma função virtual.

É possível ser atingido pelo problema dos destrutores não virtuais mesmo na completa ausência de funções virtuais. Por exemplo, o tipo padrão `string` não contém funções virtuais, mas os programadores desavisados algumas vezes usam essa classe como classe-base da mesma forma:

```
class SpecialString: public std::string {          // péssima ideia! std::string possui
   ...                                             // um destrutor não virtual
};
```

À primeira vista, isso pode parecer inócuo, mas, se em algum lugar da aplicação você, de alguma forma, converter um ponteiro para `SpecialString`

(cadeia de caracteres especial) em um ponteiro para `string`, e usar `delete` no ponteiro para `string`, será instantaneamente transportado para o mundo do comportamento indefinido:

```
SpecialString *pss =new SpecialString("Impending Doom");
std::string *ps;
...
ps = pss;                                   // SpecialString* ⇒ std::string*
...
delete ps;                                  // indefinido! Na prática, os
                                            // recursos de SpecialString *ps
                                            // serão vazados, porque o
                                            // destrutor de SpecialString
                                            // não será chamado.
```

A mesma análise se aplica a qualquer classe que não possui um destrutor virtual, incluindo todos os tipos contêiner da STL (ou seja, `vector`, `list`, `set`, `tr1:unordered_map` – veja o Item 54, etc.). Se você alguma vez se sentir tentado a herdar de um contêiner padrão ou de qualquer outra classe com um destrutor não virtual, resista à tentação! (Infelizmente, C++ não oferece mecanismos de prevenção de derivação, como as classes `final` de Java ou as classes seladas `[sealed]` de C#.)

Ocasionalmente, pode ser conveniente dar um destrutor puramente virtual para uma classe. Lembre-se de que as funções virtuais puras resultam em classes *abstratas* – classes que não podem ser instanciadas (ou seja, você não pode criar objetos desse tipo). Algumas vezes, entretanto, você tem uma classe que gostaria que fosse abstrata, mas não tem função virtual pura. O que fazer? Bem, como há a intenção de que uma classe abstrata seja usada como classe-base, e como uma classe-base deve ter um destrutor virtual, e como também uma função puramente virtual leva a uma classe abstrata, a solução é simples: declare um destrutor puramente virtual na classe que você quer que seja abstrata. Aqui está um exemplo:

```
class AWOV {                    // AWOV = "Abstract w/o virtuals" (Abstrata sem funções virtuais)
public:
   virtual ~AWOV( ) = 0;        // declara um destrutor puramente virtual
};
```

Essa classe possui uma função virtual pura, então ela é abstrata e possui um destrutor virtual; com isso, você não precisa se preocupar com o problema do destrutor. Existe um detalhe, no entanto: você deve fornecer uma *definição* para o destrutor virtual puro:

```
AWOV::~AWOV( ) { }              // definição do destrutor puramente virtual
```

Os destrutores funcionam assim: o destrutor da classe mais derivada é chamado primeiro, então o destrutor de cada uma das classes-base é chamado. Os compiladores gerarão uma chamada para `~AWOV` a partir dos destrutores de

suas classes derivadas, então você precisa se certificar para fornecer um corpo para a função. Se você não o fizer, o ligador reclamará.

A regra para dar destrutores virtuais às classes-base aplica-se apenas àquelas polimórficas – a classes-base projetadas para permitir a manipulação de tipos da classe derivada por meio de interfaces da classe-base. `TimeKeeper` é uma classe-base polimórfica, porque esperamos ser capazes de manipular objetos `AtomicClock` e `WaterClock` (relógio de água), mesmo que tenhamos ponteiros do tipo `TimeKeeper` para eles.

Nem todas as classes-base são projetadas para a utilização de maneira polimórfica. Nem o tipo padrão `string`, por exemplo, nem os tipos contêiner da SQL, são projetados para serem classes-base de alguma forma, muito menos para ser classes-base polimórficas. Algumas classes são projetadas para uso como classes-base, mas não são projetadas para uso polimórfico. Essas classes – por exemplo, `Uncopyable` do Item 6 e `input_iterator_tag` da biblioteca padrão (veja o Item 47) – não são projetadas para permitir a manipulação de objetos da classe derivada através de interfaces da classe-base. Como resultado, elas não precisam de destrutores virtuais.

Lembretes

- As classes-base polimórficas devem declarar destrutores virtuais. Se uma classe possui quaisquer funções virtuais, ela deve ter destrutores virtuais.
- As classes não projetadas para serem classes-base ou não projetadas para uso de forma polimórfica não devem declarar destrutores virtuais.

Item 8: Impeça que as exceções deixem destrutores

C++ não proíbe que os destrutores emitam exceções, mas certamente desestimula a prática. Com boas razões. Considere o seguinte:

```
class Widget {
public:
  ...
  ~Widget( ) { ... }                    // assuma que aqui pode ser emitida uma exceção
};

void doSomething( )
{
  std::vector<Widget> v;
  ...
}                                       // v é automaticamente destruída aqui
```

Quando o vetor `v` é destruído, ele é responsável por destruir todos os `Widget`s que contém. Suponhamos que `v` possua dez `Widget`s dentro, e que, durante a destruição do primeiro, seja lançada uma exceção. Os outros nove `Widget`s ainda precisam ser destruídos (caso contrário, quais-

quer recursos que eles mantêm seriam vazados), então `v` deve invocar seus destrutores. Mas suponhamos que, durante essas chamadas, um segundo destrutor de `Widget` lance uma exceção. Agora existem duas exceções simultaneamente ativas, e isso já é demais para C++. Dependendo das condições precisas nas quais esses pares de exceções ativas são lançadas simultaneamente, a execução do programa ou termina ou leva a comportamento indefinido. Nesse exemplo, isso levaria a um comportamento indefinido. Isso também ocorreria usando qualquer outro contêiner da biblioteca padrão (por exemplo, `list`, `set`), qualquer contêiner em TR1 (veja o Item 54) ou mesmo um vetor. Não que os contêineres ou os vetores obrigatoriamente devam ter problemas. O término prematuro de programas ou o comportamento indefinido podem ser resultado da emissão de exceções por parte dos destrutores mesmo sem usar contêineres e vetores. C++ *não* gosta de destrutores que emitem exceções!

Isso é fácil o bastante para entender, mas o que fazer se o seu destrutor precisar realizar uma operação que pode falhar lançando uma exceção? Por exemplo, suponhamos que você esteja trabalhando com uma classe para conexões em bancos de dados:

```
class DBConnection {
public:
   ...
   static DBConnection create( );          // função que retorna um objeto de
                                           // DBConnection; os parâmetros foram
                                           // omitidos por questões de simplicidade
   void close( );                          // fecha a conexão; lança uma
};                                         // exceção se houver falha no fechamento
```

Para garantir que os clientes não se esqueçam de chamar `close` (fechar) em objetos `DBConnection` (conexão ao banco de dados), uma ideia razoável seria criar uma classe de gerenciamento de recursos para `DBConnection` que chame `close` em seu destrutor. Essas classes de gerenciamento de recursos são analisadas em detalhes no Capítulo 3, mas aqui basta considerar como se pareceria um destrutor para essa classe:

```
class DBConn {                             // classe para gerenciar objetos da classe
public:                                    // DBConnection
   ...
   ~DBConn( )                              // certifica que as conexões com a base
   {                                       // de dados são sempre fechadas
      db.close( );
   }
private:
   DBConnection db;
};
```

Isso permite que os clientes programem da seguinte forma:

```
{                                       // abre um bloco

  DBConn dbc(DBConnection::create( ));   // cria um objeto DBConnection
                                        // e repassa-o para um objeto
                                        // DBConn para gerenciá-lo

  ...                                   // use o objeto DBConnection
                                        // através da interface DBConn

}                                       // no final do bloco, o objeto
                                        // DBConnection é destruído,
                                        // chamando automaticamente close
                                        // no objeto DBConnection
```

Isso funciona, contanto que a chamada a `close` seja bem-sucedida; mas, se a chamada levar a uma exceção, o destrutor de `DBConn` (conexão de banco de dados) propagará essa exceção, ou seja, permitirá que ela deixe o destrutor. Isso é um problema, porque os destrutores que lançam exceções indicam problemas.

Existem duas maneiras principais para evitar o problema. O destrutor de `DBConn` poderia:

- **Terminar o programa** se `close` lançasse uma exceção, em geral chamando `abort` (abortar):

```
DBConn::~DBConn( )
{
  try { db.close( ); }
  catch (...) {
    crie uma entrada de log que a chamada a close falhou;
    std::abort();
  }
}
```

  Essa é uma opção razoável se o programa não puder continuar a ser executado após um erro ser encontrado durante a destruição. Essa maneira tem a vantagem de que, se fosse permitida a propagação da exceção do destrutor, isso levaria a um comportamento indefinido, impedindo que isso acontecesse. Ou seja, chamar `abort` pode evitar a ocorrência de comportamento indefinido.

- **Engolir a exceção advinda da chamada a** `close`:

```
DBConn::~DBConn( )
{
  try { db.close( ); }
  catch (...) {
    crie uma entrada de log em que a chamada a close falhou;
  }
}
```

  Em geral, engolir exceções não é uma boa ideia, porque suprime informações importantes – *algo falhou*! Algumas vezes, entretanto, é preferível engolir exceções a correr o risco do término prematuro do

programa ou a ocorrência de comportamento indefinido. Para que isso seja uma opção viável, o programa deve ser capaz de continuar de maneira confiável mesmo após ter sido encontrado um erro e o mesmo ter sido ignorado.

Nenhuma dessas abordagens é especialmente atrativa. O problema com ambas é que o programa não possui uma maneira de reagir à condição que levou ao lançamento de uma exceção em `close` em primeiro lugar.

Uma estratégia melhor é definir a interface de `DBConn` de forma que seus clientes tenham a oportunidade de reagir aos problemas que podem acontecer. Por exemplo, `DBConn` poderia oferecer ela mesma uma função `close`, dando aos clientes a chance de tratar as exceções que decorram dessa operação. Ela poderia também rastrear se sua `DBConnection` foi fechada, encerrando-a no destrutor caso ela não tenha sido fechada. Isso impediria o vazamento de uma conexão. Se a chamada a `close` fosse falhar no destrutor de `DBConn`, entretanto, voltaríamos às opções de terminar o programa ou de engolir a exceção:

```
class DBConn {
public:
  ...
  void close( )                                   // nova função para
  {                                               // uso dos clientes
    db.close( );
    closed = true;
  }
~DBConn( )
{
  if (!closed) {
    try {                                         // fecha a conexão se
      db.close( );                                // o cliente não o fez
    }
    catch (...) {                                 // se o fechamento falhar,
      make log entry that call to close failed;   // anote isso e termine
      ...                                         // ou engula a exceção
    }
  }
}
private:
  DBConnection db;
  bool closed;
};
```

Mover a responsabilidade de chamar `close` do destrutor de `DBConn` para o cliente de `DBConn` (com o destrutor de `DBConn` contendo uma chamada de "backup") pode parecer um desvio de danos inescrupuloso. Você pode até mesmo ver isso como uma violação ao Item 18, que recomenda que sejam criadas interfaces fáceis de serem usadas corretamente. Na verdade, não é um, nem outro. Se uma operação pode falhar ao lançar uma exceção, e existe a necessidade de tratar essa exceção, tal exceção *precisa vir de uma função que não seja um destrutor*. Isso porque os

destrutores que emitem exceções são perigosos, e estão sempre correndo o risco de término prematuro do programa ou a ocorrência de comportamento indefinido. Nesse exemplo, dizer aos clientes que eles mesmos chamem `close` não lhes impõe uma sobrecarga; dá a eles a oportunidade de tratar erros a que, de outra forma, não teriam chance de reagir. Se eles não encontrarem uma oportunidade útil (talvez porque acreditem que nenhum erro ocorreu na verdade), eles podem ignorá-la, confiando que o destrutor de `DBConn` chame `close` para eles. Se um erro acontecer nesse ponto – se o close *realmente* lançar uma exceção – eles não estarão em posição de reclamar se `DBConn` engolir a exceção ou terminar o programa. Afinal, eles tiveram a oportunidade de lidar com o problema e escolheram não fazê-lo.

Lembretes

» Os destrutores nunca devem emitir exceções. Se as funções chamadas em um destrutor lançarem exceções, o destrutor deve capturar quaisquer exceções e então engoli-las ou terminar o programa.

» Se as classes cliente precisarem reagir às exceções lançadas durante uma operação, a classe deve fornecer uma função regular (que não seja um destrutor) que realiza a operação.

Item 9: Nunca chame funções virtuais durante a construção ou a destruição

Iniciarei com uma recapitulação: você não deve chamar as funções virtuais durante a construção ou a destruição, porque as chamadas não farão o que você pensa; se fizerem, você, mesmo assim, ficará infeliz. Se você for um programador de Java ou C#, preste muita atenção a este item, porque esse é um lugar no qual essas linguagens vão para uma direção e C++ vai para outra.

Suponhamos que você tenha uma hierarquia de classes para modelar transações de ações, como, por exemplo, pedidos de compra, pedidos de venda, etc. É importante que essas transações sejam auditáveis; assim, cada vez que se cria um objeto de transação, é necessário criar uma entrada apropriada em um *log* de auditoria. Esta parece ser uma maneira razoável de lidar com o problema:

```
class Transaction {                                  // classe-base para todas
public:                                              // as transações
   Transaction( );

   virtual void logTransaction( ) const = 0;         // cria uma entrada de log
                                                     // dependente do tipo
   ...
};
```

```
Transaction::Transaction( )                          // implementação do construtor
{                                                    // da classe-base
   ...
   logTransaction( );                                // como ação final, cria
}                                                    // um log desta transação

class BuyTransaction: public Transaction {           // classe derivada
public:
   virtual void logTransaction( ) const;             // como criar entradas de log
                                                     // de transações deste tipo
   ...
};

class SellTransaction: public Transaction {          // classe derivada
public:
   virtual void logTransaction( ) const;             // como criar entradas de log
                                                     // de transações deste tipo
   ...
};
```

Considere o que acontece quando este código é executado:

```
BuyTransaction b;
```

Sem dúvida, um construtor de `BuyTransaction` (transação de compra) será chamado, mas, primeiro, deve ser chamado um construtor de `Transaction` (transação); partes da classe-base de objetos de classes derivadas são construídas antes das partes da classe derivada. A última linha do construtor de `Transaction` chama a função virtual `logTransaction` (gravar transação em *log*), mas é aqui que a surpresa aparece. A versão de `logTransaction` que é chamada é aquela em `Transaction`, e *não* a em `BuyTransaction` – mesmo que o tipo do objeto sendo criado seja `BuyTransaction`. Durante a construção da classe-base, as funções virtuais nunca vão para baixo nas classes derivadas. Em vez disso, os objetos comportam-se como se fossem do tipo base. Informalmente falando, durante a construção da classe-base, as funções virtuais não são virtuais.

Existe uma boa razão para esse comportamento aparentemente não intuitivo. Como os construtores da classe-base são executados antes dos construtores da classe derivada, os membros de dados da classe derivada não serão inicializados quando os construtores da classe-base forem executados. Se as funções chamadas durante a construção da classe-base descessem para as classes derivadas, as funções da classe derivada quase que certamente se refeririam a membros de dados locais, mas esses membros não teriam sido inicializados. Isso seria um passe livre para o comportamento indefinido e para sessões de depuração durante a madrugada. Chamar partes abaixo na hierarquia de um objeto que não foi inicializado é perigoso, então C++ não lhe dá uma forma de fazer isso.

Na verdade, é mais fundamental que isso. Durante a construção de um objeto da classe derivada, o tipo do objeto *é* o da classe-base. Não só as funções virtuais são resolvidas para a classe-base, como as partes da lin-

guagem que usam informações de tipo em tempo de execução (por exemplo, `dynamic_cast` – veja o Item 37 – e `typeid`) tratam o objeto como se fosse do tipo da classe-base. No nosso exemplo, embora o construtor de `Transaction` esteja sendo executado para inicializar a parte da classe básica de um objeto `BuyTransaction`, o objeto é do tipo `Transaction`. É assim que cada uma das partes de C++ o tratarão, e o tratamento faz sentido: as partes específicas de `BuyTransaction` não foram inicializadas ainda, então é mais seguro tratá-las como se não existissem. Um objeto só se torna um objeto da classe derivada quando a execução de um construtor da classe derivada tem início.

O mesmo raciocínio se aplica durante a destruição. Uma vez que um destrutor de uma classe derivada precise ser executado, os membros de dados do objeto da classe derivada assumem valores indefinidos, então C++ os trata como se não mais existissem. Na entrada do destrutor da classe-base, o objeto torna-se um objeto da classe-base, e todas as partes, funções virtuais, `dynamic_cast`s, etc. de C++ o tratam dessa forma.

No código de exemplo acima, o construtor de `Transaction` fez uma chamada direta a uma função virtual, uma violação clara da recomendação deste item. A violação é tão fácil de ver que alguns compiladores lançam um aviso sobre isso. (Outros não o fazem. Veja o Item 53 para uma discussão sobre avisos.) Mesmo sem esse aviso, o problema quase que certamente se tornaria visível antes da execução, porque a função `logTransaction` é puramente virtual em `Transaction`. A menos que ela tenha sido definida (improvável, mas possível – veja o Item 34), o programa não se ligaria: o ligador seria incapaz de encontrar a implementação necessária de `Transaction::logTransaction`.

Nem sempre é fácil detectar chamadas a funções virtuais durante a construção e a destruição. Se `Transaction` possuísse múltiplos construtores, cada um deles tendo de realizar um pouco do mesmo trabalho, seria uma boa prática de engenharia de software evitar a replicação de código colocando o código de inicialização comum, incluindo a chamada a `logTransaction`, em uma função de inicialização privada não virtual, chamada, digamos, `init` (inicia):

```
class Transaction {
public:
   Transaction( )
   { init( ); }                                   // chamada a não virtual...

   virtual void logTransaction( ) const = 0;
   ...

private:
   void init( )
   {
      ...
      logTransaction( );                          // ...isso chama uma função virtual
   }
};
```

Esse código é conceitualmente igual à versão anterior, mas é mais traiçoeiro, porque compila e liga sem reclamações. Nesse caso, como `logTransaction` é puramente virtual em `Transaction`, a maioria dos sistemas de tempo de execução abortaria o programa quando a função puramente virtual fosse chamada (em geral lançando uma mensagem sobre isso). Entretanto, se `logTransaction` fosse uma função virtual "normal" (ou seja, não puramente virtual) com uma implementação em `Transaction`, essa versão seria chamada, e o programa continuaria feliz, deixando-o na tentativa de descobrir porque a versão errada de `logTransaction` foi chamada quando foi criado um objeto de uma classe derivada. A única maneira de evitar esse problema é certificar-se de que nenhum de seus construtores ou destrutores chamem funções virtuais no objeto sendo criado ou destruído e que todas as funções que eles chamam obedeçam à mesma restrição.

Mas, como você *garante* que a versão apropriada de `logTransaction` seja chamada cada vez que for criado um objeto na hierarquia de `Transaction`? Chamar uma função virtual no objeto a partir do(s) construtor(es) de `Transaction` é, claramente, a maneira errada de fazer isso.

Existem diferentes maneiras de abordar esse problema. Uma delas é fazer de `logTransaction` uma função não virtual em `Transaction`, e depois exigir que os construtores da classe derivada passem a informação de *log* necessária para o construtor de `Transaction`. Essa função poderia, então, chamar com segurança a `logTransaction` não virtual. Algo como:

```
class Transaction {
public:
   explicit Transaction(const std::string& logInfo);

   void logTransaction(const std::string& logInfo) const;        // agora uma função
                                                                 // não virtual
   ...
};

Transaction::Transaction(const std::string& logInfo)
{
   ...
   logTransaction(logInfo);                                      // agora uma chamada,
}                                                                // não virtual

class BuyTransaction: public Transaction {
public:
   BuyTransaction( parameters )
   : Transaction(createLogString( parameters ))                  // passa informações
   { ... }                                                       // de log para o construtor
   ...                                                           // da classe-base
private:
   static std::string createLogString( parameters );
};
```

Em outras palavras, já que você não pode usar funções virtuais para descer na hierarquia a partir das classes-base durante a construção, pode compensar fazendo, em vez disso, as classes derivadas passarem as informações de construção necessárias para cima na hierarquia, para os construtores da classe-base.

Nesse exemplo, observe o uso da função (privada) estática `createLogString` (cria cadeia de *log*) em `BuyTransaction`. Usar uma função auxiliar para criar um valor a ser passado a um construtor da classe-base em geral é mais conveniente (e mais legível) do que procurar desvios na lista de inicialização de membros para dar à classe-base o que ela precisa. Ao transformar a função em estática, não existe perigo de, acidentalmente, se referir aos membros de dados ainda não inicializados do objeto `BuyTransaction` que está nascendo. Isso é importante, pois, em primeiro lugar, o fato de que esses membros de dados estarão em um estado indefinido faz com que a chamada de funções virtuais durante a construção e a destruição da classe-base não desça na hierarquia para as classes derivadas.

Lembretes

» Não chame as funções virtuais durante a construção ou a destruição, porque essas chamadas nunca vão para uma classe mais derivada do que aquela que está atualmente executando o construtor ou o destrutor.

Item 10: Faça com que os operadores de atribuição retornem uma referência para `*this`

Uma das coisas interessantes sobre as atribuições é que você pode encadeá-las:

```
int x, y, z;
x = y = z = 15;                                    // cadeia de atribuições
```

Algo interessante também é o fato de a atribuição ser associativa à direita; então, a cadeia de atribuição acima é analisada sintaticamente como:

```
x = (y = (z = 15));
```

Aqui, 15 é atribuído a `z`, o resultado dessa atribuição (o `z` atualizado) é atribuído a `y` e o resultado dessa atribuição (o `y` atualizado) é atribuído a `x`.

O operador de atribuição implementa isso através do retorno de uma referência para o argumento referente ao lado esquerdo da atribuição. Você deve seguir essa convenção ao implementar operadores de atribuição para as suas classes:

```
class Widget {
public:
  ...
```

```
Widget& operator=(const Widget& rhs)                    // o tipo de retorno é uma
{                                                       // referência à classe atual
    ...
    return *this;                                       // retorna o objeto do lado esquerdo
  }
  ...
};
```

Essa convenção aplica-se a todos os operadores de atribuição, não apenas à forma padrão mostrada acima. Logo:

```
class Widget {
public:
  ...
  Widget& operator+=(const Widget& rhs)                 // a convenção aplica-se a
  {                                                     // +=, -=, *=, etc.
    ...
    return *this;
  }

  Widget& operator=(int rhs)                            // ela se aplica mesmo que
  {                                                     // o tipo do parâmetro do
    ...                                                 // operador não seja convencional
    return *this;
  }
  ...
};
```

Essa é apenas uma convenção: o código que não a seguir será compilado. Entretanto, a convenção é seguida por todos os tipos predefinidos, bem como por todos os tipos na (ou que em breve estarão na – veja o Item 54) biblioteca padrão (por exemplo, `string`, `vector`, `complex`, `tr1::shared_ptr`, etc). Siga essa convenção, a menos que você tenha uma boa razão para fazer as coisas de maneira diferente.

Lembretes

» Faça com que os operadores de atribuição retornem uma referência para `*this`.

Item 11: Trate as autoatribuições em `operator=`

Ocorre uma autoatribuição quando um objeto é atribuído a si próprio:

```
class Widget { ... };

Widget w;

...

w = w;                                                  // autoatribuição
```

Isso parece tolice, mas é permitido, então tenha certeza de que seus clientes farão isso. Além do mais, a atribuição nem sempre é tão reconhecível. Por exemplo,

```
a[i] = a[j];                                    // autoatribuição em potencial
```

é uma autoatribuição se `i` e `j` tiverem o mesmo valor, e

```
*px = *py;                                      // autoatribuição em potencial
```

é uma autoatribuição se `px` e `py` apontarem para a mesma coisa. Essas autoatribuições menos óbvias são o resultado do uso de *apelidos*: ter mais de uma maneira de se referenciar a um objeto. Em geral, o código que opera em referências ou em ponteiros para múltiplos objetos do mesmo tipo precisa considerar que os objetos podem ser iguais. Na verdade, os dois objetos não precisam nem ser declarados como do mesmo tipo se forem da mesma hierarquia, porque uma referência ou um ponteiro para a classe-base pode se referenciar ou apontar para um objeto do tipo da classe derivada:

```
class Base { ... };

class Derived: public Base { ... };

void doSomething(const Base& rb,                // rb e *pd podem, na verdade,
                 Derived* pd);                  // ser o mesmo objeto
```

Se você seguir a recomendação dos Itens 13 e 14, sempre usará objetos para gerenciar os recursos, e se certificará de que seus objetos de gerenciamento de recursos se comportem bem quando copiados. Quando esse for o caso, seus operadores de atribuição provavelmente serão seguros em relação à autoatribuição sem você precisar pensar na questão. No entanto, se você mesmo tentar gerenciar os recursos (o que certamente terá de fazer se estiver escrevendo uma classe gerenciadora de recursos), pode cair na armadilha de liberar acidentalmente um recurso antes de ter terminado de usá-lo. Por exemplo, suponhamos que você crie uma classe que mantém um ponteiro puro para um bitmap alocado dinamicamente:

```
class Bitmap { ... };

class Widget {
  ...

private:
  Bitmap *pb;                                   // ponteiro para um objeto alocado no monte
};
```

Veja uma implementação de `operator=` que parece razoável à primeira vista, mas que é insegura na presença de uma autoatribuição. (Ela também não é segura em relação às exceções, mas trataremos disso logo mais.)

```
Widget&
Widget::operator=(const Widget& rhs)            // implementação insegura de operator=
{
  delete pb;                                    // para de usar o bitmap atual
  pb = new Bitmap(*rhs.pb);                     // começa a usar uma cópia do mapa
                                                // de bits do lado direito

  return *this;                                 // veja o Item 10
}
```

O problema da autoatribuição aqui é que, dentro do operador `operator=`, `*this` (o alvo da atribuição) e o lado direito (`rhs`) podem ser o mesmo objeto. Quando são, o delete não apenas destrói o bitmap para o objeto atual, como também destrói o bitmap para `rhs`. No final da função, o `Widget` – que não deveria ser modificado pela autoatribuição – se encontra mantendo um ponteiro para um objeto apagado!*

A maneira tradicional de impedir esse erro é verificar pela autoatribuição através de um *teste de identidade* no início de `operator=`:

```
Widget& Widget::operator=(const Widget& rhs)
{
  Bitmap *pNew = pNew = new Bitmap (*rhs.pb);      // teste de identidade: se for uma
                                                   // autoatribuição, não faça nada
  delete pb;
  pb = pNew;

  return *this;
}
```

Isso funciona, mas mencionei, acima, que a versão anterior de `operator=` não era apenas insegura em relação à autoatribuição: era também insegura em relação à ocorrência de exceções, e essa versão continua apresentando problemas com exceções. Em particular, se a expressão `new Bitmap` levar a uma exceção (seja porque não existe memória suficiente para a alocação, seja porque o construtor de cópia de `Bitmap` lança uma exceção), o `Widget` terminará mantendo um ponteiro para um `Bitmap` apagado[3]. Esses ponteiros são tóxicos. Você não pode apagá-los de maneira segura; não pode nem mesmo lê-los de maneira segura. Provavelmente, a única coisa segura que você pode fazer com eles é gastar muita energia depurando para tentar descobrir de onde vieram.

Felizmente, tornar `operator=` seguro em relação a exceções em geral também o torna seguro em relação à atribuição. Como resultado, é cada vez mais comum lidar com questões de autoatribuição ignorando-as, focando-se em conseguir segurança com as exceções. O Item 29 explora a segurança em relação a exceções em profundidade, mas, neste item, é suficiente observar que, em muitos casos, uma ordenação cuidadosa de sentenças pode levar à segurança em relação às exceções (e à segurança em relação à autoatribuição). Aqui, por exemplo, só temos de ser cuidadosos para não apagar `pb` até termos copiado o item para o qual ele aponta:

```
Widget& Widget::operator=(const Widget& rhs)
{
  Bitmap *pOrig = pb;              // lembre-se do pb original
  pb = new Bitmap(*rhs.pb);        // aponta pb para uma cópia do bitmap do lado direito
  delete pOrig;                    // apaga o pb original

  return *this;
}
```

*Provavelmente, as implementações de C++ podem modificar o valor de um ponteiro apagado (por exemplo, para null ou para algum outro padrão de bits especial), mas desconheço alguma que faça isso.

Agora, se "`new Bitmap`" lança uma exceção, `pb` (e o `Widget` dentro dele) permanece inalterado. Mesmo sem o teste de identidade, esse código trata da autoatribuição porque fazemos uma cópia do bitmap original, apontamos para a cópia que fizemos e, então, apagamos o bitmap original. Pode não ser a maneira mais eficiente de lidar com a autoatribuição, mas ela realmente funciona.

Se você estiver preocupado com a eficiência, pode colocar o teste de identidade de volta ao topo da função. Antes de fazer isso, entretanto, pergunte-se sobre a frequência com que espera que ocorram as autoatribuições, porque o teste não é gratuito. Ele torna o código (tanto o fonte quanto o objeto) um pouco maior e introduz um desvio no fluxo de controle, o que pode diminuir a velocidade de execução. A efetividade de algumas operações (como operações de busca antecipada de instruções [*prefetching*], uso de memória *cache* e *pipelining*) pode ser reduzida, por exemplo.

Uma alternativa à ordenação manual das sentenças em `operator=` para se certificar de que a implementação é segura tanto em relação às exceções quanto à autoatribuição é usar a técnica conhecida como "copiar e trocar" (*copy and swap*). Essa técnica está fortemente associada com a segurança em relação a exceções, então é descrita no Item 29. Entretanto, é uma maneira suficientemante comum para escrever `operator=` para valer a pena ver como essa implementação costuma se parecer:

```
class Widget {
  ...
  void swap(Widget& rhs);                    // troca os dados de *this e rhs;
  ...                                        // veja o Item 29 para obter detalhes
};

Widget& Widget::operator=(const Widget& rhs)
{
  Widget temp(rhs);                          // faz uma cópia dos dados de rhs

  swap(temp);                                // troca os dados de *this pelos dados da cópia

  return *this;
}
```

Uma variação nesse tema tira vantagem do fato de que (1) pode ser declarado um operador de atribuição por cópia de uma classe para receber os valores por valor e (2) passar algo por valor cria uma *cópia* disso (veja o Item 20):

```
Widget& Widget::operator=(Widget rhs)        // rhs é uma cópia do objeto passado como
{                                            // argumento – observe a passagem por valor

  swap(rhs);                                 // troca os dados de *this
                                             // pelos dados da cópia
  return *this;
}
```

Pessoalmente, me preocupo com o fato de que essa abordagem sacrifica a clareza para melhorar a inteligência do código, mas, ao mover a operação de cópia do corpo da função para a construção do parâmetro, os compiladores podem, algumas vezes, gerar código mais eficiente.

Lembretes

» Certifique-se de que `operator=` esteja bem-comportado quando um objeto for autoatribuído. Entre as técnicas para fazer isso estão a comparação dos endereços de origem e de destino, a ordenação cuidadosa das sentenças, e a cópia e troca.

» Certifique-se de que qualquer função que esteja operando em mais de um objeto se comporte corretamente se dois ou mais dos objetos forem iguais.

Item 12: Copie todas as partes de um objeto

Em sistemas orientados a objetos bem-projetados, que encapsulam as partes internas dos objetos, apenas duas funções copiam objetos: a adequadamente chamada de *construtor de cópia* e a *operador de atribuição por cópia*. Chamaremos essas funções de *funções de cópia*. O Item 5 observa que os compiladores gerarão as funções de cópia, se necessário, e explica que as versões geradas pelo compilador fazem precisamente o que você espera: elas copiam todos os dados do objeto que está sendo copiado.

Quando você declara suas próprias funções de cópia, está indicando aos compiladores que existe algo a respeito da implementação padrão que você não gosta. Os compiladores parecem ficar ofendidos com isso, e eles fazem sua retaliação de uma maneira curiosa: não dizem a você quando sua implementação está quase que certamente errada.

Considere uma classe que representa clientes, em que as funções de cópia foram manualmente escritas de forma que as chamadas a elas fossem armazenadas em log:

```
void logCall(const std::string& funcName);                    // cria uma entrada de log

class Customer {
public:
   ...
   Customer(const Customer& rhs);
   Customer& operator=(const Customer& rhs);
   ...

private:
   std::string name;
};
```

```
Customer::Customer(const Customer& rhs)
: name(rhs.name)                                    // copia os dados de rhs
{
   logCall("Customer copy constructor");
}

Customer& Customer::operator=(const Customer& rhs)
{
   logCall("Customer copy assignment operator");

   name = rhs.name;                                 // copia os dados de rhs

   return *this;                                    // veja o Item 10
}
```

Tudo aqui parece bem e, na verdade, tudo está bem – até outro membro de dados ser adicionado.

```
class Date { ... };                                 // para datas no tempo

class Customer {
public:
   ...                                              // como antes

private:
   std::string name;
   Date lastTransaction;
};
```

Neste ponto, as funções de cópia existentes estão realizando uma *cópia parcial*: estão copiando `name` (o nome do cliente), mas não `lastTransaction` (a data da última transação). Mesmo assim, a maioria dos compiladores não diz nada a respeito disso, nem mesmo no nível máximo de avisos (veja também o Item 53). Essa é a vingança deles por você estar escrevendo suas funções de cópia. Você rejeita as funções de cópia que eles escrevem, então eles não dizem a você que o seu código está incompleto. A conclusão é óbvia: se você adicionar um membro de dados em uma classe, você precisa se certificar de atualizar também as funções de cópia. (Você também precisa atualizar todos os construtores [veja os Itens 4 e 45] bem como quaisquer formas não padrão de `operator=` na classe (o Item 10 dá um exemplo). Se esquecer disso, os compiladores provavelmente não o lembrarão.)

Uma das maneiras mais prejudiciais em que essa questão pode acontecer é pela herança. Considere:

```
class PriorityCustomer: public Customer {           // uma classe derivada
public:
   ...
   PriorityCustomer(const PriorityCustomer& rhs);
   PriorityCustomer& operator=(const PriorityCustomer& rhs);
   ...

private:
   int priority;
};
```

```
PriorityCustomer::PriorityCustomer(const PriorityCustomer& rhs)
: priority(rhs.priority)
{
   logCall("PriorityCustomer copy constructor");
}

PriorityCustomer&
PriorityCustomer::operator=(const PriorityCustomer& rhs)
{
   logCall("PriorityCustomer copy assignment operator");

   priority = rhs.priority;

   return *this;
}
```

As funções de cópia de `PriorityCustomer` (cliente com prioridade) parecem estar copiando tudo em `PriorityCustomer`, mas olhe novamente. Sim, elas copiam os membros de dados que `PriorityCustomer` declara, mas cada `PriorityCustomer` também contém uma cópia dos membros de dados que ele herda de `Customer` (cliente), e esses membros de dados não estão sendo copiados! O construtor de cópia de `PriorityCustomer` não especifica os argumentos a serem passados para o construtor de sua classe-base (ou seja, não faz menção a `Customer` em sua lista de inicialização de membros); então, a parte `Customer` do objeto `PriorityCustomer` será inicializada pelo construtor de `Customer` não recebendo argumentos pelo construtor padrão (Considerando que exista um. Se não existir, o código não será compilado). Esse construtor realizará uma inicialização *padrão* para `name` e `lastTransaction`.

A situação é apenas um pouco diferente para o operador de atribuição por cópia de `PriorityCustomer`. Ele não faz tentativa alguma de modificar os membros de dados de sua classe-base, então permanecem iguais.

Sempre que estiver escrevendo funções de cópia para uma classe derivada, você deve também tomar o cuidado de copiar as partes da classe-base. Essas partes, em geral, são privadas, é claro (veja o Item 22), então você não pode acessá-las diretamente. Em vez disso, as funções de cópia da classe derivada devem invocar as funções correspondentes da classe-base:

```
PriorityCustomer::PriorityCustomer(const PriorityCustomer& rhs)
:  Customer(rhs),                          // invoca o construtor de cópia da classe-base
   priority(rhs.priority)
{
   logCall("PriorityCustomer copy constructor");
}

PriorityCustomer&
PriorityCustomer::operator=(const PriorityCustomer& rhs)
{
   logCall("PriorityCustomer copy assignment operator");

   Customer::operator=(rhs);               // atribui as partes da classe-base
   priority = rhs.priority;

   return *this;
}
```

O significado de "copie todas as partes" no título deste item deve estar claro agora. Quando você estiver escrevendo uma função de cópia, certifique-se de (1) copiar todos os membros de dados locais e (2) invocar a função de cópia apropriada em todas as classes-base também.

Na prática, as duas funções de cópia com frequência terão corpos similares, o que pode tentá-lo a evitar a duplicação de código fazendo uma função chamar a outra. Seu desejo de evitar duplicação de código é louvável, mas fazer uma função de cópia chamar outra é a maneira errada de conseguir isso.

Não faz sentido fazer o operador de atribuição por cópia chamar o construtor de cópia, porque você estará tentando construir um objeto que já existe. Isso é tão desprovido de sentido que não existe nem mesmo uma sintaxe. Existem sintaxes que *parecem* que você está fazendo isso, mas você não está; e existem sintaxes que realmente *fazem* isso de alguma forma nos bastidores, mas que corrompem seu objeto em algumas condições. Não vou mostrar nenhuma dessas sintaxes por esse motivo. Não é recomendável fazer com que o operador de atribuição por cópia chame o construtor de cópia.

Seguir o caminho inverso (fazer com que o construtor de cópia chame o operador de atribuição por cópia) também não tem sentido. Um construtor inicializa novos objetos, mas um operador de atribuição por cópia se aplica apenas a objetos que já foram inicializados. Realizar uma atribuição em um objeto que está em construção pode significar fazer algo em um objeto que não foi inicializado, ainda que faça sentido apenas para um objeto inicializado. Bobagem! Nem tente.

Em vez disso, se você achar que seu construtor de cópia e seu operador de atribuição por cópia possuem corpos de código similares, elimine a duplicação criando uma terceira função membro que ambos chamem. Em geral, essa função é privada e chamada de `init`. Essa estratégia é uma maneira segura e comprovada de eliminar duplicação de código em construtores de cópia e em operadores de atribuição por cópia.

Lembretes

» As funções de cópia devem ter a certeza de copiar todos os membros de dados de um objeto e todas as partes de suas classes-base.

» Não tente implementar uma das funções de cópia em termos da outra. Em vez disso, coloque a funcionalidade comum em uma terceira função que ambas chamem.

CAPÍTULO 3

GERENCIAMENTO DE RECURSOS

Depois de utilizar um recurso, você precisa retorná-lo para o sistema. Se não fizer isso, coisas ruins poderão acontecer. Em programas C++, o recurso mais usado é o da memória alocada dinamicamente (se você aloca memória e nunca a libera, ocorre um vazamento de memória). No entanto, a memória é apenas um dos muitos recursos que você precisa gerenciar. Entre os outros recursos comuns estão os descritores de arquivos, os cadeados de exclusão mútua, fontes e pincéis em interfaces gráficas com o usuário, conexões a bancos de dados e sockets de rede. Seja qual for o recurso, o mais importante é que ele seja liberado após o uso.

É difícil garantir isso manualmente em todas as situações, mas, quando você pensa em exceções, em funções com múltiplos caminhos de retorno e em programadores de manutenção modificando os sistemas de software sem compreender completamente o impacto de suas mudanças, torna-se evidente que as maneiras *ad hoc* de gerenciamento de recursos não são suficientes.

Este capítulo começa com uma abordagem bastante direta baseada em objetos para o gerenciamento de recursos fundamentada no suporte de C++ para construtores, destrutores e operações de cópia. A experiência tem mostrado que o respeito disciplinado a essa abordagem pode minimizar os problemas de gerenciamento de recursos. O capítulo então passa para itens dedicados especificamente ao gerenciamento de memória. Eles complementam os itens mais gerais que vêm antes, porque os objetos que gerenciam memória precisam saber como fazer isso de maneira apropriada.

Item 13: Use objetos para gerenciar recursos

Suponhamos que estivéssemos trabalhando com uma biblioteca para modelar investimentos (por exemplo, ações, títulos de crédito, etc.), dos quais vários herdam de uma classe raiz chamada `Investment` (investimento):

```
class Investment { ... };                    // classe raiz de uma hierarquia
                                             // de tipos de investimento
```

Além disso, suponhamos que a maneira como a biblioteca nos fornece objetos de investimento específicos é por uma função fábrica (veja o Item 7):

```
Investment* createInvestment( );          // retorna um ponteiro para um objeto alocado
                                          // dinamicamente na hierarquia de investimento;
                                          // o chamador deve apagá-lo (os parâmetros
                                          // foram omitidos por questões de simplicidade)
```

Como o comentário indica, os chamadores de `createInvestment` (criar investimento) são responsáveis por apagar o objeto que a função retorna quando já tiverem terminado de usá-lo. Considere, então, uma função `f` escrita para satisfazer essa obrigação:

```
void f( )
{
  Investment *pInv = createInvestment( );     // chama a função fábrica
  ...                                         // usa pInv
  delete pInv;                                // libera o objeto
}
```

Isso parece bom, mas existem diversas maneiras pelas quais `f` pode não conseguir apagar o objeto de investimento que recebe de `createInvestment`. Pode existir uma sentença de retorno (`return`) prematura em algum lugar da parte "..." da função. Se tal `return` fosse executado, o controle nunca alcançaria a sentença `delete` (liberar/apagar). Uma situação similar aconteceria se os usos de `createInvestment` e de `delete` estivessem em um laço e esse laço fosse prematuramente abandonado por meio de uma sentença `break` (sair do laço) ou `goto` (desvio incondicional). Por fim, alguma sentença dentro de "..." poderia lançar uma exceção. Se isso acontecesse, o controle mais uma vez não chegaria a `delete`. Independentemente de como `delete` fosse evitado, estaríamos não só vazando a memória que contém o objeto de investimento, como também quaisquer recursos mantidos por esse objeto.

Obviamente, uma programação cuidadosa poderia impedir esses tipos de erros, mas pense em como o código pode mudar ao longo do tempo. À medida que o software vai sendo mantido, alguém pode adicionar uma sentença `return` ou `continue` (continua o laço) sem compreender completamente as repercussões no resto da estratégia de gerência de recursos da função. Ou, até mesmo pior, a parte "..." de `f` pode chamar uma função que não costumava lançar uma exceção, mas agora, subitamente, começa a fazer isso depois que foi "melhorada". Simplesmente não é algo viável depender de que `f` sempre chegue à sua sentença `delete`.

Para se certificar de que o recurso retornado por `createInvestment` seja sempre liberado, precisamos colocar esse recurso dentro de um objeto cujo destrutor automaticamente o libere quando o controle deixar `f`. Na verdade, essa é metade da ideia deste item: ao colocar recursos dentro de objetos, podemos confiar na invocação automática de destrutores de C++ para nos certificarmos de que os recursos tenham sido liberados. (Vamos discutir a outra metade da ideia daqui a alguns momentos.)

Muitos recursos são dinamicamente alocados no monte e são usados apenas dentro de um único bloco ou função, e devem ser liberados quando o controle deixar esse bloco ou função. O recurso `auto_ptr` (ponteiro automático) da biblioteca padrão é feito para esse tipo de situação. Trata-se de um objeto que se parece com um ponteiro (um ponteiro *esperto*) cujo destrutor automaticamente chama `delete` a qualquer coisa para a qual ele aponta. Veja como `auto_ptr` poderia ser usado para impedir o vazamento de recursos em potencial de `f`:

```
void f( )
{
   std::auto_ptr<Investment> pInv(createInvestment( ));   // chama a função
                                                          // fábrica
   ...                                                    // use pInv
                                                          // como antes
}                                                         // apaga pInv automaticamente
                                                          // através do destrutor
                                                          // de auto_ptr
```

Esse exemplo simples demonstra os dois aspectos cruciais de usar objetos para gerenciar recursos:

- **Os recursos são adquiridos e imediatamente convertidos em objetos de gerenciamento de recursos.** Acima, o resultado retornado por `createInvestment` é usado para inicializar o `auto_ptr` que o gerenciará. Na verdade, a ideia de usar objetos para gerenciar recursos é comumente chamada de RAII (*Resource Acquisition Is Initialization*), pois é muito comum adquirir um recurso e inicializar um objeto de gerenciamento de recurso na mesma sentença. Algumas vezes, os recursos adquiridos são *atribuídos* aos objetos de gerenciamento de recursos, em vez de inicializá-los. De qualquer forma, cada um dos recursos é imediatamente convertido em um objeto de gerenciamento de recursos no momento em que é adquirido.

- **Os objetos de gerenciamento de recursos usam seus destrutores para garantir que os recursos sejam liberados.** Como os destrutores são chamados automaticamente quando um objeto é destruído (por exemplo, quando um objeto sai de escopo), os recursos são corretamente liberados, independentemente de como o controle deixa um bloco. As coisas podem ficar complicadas quando o ato de liberar recursos puder levar à geração de exceções, mas essa é uma questão tratada pelo Item 8, então não nos preocuparemos com ela aqui.

Uma vez que um `auto_ptr` apaga automaticamente tudo que aponta quando é destruído, é importante que nunca exista mais de um `auto_ptr` apontando para um objeto. Se existir mais de um, o objeto seria apagado mais de uma vez, o que colocaria seu programa em direção ao comportamento indefinido. Para impedir tais problemas, os `auto_ptr`s possuem uma característica incomum: copiá-los (através do construtor de cópia ou do operador de atribuição por cópia) os configura como nulo, e o ponteiro de cópia assume toda a propriedade sobre o recurso!

```
std::auto_ptr<Investment>                    // plnv1 aponta para
   plnv1(createInvestment( ));               // o objeto retornado
                                             // de createInvestment

std::auto_ptr<Investment> plnv2(plnv1);      // plnv2 aponta para
                                             // o objeto; plnv1 é agora nulo

plnv1 = plnv2;                               // agora plnv1 aponta para
                                             // o objeto, e plnv2 é nulo
```

Esse comportamento estranho de cópia, juntamente com o requisito subjacente de que os recursos gerenciados por `auto_ptr`s nunca podem ter mais de um `auto_ptr` apontando para eles, significa que os `auto_ptr`s não são a melhor maneira de gerenciar todos os recursos alocados dinamicamente. Por exemplo, os contêineres da STL requerem que seu conteúdo exiba comportamento de cópia "normal", então os contêineres de `auto_ptr`s não são permitidos.

Uma alternativa ao uso de um `auto_ptr` é um "ponteiro esperto de contagem de referência" (RCSP – *Reference-Counting Smart Pointer*). Um RCSP é um ponteiro esperto que mantém um acompanhamento de quantos objetos apontam para um recurso em particular e apaga automaticamente o recurso quando ninguém estiver apontando para ele. Dessa forma, os RCSPs oferecem um comportamento que é similar àquele da coleta de lixo. Diferentemente da coleta de lixo, entretanto, os RCSPs não podem quebrar os ciclos de referências (por exemplo, quando dois objetos de outra forma não usados apontam um para o outro).

A função `tr1::shared_ptr` (ponteiro compartilhado de TR1) é um RCSP (veja o Item 54), então você poderia escrever `f` da seguinte maneira:

```
void f( )
{
   ...
   std::tr1::shared_ptr<Investment>
      plnv(createInvestment( ));             // chama a função fábrica

   ...                                       // usa plnv como antes

}                                            // apaga plnv automaticamente através
                                             // do destrutor de shared_ptr
```

Esse código parece praticamente o mesmo que emprega `auto_ptr`, mas a cópia de `shared_ptr`s comporta-se muito mais naturalmente:

```
void f( )
{
   ...

   std::tr1::shared_ptr<Investment>          // plnv1 aponta para o
      plnv1(createInvestment( ));            // objeto retornado de
                                             // createInvestment

   std::tr1::shared_ptr<Investment>          // tanto plnv1 quanto plnv2
      plnv2(plnv1);                          // apontam para o objeto
```

```
    pInv1 = pInv2;                          // novamente, nada
                                            // mudou
    ...
}                                           // pInv1 e pInv2 são
                                            //destruídos, e o objeto
                                            // para o qual eles apontam é
                                            // automaticamente apagado
```

Já que copiar `tr1::shared_ptr`s funciona "como o esperado", eles podem ser usados em contêineres da STL e em outros contextos nos quais o comportamento de cópia não ortodoxo dos `auto_ptr`s é inapropriado.

Não se engane, no entanto. Este item não é sobre `auto_ptr`, `tr1::shared_ptr` ou qualquer outro tipo de ponteiro esperto – é sobre a importância de usar objetos para gerenciar recursos. `auto_ptr` e `tr1::shared_ptr` são apenas exemplos de objetos que fazem isso. (Para mais informações sobre `tr1::shared_ptr`, consulte os Itens 14, 18 e 54.)

`auto_ptr` e `tr1::shared_ptr` usam `delete` em seus destrutores, e não `delete []` (o Item 16 descreve a diferença). Significa que usar `auto_ptr` ou `tr1::shared_ptr` com vetores ou matrizes alocadas dinamicamente é uma ideia ruim, embora lamentavelmente, compile:

```
std::auto_ptr<std::string>                  // má ideia! A forma errada
  aps(new std::string[10]);                 // de delete será usada

std::tr1::shared_ptr<int> spi(new int[1024]);   // o mesmo problema
```

Você pode estar surpreso por descobrir que não existe nada como `auto_ptr` ou `tr1::shared_ptr` para vetores e matrizes alocados dinamicamente em C++, nem mesmo em TR1. Isso porque `vector` (vetor) e `string` podem praticamente sempre substituir vetores e matrizes alocadas dinamicamente. Se você ainda pensa que seria legal ter classes parecidas com `auto_ptr` e `tr1::shared_ptr` para vetores e matrizes, dê uma olhada em Boost (consulte o Item 55) – aí, sim, você ficará satisfeito de encontrar as classes `boost::scoped_array` e `boost:shared_array` que oferecem o comportamento que está procurando.

A recomendação deste item para usar objetos para gerenciar recursos sugere que, se você estiver liberando os recursos manualmente (ou seja, usando `delete` fora de uma classe de gerência de recursos), está fazendo algo errado. As classes de gerência de recursos predefinidas como `auto_ptr` e `tr1::shared_ptr` frequentemente fazem com que seguir os conselhos deste item seja fácil, mas, às vezes, você está usando um recurso em que essas classes pré-fabricadas não fazem o que você quer. Quando esse for o caso, você precisará construir suas próprias classes de gerenciamento de recursos. Isso não é algo horrível de fazer, mas envolve algumas sutilezas que você precisa considerar. Essas considerações são o tópico dos Itens 14 e 15.

Como comentário final, tenho de destacar que o tipo de retorno bruto de `createInvestment` é um convite para um vazamento de recursos, porque é muito fácil para os chamadores se esquecerem de chamar `delete` no pon-

teiro que recebem (mesmo que usem um `auto_ptr` ou `tr1::shared_ptr` para realizar o `delete`, eles ainda assim precisam se lembrar de armazenar o valor de retorno de `createInvestment` em um objeto ponteiro esperto). Combater esse problema pede uma modificação de interface a `createInvestment`, um tópico de que tratarei no Item 18.

Lembretes

» Para impedir vazamentos de recursos, use objetos RAII para adquirir recursos em seus construtores e liberá-los nos destrutores.

» Duas classes RAII úteis são `tr1::shared_ptr` e `auto_ptr`. `tr1::shared_ptr` é normalmente a melhor escolha, porque seu comportamento quando copiado é intuitivo. Copiar um `auto_ptr` o configura como nulo.

Item 14: Pense cuidadosamente no comportamento de cópia em classes de gerenciamento de recursos

O Item 13 introduziu a ideia de RAII (*Resource Acquisition is Inicialization*) como a espinha dorsal de classes de gerenciamento de recursos, e ele descreve como `auto_ptr` e `tr1::shared_ptr` são manifestações dessa ideia para recursos baseados no monte (*heap*). Nem todos os recursos são baseados no monte, entretanto, e, para esses recursos, os ponteiros espertos como `auto_ptr` e `tr1::shared_ptr` são geralmente inapropriados como manipuladores de recursos. Nesse caso, você provavelmente precisará criar suas próprias classes de gerenciamento de recursos de tempos em tempos.

Por exemplo, suponhamos que você esteja usando uma API de C que manipule objetos de exclusão mútua do tipo `Mutex`, oferecendo funções `lock` (trancar) e `unlock` (destrancar):

```
void lock(Mutex *pm);                    // tranca o mutex apontado por pm
void unlock(Mutex *pm);                  // destranca o mutex
```

Para se assegurar de nunca se esquecer de destrancar um `Mutex` que você tenha trancado, você provavelmente criaria uma classe para gerenciar trancamentos. A estrutura básica dessa classe é ditada pelo princípio RAII, segundo o qual os recursos são adquiridos durante a construção e liberados durante a destruição:

```
class Lock {
public:
   explicit Lock(Mutex *pm)
   : mutexPtr(pm)
   { lock(mutexPtr); }                   // adquire recurso
   ~Lock( ) { unlock(mutexPtr); }        // libera recurso
private:
   Mutex *mutexPtr;
};
```

Os clientes usam `Lock` (cadeado) na maneira RAII tradicional:

```
Mutex m;                        // define o mutex que você precisa usar
...
{                               // cria um bloco para definir a seção crítica
  Lock ml(&m);                  // tranca o mutex
  ...                           // realiza operações na seção crítica
}                               // destranca automaticamente o mutex
                                // no fim do bloco
```

Sem problemas com isso, mas o que aconteceria se um objeto `Lock` fosse copiado?

```
Lock ml1(&m);                   // tranca m
Lock ml2(ml1);                  // copia ml1 para ml2
                                // – o que deve acontecer aqui?
```

Esse é um exemplo específico de uma questão mais geral, uma questão com a qual um autor de uma classe RAII deve se confrontar: o que deve acontecer quando um objeto RAII é copiado? Na maioria das vezes, você vai querer uma das seguintes possibilidades:

- **Proibir cópias.** Em muitos casos, não faz sentido permitir que os objetos RAII sejam copiados. Provavelmente, isso é verdade para uma classe como `Lock`, porque raras vezes faz sentido ter cópias de primitivas de sincronização. Quando as cópias não fazem sentido para uma classe RAII, você deve proibi-las. O Item 6 explica como fazer isso: declare as operações de cópia como privadas. Para `Lock`, poderia ser tal como segue:

  ```
  class Lock: private Uncopyable {    // proibindo cópias
  public:                             // – veja o Item 6
    ...                               // como antes
  };
  ```

- **Fazer contagem de referências no recurso subjacente.** Algumas vezes, é desejável manter um recurso até que o último objeto que o esteja usando tenha sido destruído. Quando esse for o caso, copiar um objeto RAII deve incrementar o número de objetos que se referem ao recurso. Esse é o significado da "cópia" usada por `tr1::shared_ptr`.

  Muitas vezes, as classes RAII podem implementar o comportamento de cópia com contagem de referências através de um membro de dados `tr1::shared_ptr`. Por exemplo, se `Lock` quisesse empregar a contagem de referências, ela deveria mudar o tipo de `mutexPtr` (ponteiro para um objeto de exclusão mútua) de `Mutex*` para `tr1::shared_ptr<Mutex>`. Infelizmente, o comportamento padrão de `tr1::shared_ptr` é apagar o que é apontado por ele quando a contagem de referências vai a zero, e não é isso o que queremos. Quando tivermos terminado com um `Mutex`, queremos deschaveá-lo, e não apagá-lo.

Felizmente, `tr1::shared_ptr` permite a especificação de um "apagador" – uma função ou objeto de função que é chamada quando a contagem de referências cai a zero. (Essa funcionalidade não existe para `auto_ptr`, que *sempre* apaga seu ponteiro.) O apagador é um segundo parâmetro opcional para o construtor de `tr1::shared_ptr`, então o código se pareceria com o seguinte:

```
class Lock {
public:
   explicit Lock(Mutex *pm)          // inicializa shared_ptr com o
   : mutexPtr(pm, unlock)            // Mutex para apontar e a função unlock
   {                                 // como a apagadora

      lock(mutexPtr.get( ));         // veja o Item 15 para obter informações sobre o "get"
   }
private:
   std::tr1::shared_ptr<Mutex> mutexPtr;          // usa shared_ptr em vez
};                                                // de um ponteiro bruto
```

Nesse exemplo, observe como a classe `Lock` não mais declara um destrutor. Isso porque não há necessidade para tanto. O Item 5 explica que um destrutor de classe (independentemente de ele ser gerado pelo compilador ou de ser definido pelo usuário) automaticamente invoca os destrutores dos membros de dados não estáticos da classe. Nesse exemplo, é `mutexPtr`. Mas o destrutor de `mutexPtr` automaticamente chamará o apagador de `tr1::shared_ptr` – `unlock`, nesse caso – quando a contagem de referência do objeto de exclusão mútua cair para zero. (As pessoas que olharem o código-fonte provavelmente vão gostar de um comentário indicando que você não se esqueceu da destruição, e sim que está apenas se baseando no comportamento padrão gerado pelo compilador.)

- **Copiar o recurso subjacente.** Algumas vezes, você pode ter quantas cópias de um recurso quiser, e a única maneira pela qual você precisa de uma classe gerenciadora de recursos é se certificar de que cada cópia seja liberada quando você não estiver precisando mais dela. Nesse caso, copiar o objeto de gerência de recursos também pode implicar copiar o recurso que ele envolve. Ou seja, copiar um objeto de gerenciamento de recursos realiza uma "cópia profunda".

  Algumas implementações do tipo padrão `string` consistem em ponteiros para memória do monte, em que os caracteres que fazem parte da `string` são armazenados. Os objetos dessas `string`s contêm um ponteiro para a memória no monte. Quando se copia um objeto `string`, faz-se uma cópia tanto do ponteiro quanto da memória para o qual ele aponta. Essas `string`s exibem cópia em profundidade.

- **Transferir a propriedade do recurso subjacente.** Em raras ocasiões, você pode querer se certificar de que apenas um objeto RAII

se refira a um recurso puro, e de que, quando o objeto RAII for copiado, a propriedade do recurso seja transferida do objeto copiado para o objeto copiador. Como explicado no Item 13, esse é o significado da "cópia" usada por `auto_ptr`.

As funções de cópia (construtor de cópia e operador de atribuição por cópia) podem ser geradas pelos compiladores; então, a menos que a versão gerada pelo compilador faça o que você quer (o Item 5 explica o comportamento padrão), você mesmo precisará escrevê-las. Em alguns casos, você também vai querer suportar versões generalizadas dessas funções. Tais versões são descritas no Item 45.

Lembretes

» Copiar um objeto RAII envolve copiar o recurso que ele gerencia, então o comportamento de cópia do recurso determina o comportamento de cópia do objeto RAII.

» Os comportamentos de cópia de classes RAII usuais são desabilitar cópias e realizar contagem de referências, mas outros comportamentos são possíveis.

Item 15: Forneça acesso a recursos brutos em classes de gerenciamento de recursos

As classes de gerenciamento de recursos são maravilhosas. Constituem sua defesa contra os vazamentos de recursos – a ausência desses vazamentos é uma característica fundamental de sistemas bem projetados. Em um mundo perfeito, você se basearia nessas classes para todas as interações com recursos e nunca sujaria as mãos com o acesso direto a esses recursos. Mas o mundo não é perfeito. Muitas APIs referem-se diretamente aos recursos, então, a menos que você queira prever o uso dessas APIs (algo que raramente é prático), você terá que passar por cima dos objetos de gerenciamento de recursos e lidar com recursos brutos de tempos em tempos.

Por exemplo, o Item 13 introduz a ideia de usar ponteiros espertos como `auto_ptr` ou `tr1::shared_ptr` para manter o resultado de uma chamada a uma função fábrica tal como `createInvestment`:

```
std::tr1::shared_ptr<Investment> pInv(createInvestment( ));          // do Item 13
```

Suponhamos que uma função que você gostaria de usar ao trabalhar com objetos da classe `Investment` seja a seguinte:

```
int daysHeld(const Investment *pi);                    // retorna o número de dias que um
                                                       // investimento vem sendo mantido
```

Você gostaria de chamá-la da seguinte forma:

```
int days = daysHeld(plnv);                    // erro!
```

mas o código não é compilado: `daysHeld` (dias mantidos) precisa de um ponteiro `Investment*` bruto, mas você está passando um objeto do tipo `tr1::shared_ptr<Investment>`.

Você precisa encontrar uma maneira para converter um objeto da classe RAII (nesse caso, `tr1::shared_ptr`) no recurso bruto que ele contém (no caso, o `Investment*` subjacente). Existem duas maneiras gerais de se fazer isso: por meio de conversão explícita ou de conversão implícita.

Tanto `tr1::shared_ptr` quanto `auto_ptr` oferecem uma função membro `get` (obter) para realizar uma conversão explícita, ou seja, retornar (uma cópia do) ponteiro bruto dentro do objeto de ponteiro esperto:

```
int days = daysHeld(plnv.get( ));             // bom, passa o ponteiro bruto
                                              // em plnv para daysHeld
```

Como praticamente todas as classes de ponteiros espertos, `tr1::shared_ptr` e `auto_ptr` também sobrecarregam os operadores de desreferenciamento de ponteiros (`operator->` e `operator*`), e isso permite a conversão implícita para os ponteiros brutos subjacentes:

```
class Investment {                            // classe raiz para uma hierarquia
public:                                       // de tipos de investimentos
   bool isTaxFree( ) const;
   ...
};

Investment* createInvestment( );              // função fábrica

std::tr1::shared_ptr<Investment>              // faz com que tr1::shared_ptr
   pi1(createInvestment( ));                  // gerencie um recurso

bool taxable1 = !(pi1->isTaxFree( ));         // Acessa o recurso através
                                              // de operator->
...

std::auto_ptr<Investment> pi2(createInvestment( ));   // faz com que auto_ptr
                                                      // gerencie um
                                                      // recurso

bool taxable2 = !((*pi2).isTaxFree( ));       // Acessa o recurso através
                                              // de operator*
...
```

Já que às vezes é necessário obter o recurso bruto dentro de um objeto RAII, alguns projetistas de classes RAII facilitam ao oferecer uma função de conversão implícita. Por exemplo, considere essa classe RAII para fontes que são nativas para uma API C:

```
FontHandle getFont( );                        // de uma API C – os parâmetros foram
                                              // omitidos por questões de simplicidade

void releaseFont(FontHandle fh);              // da mesma API C
```

```
class Font {                                    // classe RAII
public:
   explicit Font(FontHandle fh)                 // adquirir recurso;
   : f(fh)                                      // usar passagem por valor,
   { }                                          // porque a API C faz isso

   ~Font( ) { releaseFont(f ); }                // liberar recurso

private:
   FontHandle f;                                // o recurso bruto (no caso, uma fonte)
};
```

Considerando-se que existe uma grande API em C relacionada com fontes que trata inteiramente de `FontHandle`s (tratadores de fontes), haverá uma necessidade frequente de converter objetos `Font` (fonte) em `FontHandle`s. A classe `Font` poderia oferecer uma função de conversão explícita como `get`:

```
class Font {
public:
   ...
   FontHandle get( ) const { return f; }        // função de conversão explícita
   ...
};
```

Infelizmente, isso exigiria que os clientes chamassem `get` cada vez que quisessem se comunicar com a API:

```
void changeFontSize(FontHandle f, int newSize);     // da API C

Font f(getFont( ));
int newFontSize;

...

changeFontSize(f.get( ), newFontSize);              // converte explicitamente de
                                                    // Font para FontHandle
```

Alguns programadores acham que a necessidade de requerer explicitamente essas conversões é suficiente para evitar o uso da classe. Isso, por sua vez, aumentaria a chance de vazamento de fontes, aquilo que a classe `FontHandle` foi projetada para impedir.

A alternativa é fazer `Font` oferecer uma função de conversão implícita para `FontHandle`:

```
class Font {
public:
   ...
   operator FontHandle( ) const                 // função de conversão implícita
   { return f; }
   ...
};
```

Isso faz com que as chamadas à API C sejam fáceis e naturais:

```
Font f(getFont( ));
int newFontSize;

...

changeFontSize(f, newFontSize);          // converte implicitamente de
                                         // Font para FontHandle
```

A desvantagem é que as conversões implícitas aumentam a chance de erros. Por exemplo, um cliente poderia criar acidentalmente um `FontHandle` quando o desejado fosse `Font`:

```
Font f1(getFont( ));
...

FontHandle f2 = f1;                      // ops! Queria copiar um objeto
                                         // Font, mas, em vez disso, converteu
                                         // implicitamente f1 em seu FontHandle
                                         // subjacente, então o copiou.
```

Agora o programa tem um `FontHandle` sendo gerenciado pelo objeto `f1`, mas o `FontHandle` também está disponível para uso direto como `f2`. Isso quase sempre é ruim. Por exemplo, quando `f1` for destruído, a fonte será liberada, e `f2` será um ponteiro solto.

A decisão de oferecer ou não uma conversão explícita de uma classe RAII para seu recurso subjacente (por meio de uma função membro `get`, por exemplo), ou de permitir a conversão implícita, depende das tarefas específicas para as quais a classe foi projetada e as circunstâncias para as quais se pretende que ela seja usada. O melhor projeto, provavelmente, é aquele que está de acordo com a recomendação do Item 18 de criar interfaces que sejam fáceis de usar corretamente e difíceis de serem usadas incorretamente. Com frequência, a função de conversão explícita como `get` é o caminho preferível, porque minimiza as chances de conversões de tipo não pretendidas. Algumas vezes, entretanto, a naturalidade de uso que decorre do uso de conversões de tipo implícitas direcionará a escolha para essa abordagem.

Pode lhe ter ocorrido que as funções que retornam o recurso bruto dentro de uma classe RAII são contrárias ao encapsulamento. Isso é verdade, mas não é o desastre de projeto que pode parecer à primeira vista. As classes RAII não existem para encapsular algo; existem para garantir que uma ação em particular – a liberação de recursos – ocorra. Se desejado, o encapsulamento do recurso pode ser colocado em camadas sobre essa funcionalidade primária, mas não é necessário. Além disso, algumas classes RAII combinam o encapsulamento de implementação verdadeiro com um encapsulamento muito fraco do recurso subjacente. Por exemplo, `tr1::shared_ptr` encapsula toda a parafernália para a contagem de referências, mas ainda assim oferece um acesso fácil ao ponteiro que ela contém. Como a maioria das classes bem projetadas, ela oculta o que

os clientes não precisam ver, mas disponibiliza as coisas que os clientes honestamente precisam acessar.

Lembretes

» As APIs frequentemente requerem acesso a recursos brutos, então cada classe RAII deve oferecer uma maneira de chegar ao recurso que elas gerenciam.

» O acesso pode ser feito através de conversão explícita ou conversão implícita. De um modo geral, a conversão explícita é mais segura, mas a conversão implícita é mais conveniente para os clientes da classe.

Item 16: Use a mesma forma nos usos correspondentes de `new` e `delete`

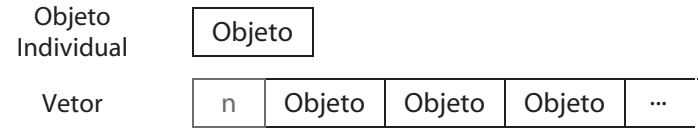
O que está errado com o código a seguir?

```
std::string *stringArray = new std::string[100];
...
delete stringArray;
```

Tudo parece estar em ordem. O `new` casa com o `delete`. Mesmo assim, algo está bem errado. O comportamento do programa é indefinido. No mínimo, 99 dos 100 objetos `string` apontados por `stringArray` (vetor de `string`s) não serão destruídos de modo adequado, porque é provável que seus destrutores nunca sejam chamados.

Quando você usa uma *expressão* `new` (ou seja, a criação dinâmica de um objeto pelo uso de `new`), duas coisas acontecem. Primeiro, a memória é alocada (através de uma função chamada `operator new` – veja os Itens 49 e 51). Segundo, um ou mais construtores são chamados para essa memória. Quando você emprega uma *expressão* `delete` (ou seja, usa `delete`), duas outras coisas acontecem: um ou mais destrutores são chamados para a memória, então a memória é liberada (através de uma função chamada `operator delete` – veja o Item 51). A grande questão aqui é: *quantos* objetos residem na memória sendo apagada? A resposta para essa pergunta determina quantos destrutores devem ser chamados.

Na verdade, a questão é mais simples: o ponteiro sendo apagado aponta para um só objeto ou para um vetor de objetos? É uma questão crítica, porque o layout de memória para objetos individuais é geralmente diferente do layout de memória para vetores. Em particular, a memória para um vetor normalmente inclui o tamanho do vetor, facilitando para `delete` saber quantos destrutores chamar. A memória para um objeto individual não possui essa informação. Você pode pensar nos diferentes layouts como segue, onde `n` é o tamanho do vetor:



Esse é apenas um exemplo, é claro. Os compiladores não são obrigados a implementar as coisas dessa forma, apesar de muitos o fazerem.

Quando você usa `delete` em um ponteiro, a única maneira para `delete` saber se as informações de tamanho estão lá é se você disser para ele. Se você usar colchetes em seu uso de `delete`, `delete` entende que um vetor está sendo apontado; caso contrário, ele entende que está sendo apontado um só objeto:

```
std::string *stringPtr1 = new std::string;
std::string *stringPtr2 = new std::string[100];
...
delete stringPtr1;                    // apaga um objeto
delete [ ] stringPtr2;                // apaga um vetor de objetos
```

O que aconteceria se você usasse a forma "`[]`" em `stringPtr1` (um ponteiro para uma `string`)? O resultado seria indefinido e, provavelmente, não seria bonito. Considerando o layout acima, `delete` leria a memória e interpretaria o que ele leu como o tamanho de um vetor, então começaria invocar os muitos destrutores, inconscientes do fato de que a memória em que ele está trabalhando não apenas não está no vetor, como também, provavelmente, não está mantendo os objetos do tipo que ele está ocupado destruindo.

O que aconteceria se você usasse a forma "`[]`" em `stringPtr2` (outro ponteiro para uma `string`)? Bem, isso é também indefinido, mas você pode ver que isso levaria a bem menos destrutores sendo chamados. Além disso, isso é indefinido também para tipos predefinidos, como `int`s, mesmo que esses tipos não possuam destrutores.

A regra é simples: se você usar `delete` em uma expressão `new`, deve usar `[]` na expressão `delete` correspondente. Se não usar `delete` em uma expressão `new`, não deve usar `[]` na expressão `delete` correspondente.

Essa é uma regra particularmente importante para lembrar quando estivermos escrevendo uma classe que contém um ponteiro para a memória alocada dinamicamente e também oferecendo múltiplos construtores, porque você deve cuidar para usar *a mesma forma* de `new` em todos os construtores para inicializar o membro ponteiro. Se você não fizer isso, como saberá que forma de `delete` usar em seu destrutor?

Considerar essa regra para definições de tipo por `typedef`, porque significa que um autor de um `typedef` deve documentar que forma de `delete` deve

ser empregada quando `new` for usado para criar objetos do tipo. Por exemplo, considere o `typedef` a seguir:

```
typedef std::string AddressLines[4];          // o endereço de uma pessoa possui 4 linhas,
                                              // cada uma delas é uma string
```

Como `AddressLines` (linhas de endereço) é um vetor, o uso de `new`

```
std::string *pal = new AddressLines;          // observe que "new AddressLines"
                                              // retorna uma string*, como
                                              // new string[4] faria
```

deve ser combinado com a forma de *vetor* de `delete`

```
delete pal;                                   // indefinido!

delete [ ] pal;                               // está bem
```

Para evitar essa confusão, abstenha-se da criação de `typedef`s para vetores. Isso é fácil, pois a biblioteca padrão de C++ (veja o Item 54) inclui `string` e `vector`, e esses templates reduzem a necessidade de matrizes alocadas de forma dinâmica praticamente a zero. Aqui, por exemplo, `AddressLines` poderia ser definida como um vetor de cadeias, ou seja, o tipo `vector<string>`.

Lembrete

» Se você usar `[]` em uma expressão `new`, deve usar `[]` na expressão `delete` correspondente. Se não usar `[]` em uma expressão `new`, não deve usar `[]` na expressão `delete` correspondente.

Item 17: Armazene objetos criados com `new` em ponteiros espertos em sentenças autocontidas

Suponhamos que temos uma função para revelar nossa prioridade de processamento e uma segunda função para realizar algum processamento em um `Widget` alocado dinamicamente de acordo com uma prioridade:

```
int priority( );

void processWidget(std::tr1::shared_ptr<Widget> pw, int priority);
```

Ciente da sabedoria de usar objetos para gerenciar recursos (veja o Item 13), `processWidget` (processar o `Widget`) usa um ponteiro esperto (aqui, um `tr1::shared_ptr`) para o `Widget` dinamicamente alocado que ele processa.

Considere, agora, uma chamada a `processWidget`:

```
processWidget(new Widget, priority( ));
```

Espere, não considere essa chamada. Ela não é compilada. O construtor de `tr1::shared_ptr` que recebe um ponteiro bruto é explícito, então não

existe uma conversão implícita do ponteiro bruto retornado pela expressão "`new Widget`" para o `tr1::shared_ptr` requerido por `processWidget`. O código a seguir, entretanto, será compilado:

```
processWidget(std::tr1::shared_ptr<Widget>(new Widget), priority( ));
```

Surpreendentemente, apesar de estarmos usando recursos gerenciados por objetos em todos os lugares aqui, essa chamada pode vazar recursos. É ilustrativo ver por quê.

Antes de os compiladores poderem gerar uma chamada a `processWidget`, eles precisam avaliar os argumentos sendo passados como parâmetros. O segundo argumento é uma chamada para a função `priority` (prioridade), mas o primeiro argumento, `("std::tr1::shared_ptr<Widget>(new Widget)")` consiste em duas partes:

- Execução da expressão "`new Widget`".
- Uma chamada para o construtor `tr1::shared_ptr`.

Antes de `processWidget` ser chamada, então, os compiladores devem gerar código para fazer estas três coisas:

- Chamar `priority`.
- Executar "`new Widget`".
- Chamar o construtor de `tr1::shared_ptr`.

Aos compiladores C++ é dada uma amplitude considerável para determinar a ordem na qual essas coisas são feitas (isso é diferente da maneira que linguagens como Java e C# funcionam, em que os parâmetros das funções são sempre avaliados em uma ordem em particular). A expressão "`new Widget`" deve ser executada antes de o construtor de `tr1::shared_ptr` ser chamado, porque o resultado da expressão é passado como argumento para o construtor de `tr1::shared_ptr`, mas a chamada a `priority` pode ser realizada em primeiro, em segundo ou em terceiro. Se os compiladores escolherem realizá-la em segundo lugar (algo que pode lhes permitir gerar código mais eficiente), terminamos com a seguinte sequência de operações:

1. Executar "`new Widget`".
2. Chamar `priority`.
3. Chamar o construtor de `tr1::shared_ptr`.

Mas considere o que aconteceria se a chamada a `priority` lançasse uma exceção. Nesse caso, o ponteiro retornado de "`new Widget`" seria perdido, porque ele não teria sido armazenado no `tr1::shared_ptr`, que esperávamos que nos protegesse de vazamentos de recursos. Pode surgir um vazamento na chamada a `processWidget` porque pode ocorrer uma exceção entre o

momento em que se cria um recurso (através de "`new Widget`") e o momento em que o recurso é enviado para um objeto de gerenciamento de recursos.

A maneira de evitar problemas como esse é simples: use uma sentença separada para criar o `Widget` e armazená-lo em um ponteiro esperto; então, passe o ponteiro esperto para `processWidget`:

```
std::tr1::shared_ptr<Widget> pw(new Widget);        // armazena o objeto criado por new
                                                    // em um ponteiro esperto em uma
                                                    // sentença autocontida
processWidget(pw, priority());                      // essa chamada não vazará
```

Isso funciona em diferentes compiladores porque é dada a eles menos liberdade em reordenar operações *entre* sentenças do que *dentro* de sentenças. Nesse código revisado, a expressão "`new Widget`" e a chamada ao construtor de `tr1::shared_ptr` estão em uma sentença diferente daquela que chama `priority`, então os compiladores não podem mover a chamada a `priority` para o meio delas.

Lembrete

» Armazene os objetos criados com `new` em ponteiros espertos em sentenças autocontidas. Não fazer isso pode levar a sutis vazamentos de recursos quando são lançadas exceções.

CAPÍTULO 4

PROJETOS E DECLARAÇÕES

Em geral, os projetos de software – abordagens para que os softwares façam aquilo que você quer que eles façam – começam como ideias bem gerais, mas, por fim, tornam-se detalhados o suficiente para o desenvolvimento de interfaces específicas. Essas interfaces devem, então, ser traduzidas em declarações C++. Neste capítulo, analisamos o problema de projetar e declarar boas interfaces em C++. Começamos com a recomendação que talvez seja a mais importante sobre o projeto de interfaces de qualquer tipo: devem ser fáceis de usar corretamente e difíceis de usar incorretamente. Isso prepara o terreno para diversas recomendações mais específicas que tratam de uma ampla faixa de tópicos, incluindo a correção, a eficiência, o encapsulamento, a capacidade de manutenção e extensão e a conformidade às convenções.

O material a seguir não é tudo o que você precisa saber sobre bons projetos de interface, mas ele destaca algumas das considerações mais importantes, avisa sobre alguns erros mais frequentes e fornece soluções para os problemas frequentemente encontrados por projetistas de classes, de funções e de templates.

Item 18: Deixe as interfaces fáceis de usar corretamente e difíceis de usar incorretamente

A linguagem C++ é coberta de interfaces. Interfaces de funções. Interfaces de classes. Interfaces de templates. Cada interface é um meio através do qual os clientes interagem com o código. Assumindo que você está lidando com uma quantidade razoável de pessoas, esses clientes estão tentando fazer um bom trabalho. Eles *querem* usar as interfaces corretamente. Sendo esse o caso, se usarem uma de suas interfaces incorretamente, ela é, ao menos, parcialmente culpada. Em teoria, se uma tentativa de uso de uma interface não faz aquilo que o cliente esperaria, o código não deveria ser compilado; e, se o código realmente for compilado, ele deveria fazer aquilo que o cliente quer.

Desenvolver interfaces que são fáceis de serem usadas corretamente e difíceis de serem usadas incorretamente requer que você pense nos tipos de

erros que os clientes podem fazer. Por exemplo, suponhamos que você esteja projetando o construtor para uma classe que representa datas no tempo:

```
class Date {
public:
   Date(int month, int day, int year);
   ...
};
```

À primeira vista, essa interface pode parecer razoável (pelo menos nos EUA), mas existem no mínimo dois erros que os clientes podem facilmente cometer. Primeiro, eles podem passar os parâmetros na ordem errada:

```
Date d(30, 3, 1995);                    // Oops! Deveria ser "3, 30" , e não "30, 3"
```

Segundo, eles podem tentar inserir um mês inválido ou um dia inválido:

```
Date d(3, 40, 1995);                    // Oops! Deveria ser “3, 30” , not “3, 40”
```

(Esse último exemplo pode parecer tolo, mas lembre-se de que, em um teclado, o 4 fica perto do 3. Esses erros de digitação do tipo “deslocamento por um” não são raros.)

Muitos erros dos clientes podem ser impedidos pela introdução de novos tipos. De fato, o sistema de tipos é seu principal aliado para a prevenção da compilação de código indesejado. Nesse caso, podemos introduzir tipos adaptadores (wrappers) simples para diferenciar dias, meses e anos e, então, usar esses tipos no construtor de `Date` (data):

```
struct Day {
   explicit Day(int d)
   : val(d) { }

   int val;
};

struct Month {
   explicit Month(int m)
   : val(m) { }

   int val;
};

struct Year {
   explicit Year(int y)
   : val(y){ }

   int val;
};

class Date {
public:
   Date(const Month& m, const Day& d, const Year& y);
   ...
};

Date d(30, 3, 1995);                    // erro! tipos errados

Date d(Day(30), Month(3), Year(1995)); // erro! tipos errados

Date d(Month(3), Day(30), Year(1995)); // tudo bem, os tipos estão corretos
```

Transformar `Day`, `Month` e `Year` (dia, mês e ano, respectivamente) em classes completas com dados encapsulados seria melhor do que o simples uso de estruturas acima (veja o Item 22), mas até as estruturas são suficientes para demonstrar que a introdução criteriosa de novos tipos pode fazer maravilhas para a prevenção de erros no uso de interfaces.

Uma vez que os tipos corretos estiverem prontos, pode ser razoável restringir os valores para esses tipos. Por exemplo, existem apenas 12 valores válidos para meses, então o tipo `Month` deve refletir isso. Uma maneira de fazer isso seria usar uma enumeração para representar o mês, mas

as enumerações não são tão seguras em relação aos tipos quanto gostaríamos. Por exemplo, as enumerações podem ser usadas como valores inteiros (veja o Item 2). Uma solução mais segura é predefinir o conjunto de todos os meses válidos:

```
class Month {
public:
   static Month Jan( ) { return Month(1); }          // funções retornando
   static Month Feb( ) { return Month(2); }          // todos os valores
   ...                                               // válidos de
   static Month Dec( ) { return Month(12); }         // Month

   ...                                               // outras funções membro

private:
   explicit Month(int m);                            // previne a criação de
                                                     // novos valores de mês
   ...                                               // dados específicos de um mês
};

Date d(Month::Mar( ), Day(30), Year(1995));
```

A ideia de usar funções em vez de objetos para representar meses específicos pode parecer estranha para você, talvez porque você tenha esquecido que a inicialização confiável de objetos estáticos não locais pode ser problemática. O Item 4 pode refrescar a sua memória.

Outra maneira de impedir erros prováveis dos clientes é restringir o que pode ser feito com um tipo. Uma maneira comum de impor restrições é adicionar `const`. Por exemplo, o Item 3 explica como o ato de qualificar o tipo de retorno de `operator*` como `const` pode impedir que os clientes cometam esse erro para tipos de dados definidos pelo usuário:

```
if (a * b = c) ...                                   // oops, queria fazer uma comparação
```

Na verdade, essa é apenas uma manifestação de outra recomendação geral para tornar os tipos fáceis de usar corretamente e difíceis de usar incorretamente: a menos que exista uma boa razão para não fazer isso, faça com que seus tipos se comportem de modo coerente com os tipos predefinidos. Os clientes já sabem de que forma os tipos como `int` se comportam; assim, você deve tentar fazer seus tipos se comportarem da mesma forma quando for razoável. Por exemplo, a atribuição a `a*b` não é legal se `a` e `b` forem inteiros; assim, a menos que exista uma boa razão para divergir desse comportamento, essa atribuição deve ser ilegal também para os seus tipos. Quando estiver em dúvida, faça como os `int`s.

A razão real para evitar incompatibilidades gratuitas com os tipos predefinidos é oferecer interfaces que tenham comportamento estável. Poucas características levam a interfaces que são fáceis de serem usadas corretamente no que diz respeito à consistência, e poucas características agravam as interfaces quanto às inconsistências. As interfaces para os contêineres STL são amplamente (apesar de não perfeitamente) consistentes, e isso ajuda a torná-las relativamente fáceis de serem usadas. Por exemplo, cada contêiner STL possui uma função membro chamada `size` (tamanho) que

lhe diz quantos objetos estão no contêiner. Compare isso com Java, em que você usa a *propriedade* `length` (comprimento) para vetores, o *método* `length` para cadeias (`String`) e o *método* `size` (tamanho) para listas (`List`); e com .NET, em que `Array`s (vetores) possuem uma propriedade chamada `Length`, enquanto `ArrayList`s (listas implementadas como vetores) possuem uma propriedade chamada `Count` (contador). Alguns desenvolvedores pensam que os ambientes integrados de desenvolvimento (IDEs – integrated development environments) fazem essas inconsistências serem irrelevantes, mas eles estão enganados. As inconsistências impõem um esforço mental no trabalho de um desenvolvedor que nenhuma IDE pode eliminar completamente.

Qualquer interface que exija que os clientes se lembrem de fazer algo é passível de ser usada incorretamente, porque os clientes podem se esquecer de fazê-lo. Por exemplo, o Item 13 inclui uma função fábrica que retorna ponteiros para objetos alocados dinamicamente em uma hierarquia de investimentos (cuja raiz é `Investment`):

```
Investment* createInvestment( );                    // do Item 13; parâmetros omitidos
                                                    // por questões de simplicidade
```

Para evitar vazamentos de recursos, os ponteiros retornados de `createInvestment` devem ser liberados no final de seu uso, mas isso cria uma oportunidade para, pelo menos, dois tipos de erros dos clientes: não conseguir liberar um ponteiro e liberar o mesmo ponteiro mais de uma vez.

O Item 13 mostra como os clientes podem armazenar o valor de retorno de `createInvestment` em um ponteiro esperto como `auto_ptr` ou `tr1::shared_ptr`, repassando para o ponteiro esperto a responsabilidade de usar `delete`. Mas, e se os clientes se esquecerem de usar o ponteiro esperto? Em muitos casos, uma decisão de interface melhor pode prever o problema fazendo com que a função fábrica retorne, desde o início, um ponteiro esperto:

```
std::tr1::shared_ptr<Investment> createInvestment( );
```

Essencialmente, isso força os clientes a armazenar o valor de retorno em um `tr1::shared_ptr`, eliminando a possibilidade de esquecimento da liberação do objeto `Investment` subjacente quando ele não estiver mais sendo usado.

Na verdade, retornar um `tr1::shared_ptr` possibilita que um projetista de interface impeça diversos outros erros relacionados à liberação de recursos, porque, como explica o Item 14, `tr1::shared_ptr` permite que uma função de liberação de recursos – um "apagador" – seja vinculado ao ponteiro esperto quando esse ponteiro é criado (`auto_ptr` não possui essa capacidade).

Suponhamos que os clientes que obtenham um ponteiro `Investment*` de `createInvestment` tenham que passar esse ponteiro para uma função chamada `getRidOfInvestment` (livrar-se do investimento), em vez de usar `delete` sobre ele. Essa interface abriria a porta para um novo tipo de erro

do cliente, em que os clientes usariam o mecanismo de destruição de recursos errado (usando `delete` em vez de `getRidOfInvestment`, por exemplo). O implementador de `createInvestment` pode evitar esses problemas retornando um `tr1::shared_ptr` com `getRidOfInvestment` vinculado a ele como seu apagador.

`tr1::shared_ptr` oferece um construtor que recebe dois argumentos: o ponteiro a ser gerenciado e o apagador a ser chamado quando a contagem de referências cair a zero. Isso sugere que a maneira de criar um `tr1::shared_ptr` nulo com `getRidOfInvestment` como seu apagador seria a seguinte:

```
std::tr1::shared_ptr<Investment>                // tenta criar um shared_ptr nulo
   pInv(0, getRidOfInvestment);                 // com um apagador específico;
                                                // isso não será compilado
```

Infelizmente, isso não é C++ válido. O construtor de `tr1::shared_ptr` insiste que seu primeiro parâmetro seja um ponteiro, e 0 não é um ponteiro, é um inteiro. Sim, ele pode ser convertido em um ponteiro, mas isso não é bom o suficiente nesse caso; `tr1::shared_ptr` insiste em um ponteiro real. Uma conversão explícita resolve o problema:

```
std::tr1::shared_ptr<Investment>                // cria um shared_ptr nulo com
   pInv( static_cast<Investment*>(0),           // getRidOfInvestment como seu apagador;
         getRidOfInvestment);                   // veja o Item 27 para obter mais
                                                // informações sobre static_cast
```

Isso significa que o código para implementar `createInvestment` para que retorne um `tr1::shared_ptr` com `getRidOfInvestment` como apagador se pareceria com o seguinte:

```
std::tr1::shared_ptr<Investment> createInvestment( )
{
   std::tr1::shared_ptr<Investment> retVal(static_cast<Investment*>(0),
                                           getRidOfInvestment);

   ...                                          // faça com que retVal
                                                // aponte para o objeto correto

   return retVal;
}
```

Obviamente, se o ponteiro bruto a ser gerenciado por `retVal` puder ser determinado antes de criar `retVal`, seria melhor passar o ponteiro bruto para o construtor de `retVal` em vez de inicializar `retVal` como nulo e então lhe fazer uma atribuição. Para obter detalhes sobre o motivo disso, consulte o Item 26.

Um recurso especialmente interessante de `tr1::shared_ptr` é que ele automaticamente usa seu apagador por ponteiro para eliminar outro erro em potencial dos clientes, o "problema entre DLLs". Esse problema surge quando se cria um objeto com `new` em uma biblioteca vinculada dinamicamente (DLL), mas é apagado em uma DLL diferente. Em muitas plataformas, esses pares `new/delete` em diferentes DLLs levam a erros em tempo de execução. O uso de `tr1::shared_ptr` evita o problema, porque seu apagador padrão usa `delete` da mesma DLL em que `tr1::shared_ptr` foi criado.

Isso significa, por exemplo, que, se `Stock` (ação) é uma classe derivada de `Investment` e `createInvestment` é implementada como segue

```
std::tr1::shared_ptr<Investment> createInvestment( )
{
   return std::tr1::shared_ptr<Investment>(new Stock);
}
```

então o `tr1::shared_ptr` retornado pode ser passado entre DLLs sem se preocupar com o problema entre DLSs. O `tr1::shared_ptr` que aponta para `Stock` rastreia o `delete` do qual DLL deve ser usado quando a contagem de referências para `Stock` se torna zero.

Este item não é sobre `tr1::shared_ptr` – é sobre tornar as interfaces fáceis de serem usadas corretamente e difíceis de serem usadas incorretamente – mas `tr1::shared_ptr` é uma maneira fácil de eliminar alguns erros dos clientes, e vale a pena ter uma visão geral do custo de sua utilização. A implementação mais comum de `tr1::shared_ptr` vem de Boost (veja o Item 55). O `tr1::shared_ptr` de Boost é duas vezes o tamanho de um ponteiro bruto, usa memória alocada dinamicamente para contabilidade e para dados específicos do apagador, usa uma chamada de função virtual quando invoca seu apagador e incorre em uma sobrecarga de sincronização de linhas de execução quando modifica a contagem de referências em uma aplicação que ele acredita ter múltiplas linhas de execução. (Você pode desabilitar o suporte a múltiplas linhas de execução ao definir um símbolo de pré-processador.) Em resumo, é maior do que um ponteiro bruto, mais lento do que um ponteiro bruto, e usa memória dinâmica auxiliar. Em muitas aplicações, esses custos adicionais em tempo de execução podem não ser percebidos, mas a redução em erros nos clientes será visível para todos.

Lembretes

- As boas interfaces são fáceis de serem usadas corretamente e difíceis de serem usadas incorretamente. Você deve buscar essas características em todas as suas interfaces.
- Entre as maneiras de facilitar o uso correto estão a consistência nas interfaces e a compatibilidade comportamental com os tipos predefinidos.
- Entre as formas de impedir erros estão a criação de novos tipos, a restrição de operações em tipos, a restrição de valores de objetos e a eliminação de responsabilidades de gerenciamento de recursos por parte dos clientes.
- `tr1::shared_ptr` suporta apagadores personalizados. Isso impede o problema entre DLLs, e pode ser usado para automaticamente destrancar objetos de exclusão mútua, etc.

Item 19: Trate o projeto de classe como projeto de tipo

Em C++, como em outras linguagens de programação orientada a objetos, a definição de uma nova classe define um novo tipo. Muito de seu tempo como desenvolvedor C++ será gasto melhorando o sistema de tipos. Isso significa que você não é apenas um projetista de classes, é um projetista de *tipos*. Sobrecarregar funções e operadores, controlar alocação e liberação de memória, definir a inicialização e a finalização de objetos – está tudo em suas mãos. Logo, você deve abordar o projeto de classes com o mesmo cuidado que os projetistas de linguagens dão aos tipos predefinidos nas linguagens.

Projetar classes boas é um desafio porque projetar tipos bons é um desafio. Bons tipos possuem uma sintaxe natural, uma semântica intuitiva e uma ou mais implementações eficientes. Em C++, uma definição de classe planejada de forma deficiente pode fazer com que fique impossível alcançar qualquer um desses objetivos. Mesmo as características de desempenho das funções membro de uma classe podem ser afetadas devido à forma como as funções são declaradas.

Como, então, você projeta classes eficazes? Primeiro, você deve entender as questões que está enfrentando. Praticamente todas as classes exigem que você se confronte com as questões a seguir, em que suas respostas muitas vezes levam a restrições em seu projeto:

- **Como os objetos de seu novo tipo devem ser criados e destruídos?** A forma como isso é feito influencia o projeto dos construtores e destrutores de sua classe, bem como suas funções de alocação e de liberação (`operator new`, `operator new[]`, `operator delete` e `operator delete[]` – veja o Capítulo 8), se você as escrever.

- **Como a inicialização de objetos difere da atribuição de objetos?** A resposta para essa questão determina o comportamento e as diferenças entre seus construtores e os operadores de atribuição. É importante não confundir a inicialização com a atribuição, porque essas operações correspondem a chamadas de diferentes funções (veja o Item 4).

- **O que significa os objetos de seu novo tipo serem passados por valor?** Lembre-se, o construtor de cópia define como a passagem por valor é implementada para um tipo.

- **Quais são as restrições nos valores legais para o seu novo tipo?** Normalmente, apenas algumas combinações de valores para os membros de dados de uma classe são válidas. Essas combinações determinam as invariantes que suas classes precisarão manter. As invariantes determinam a verificação de erros que você precisará fazer dentro de suas funções membro, especialmente seus construtores, operadores de atribuição e funções de escrita (setters). Elas também podem afetar as exceções que suas funções lançam e, caso você as utilize, suas especificações de exceções das funções.

- **Seu novo tipo se encaixa em um grafo de herança?** Se você herda de classes existentes, estará restrito ao projeto dessas classes, especialmente pelo fato de as funções serem virtuais ou não virtuais (veja os Itens 34 e 36). Se quiser permitir que outras classes herdem de sua classe, isso determinará se as funções que você declarar serão virtuais, especialmente seu destrutor (veja o Item 7).

- **Que conversões de tipo são permitidas para seu novo tipo?** Seu tipo existe em um mar de outros tipos, então será que deveria existir conversões entre seu tipo e outros tipos? Se você quiser permitir que objetos do tipo `T1` sejam *implicitamente* convertidos em objetos do tipo `T2`, vai querer escrever uma função de conversão de tipos na classe `T1` (como `operator T2`) ou um construtor não explícito na classe `T2` que possa ser chamado com um só argumento. Se quiser permitir apenas conversões *explícitas*, vai querer escrever funções para realizar as conversões, mas você precisa fazer com que elas não sejam operadores de conversão de tipos ou construtores não explícitos que possam ser chamados com um argumento. (Para ver um exemplo de funções de conversão implícitas e explícitas, veja o Item 15.)

- **Quais operações e funções fazem sentido para o novo tipo?** A resposta para essa questão determina quais funções você vai declarar para a sua classe. Algumas funções serão funções membro, mas outras não (veja os Itens 23, 24 e 46).

- **Que funções padrão devem ser desabilitadas?** Aquelas que você precisará declarar como privadas (veja o Item 6).

- **Quem deve ter acesso aos membros de seu novo tipo?** Essa questão ajuda a determinar quais membros são públicos, quais são protegidos e quais são privados. Ela também ajuda a determinar quais classes e/ou funções devem ser amigas, bem como se faz sentido aninhar uma classe dentro da outra.

- **Qual é a "interface não declarada" de seu novo tipo?** Que tipo de garantias de desempenho ela oferece à segurança das exceções (veja o Item 29) e ao uso de recursos (por exemplo, cadeados e memória dinâmica)? As garantias que você oferece nessas áreas vão impor restrições na implementação de sua classe.

- **Seu novo tipo é geral?** Até que ponto? Talvez você não esteja realmente definindo um novo tipo; talvez esteja definindo uma *família* completa de tipos. Se for esse o caso, você não quer definir uma nova classe, quer definir um template de classe.

- **Um novo tipo é realmente o que você precisa?** Se você está definindo uma nova classe derivada apenas para que possa adicionar funcionalidades a uma classe existente, talvez possa atingir seus objetivos de

uma maneira melhor simplesmente definindo uma ou mais funções não membro ou templates.

Essas questões são difíceis de serem respondidas, então definir classes eficazes pode ser um desafio. Feitas corretamente, entretanto, as classes definidas pelo usuário em C++ levam a tipos que são no mínimo tão bons quanto os tipos predefinidos, e isso faz todo o esforço valer a pena.

Lembrete

» O projeto de classe é o projeto de tipo. Antes de definir um novo tipo, certifique-se de considerar todas as questões discutidas neste item.

Item 20: Prefira a passagem por referência para constante em vez da passagem por valor

Por padrão, C++ passa objetos para e a partir de funções por valor (uma característica herdada de C). A menos que você especifique de outra forma, os parâmetros das funções são inicializados com *cópias* dos argumentos reais, e os chamadores das funções recebem de volta uma *cópia* do valor retornado pela função. Essas cópias são produzidas pelos construtores de cópia dos objetos. Isso pode encarecer a passagem por valor. Por exemplo, considere a seguinte hierarquia de classes:

```
class Person {
public:
   Person( );                          // parâmetros omitidos por questões de simplicidade
   virtual ~Person( );                 // veja o Item 7 para saber por que ele é virtual
   ...

private:
   std::string name;
   std::string address;
};

class Student: public Person {
public:
   Student( );                         // parâmetros omitidos novamente
   virtual ~Student( );
   ...

private:
   std::string schoolName;
   std::string schoolAddress;
};
```

Agora, considere o seguinte código, no qual chamamos uma função `validateStudent` (valida um estudante), que recebe um `Student` (estudante) como parâmetro (por valor) e retorna se ele foi validado:

```
bool validateStudent(Student s);                    // função que recebe um
                                                    // estudante por valor
Student plato;                                      // Platão estudou com Sócrates
bool platoIsOK = validateStudent(plato);            // chama a função
```

O que acontece quando essa função é chamada?

Claramente, o construtor de cópia de `Student` é chamado para inicializar o parâmetro `s` a partir de `plato`. Igualmente claro é o fato de que `s` é destruído quando `validadeStudent` retorna. Então, o custo da passagem de parâmetros dessa função é uma chamada ao construtor de cópia de `Student` e uma chamada ao destrutor de `Student`.

Mas essa não é a história completa. Um objeto da classe `Student` possui dois objetos `string` dentro dele, então, cada vez que você constrói um objeto `Student`, também deve construir dois objetos `string`. Um objeto `Student` também herda de um objeto `Person` (pessoa); assim, cada vez que você construir um objeto `Student`, também deve construir um objeto `Person`. Um objeto `Person` possui mais dois objetos `string` dentro dele; por isso, cada construção de `Person` também cria mais duas `string`s. O resultado final é que passar um objeto `Student` por valor leva a uma chamada ao construtor de cópia de `Student`, uma chamada para o construtor de cópia de `Person` e quatro chamadas ao construtor de cópia de `string`. Quando a cópia do objeto `Student` é destruída, cada chamada ao construtor é combinada com uma chamada ao destrutor, então o custo geral de passar um `Student` por valor é de seis construtores e seis destrutores!

Ora, esse comportamento está correto e é o desejado, até porque você *quer* que todos os objetos sejam inicializados e destruídos de maneira confiável. Mesmo assim, seria legal se existisse uma maneira de evitar todas essas construções e destruições. Existe uma: a passagem por referência a constante (`const`):

```
bool validateStudent(const Student& s);
```

Isso é muito mais eficiente: nenhum construtor ou destrutor é chamado, porque não se está criando objeto novo. O `const` na declaração de parâmetros revisada é importante. A versão original de `validateStudent` recebe um parâmetro `Student` por valor, então os chamadores sabem que eles estarão protegidos de quaisquer mudanças que a função possa fazer ao `Student` que eles passaram como parâmetro: `validateStudent` será capaz de modificar apenas uma *cópia* dele. Agora que o `Student` está sendo passado por referência, é necessário também declará-lo como constante (`const`), porque, caso contrário, os chamadores teriam que se preocupar com `validateStudent` fazendo ou não mudanças ao `Student` que eles passaram como parâmetro.

Passar parâmetros por referência também evita o *problema do fatiamento*. Quando um objeto da classe derivada é passado (por valor) como objeto da

classe-base, é chamado o construtor de cópia da classe-base, e os recursos especializados que fazem com que o objeto se comporte como objeto da classe derivada são "fatiados" e descartados. Você acaba ficando com um objeto simples da classe-base – quase nenhuma surpresa aqui, já que um construtor da classe-base o criou. Em geral, não é isso o que você quer. Por exemplo, suponhamos que você esteja trabalhando em um conjunto de classes para implementar um sistema de janelas gráficas:

```
class Window {
public:
  ...
  std::string name( ) const;                    // retorna o nome da janela
  virtual void display( ) const;                // desenha a janela e seu conteúdo
};

class WindowWithScrollBars: public Window {
public:
  ...
  virtual void display( ) const;
};
```

Todos os objetos `Window` (janela) têm um nome, o qual você pode obter por meio da função `name` (nome), e todas as janelas podem ser mostradas, o que você pode fazer invocando a função `display` (mostrar). O fato de `display` ser virtual lhe diz que a maneira pela qual os objetos da classe-base simples `Window` são mostrados é diferente da maneira pela qual são mostrados os objetos da classe mais sofisticada `WindowWithScrollBars` (janela com barras de rolagem) (veja os Itens 34 e 36).

Agora, suponhamos que você quisesse escrever uma função para mostrar o nome de uma janela e depois mostrar essa janela. Aqui está a maneira *errada* de escrever essa função:

```
void printNameAndDisplay(Window w)              // incorreto! o parâmetro
{                                               // pode ser fatiado.
  std::cout << w.name( );
  w.display( );
}
```

Considere o que acontece quando você chama essa função com um objeto `WindowWithScrollBars`:

```
WindowWithScrollBars wwsb;

printNameAndDisplay(wwsb);
```

O parâmetro `w` será construído – ele é passado por valor, lembra-se? – como objeto `Window`, e todas as informações especializadas que fazem com que `wwsb` aja como objeto `WindowWithScrollBars` seriam fatiadas e descartadas. Dentro de `printNameAndDisplay` (imprimir nome e mostrar), `w` sempre agirá como objeto da classe `Window` (porque ele *é* um objeto da classe `Window`), independentemente do tipo do objeto passado para a função. Em particular, a chamada a `display` dentro de `printNameAndDisplay` ***sempre*** chamará `Window::display` e nunca `WindowWithScrollBars::display`.

A maneira de evitar o problema do fatiamento é passar `w` por referência para uma constante:

```
void printNameAndDisplay(const Window& w)                    // bom, o parâmetro não
{                                                            // será fatiado
   std::cout << w.name( );
   w.display( );
}
```

Agora, `w` agirá como qualquer tipo de janela passada como parâmetro deve agir.

Se você estudar os bastidores de um compilador C++, descobrirá que as referências em geral são implementadas como ponteiros, então passar algo por referência normalmente significa passar um ponteiro. Como resultado, se você tem um objeto de um tipo predefinido (por exemplo, um `int`), costuma ser mais eficiente passá-lo por valor do que por referência. Para tipos predefinidos, então, quando você tem que escolher entre passar a constante por valor ou por referência, não é ruim escolher a passagem por valor. Essa mesma recomendação se aplica a iteradores e objetos função na STL, porque, por convenção, foram projetados para serem passados por valor. Os implementadores de iteradores e de objetos função são responsáveis por garantir que sejam eficientes em relação a cópias e que não estejam sujeitos ao problema de fatiamento. (Esse é um exemplo de como as regras mudam, dependendo da parte de C++ que você está usando – veja o Item 1.)

Os tipos predefinidos são pequenos, então algumas pessoas concluem que todos os tipos pequenos são bons candidatos para a passagem por valor, mesmo que sejam definidos pelo usuário. Esse é um raciocínio questionável. Somente porque um objeto é pequeno não significa que chamar seu construtor de cópia seja algo barato. Muitos objetos – a maioria dos contêineres STL dentre eles – contêm pouco mais do que um ponteiro, mas copiar esses objetos envolve copiar tudo para o qual eles apontam. Isso pode ser *muito* caro.

Mesmo quando pequenos objetos possuem construtores de cópia que não são caros, podem existir questões de desempenho envolvidas. Alguns compiladores tratam os tipos predefinidos e os tipos definidos pelo usuário de maneira diferente, mesmo que tenham a mesma representação subjacente. Por exemplo, alguns compiladores se recusam a colocar objetos que consistem apenas em um `double` em um registrador, mesmo que ponham, com satisfação, `double`s primitivos em registradores de forma regular. Quando esse tipo de coisa acontece, você fará melhor se passar esses objetos por referência, porque os compiladores certamente colocarão ponteiros (a implementação de referências) em registradores.

Outra razão pela qual os tipos pequenos definidos pelo usuário não são necessariamente bons candidatos para a passagem por valor é que, sendo definidos pelo usuário, seu tamanho está sujeito a mudanças. Um

tipo que, no presente, é pequeno pode ser grande no futuro, porque sua implementação interna pode mudar. As coisas podem mudar até mesmo quando você trocar para uma implementação diferente de C++. No momento em que redigimos este livro, por exemplo, algumas implementações do tipo `string` da biblioteca padrão eram *sete vezes* maiores do que outras.

Em geral, os únicos tipos para os quais você pode, razoavelmente, assumir que a passagem por valor é barata são os tipos predefinidos, os iteradores da STL e os tipos de objeto função. Para todos os outros, siga a recomendação deste item e prefira a passagem por referência para constante à passagem por valor.

Lembretes

» Prefira a passagem por referência para constante à passagem por valor. Em geral, ela é mais eficiente e evita o problema de fatiamento.

» Esta regra não se aplica aos tipos predefinidos, aos iteradores da STL e aos tipos de objeto função. Para eles, a passagem por valor é normalmente apropriada.

Item 21: Não tente retornar uma referência quando você deve retornar um objeto

Uma vez que os programadores entendem as implicações, em termos de desempenho, da passagem por valor para objetos (veja o Item 20), muitos se tornam verdadeiros cruzados, determinados a eliminar a diabólica passagem por valor onde quer que ela se esconda. Incansáveis em sua busca pela pureza da passagem por referência, invariavelmente cometem um erro fatal: começam a passar referências para objetos que não existem. Isso não é bom.

Considere uma classe para representar números racionais, incluindo uma função para multiplicar dois números racionais entre si:

```
class Rational {
public:
   Rational(int numerator = 0,                  // veja o Item 24 para saber por que
            int denominator = 1);               // o construtor não é declarado explicitamente

...
private:
   int n, d;                                    // numerador e denominador

friend
   const Rational                               // veja o Item 3 para saber por que
      operator*(const Rational& lhs,            // o tipo de retorno é constante
                const Rational& rhs);
};
```

Essa versão de `operator*` está retornando seu objeto por valor, e você estaria fugindo de seus deveres se não se preocupasse com o custo dessa

construção e destruição de objeto. Você não quer pagar por esse objeto se não é necessário. Então, a questão é a seguinte: você precisa pagar?

Bem, você não precisa se puder retornar uma referência em vez de um objeto. Mas lembre-se de que uma referência é apenas um *nome*, um nome para algum objeto *existente*. Sempre que você vir a declaração para uma referência, deve se perguntar imediatamente qual seu outro nome, porque deve ser outro nome para *algo*. No caso de `operator*`, se a função retornar uma referência, ela deve retornar uma referência para algum objeto racional (`Rational`) que já existe e que contém o produto dos dois objetos que vão ser multiplicados entre si.

Certamente não há razão para esperar que esse objeto exista antes de uma chamada a `operator*`. Ou seja, se você tem

```
Rational a(1, 2);                    // a = 1/2
Rational b(3, 5);                    // b = 3/5

Rational c = a * b;                  // c deveria ser 3/10
```

parece irracional esperar que já exista um número racional com o valor três décimos. Na verdade não é, se `operator*` retornar uma referência para esse número, ele mesmo deverá criar tal objeto número.

Uma função pode criar um novo objeto só de duas maneiras: na pilha ou no monte. A criação na pilha é realizada pela definição de uma variável local. Usando essa estratégia, você pode tentar escrever `operator*` desta forma:

```
const Rational& operator*(const Rational& lhs,          // aviso! código ruim!
                          const Rational& rhs)
{
   Rational result(lhs.n * rhs.n, lhs.d * rhs.d);
   return result;
}
```

Você pode rejeitar essa abordagem de saída, porque seu objetivo era evitar uma chamada ao construtor, e `result` (resultado) precisaria ser construído tal como qualquer outro objeto. Um problema mais sério é que essa função retorna uma referência a `result`, mas `result` é um objeto local, e os objetos locais são destruídos quando a função termina sua execução. Essa versão de `operator*`, então, não retorna uma referência a um `Rational` – ela retorna uma referência a um *ex*-`Rational`: algo que anteriormente foi um `Rational`; a carcaça vazia, fedorenta e apodrecida de algo que costumava ser um `Rational`, mas que não é mais, porque ele foi destruído. Qualquer chamador que tentasse *olhar* o valor de retorno dessa função instantaneamente entraria no mundo do comportamento indefinido. O fato é que qualquer função que retorne uma referência a um objeto local é falha. (O mesmo é verdade para qualquer função que retorne um ponteiro para um objeto local.)

Vamos considerar, então, a possibilidade de construir um objeto no monte e retornar uma referência a ele. São criados objetos baseados no monte

com o uso de `new`, portanto, você pode escrever um `operator*` baseado no monte, como a seguir:

```
const Rational& operator*(const Rational& lhs,                    // aviso! mais código ruim!
                          const Rational& rhs)
{
  Rational *result = new Rational(lhs.n * rhs.n, lhs.d * rhs.d);
  return *result;
}
```

Bem, você *ainda* precisa pagar por uma chamada ao construtor, porque a memória alocada por `new` é inicializada chamando um construtor apropriado, mas agora você tem um problema diferente: quem aplicará `delete` ao objeto criado por você com o uso de `new`?

Mesmo que os chamadores sejam conscientes e bem intencionados, eles não podem fazer muito para impedir vazamentos em cenários de uso razoáveis, como o seguinte:

```
Rational w, x, y, z;

w = x * y * z;                    // mesmo que operator*(operator*(x, y), z)
```

Aqui há duas chamadas a `operator*` na mesma sentença; portanto, dois usos de `new` que precisam ser desfeitos com usos de `delete`. Mesmo assim, não existe uma maneira razoável pela qual os clientes de `operator*` possam fazer essas chamadas, porque não existe uma maneira razoável para eles obterem os ponteiros ocultos atrás das referências que estão sendo retornadas das chamadas a `operator*`. Esse é um vazamento de recursos garantido.

Mas talvez você perceba que ambas as abordagens (na pilha e no monte) são penosas por precisarem chamar um construtor para cada resultado retornado de `operator*`. Talvez você se lembre de que nosso objetivo inicial era evitar essas invocações de construtores. Talvez pense em uma maneira de evitar todas as chamadas a construtores, exceto uma. Talvez a seguinte implementação venha à sua mente, uma implementação baseada em `operator*` que retorna uma referência a um objeto `Rational` *estático*, definido *dentro* da função:

```
const Rational& operator*(const Rational& lhs,    // aviso! mais código ruim!
                          const Rational& rhs)
{
  static Rational result;                         // objeto estático para o qual
                                                  // uma referência será retornada

  result = ... ;                                  // multiplica o lado esquerdo pelo direito
                                                  // e coloca o produto dentro de result

  return result;
}
```

Tal como todos os projetos que empregam o uso de objetos estáticos, isso imediatamente levanta questões de segurança de linhas de execução, mas essa é sua fraqueza mais óbvia. Para ver sua falha mais profunda, considere o seguinte código cliente, perfeitamente factível:

```
bool operator==(const Rational& lhs,                    // um operator ==
                const Rational& rhs);                   // para racionais

Rational a, b, c, d;
...
if ((a * b) == (c * d)) {
   faz o que for apropriado quando os produtos forem iguais
} else {
   faz o que for apropriado quando eles não forem
}
```

Adivinhe? A expressão `((a*b)==(c*d))` *sempre* será avaliada como `true`, independentemente dos valores de `a`, `b`, `c` e `d`!

Essa revelação é mais facilmente entendida quando o código é reescrito em sua forma funcional equivalente:

```
if (operator==(operator*(a, b), operator*(c, d)))
```

Observe que, quando `operator==` for chamado, já existirão duas chamadas ativas para `operator*`, cada uma retornando uma referência ao objeto estático `Rational` dentro de `operator*`. Logo, `operator==` terá que comparar o valor do objeto estático `Rational` dentro de `operator*` com o valor do objeto estático `Rational` dentro de `operator*`. Seria, na verdade, surpreendente se, na comparação, eles não fossem iguais. Sempre.

Isso deve ser suficiente para convencê-lo de que retornar uma referência a partir de uma função como `operator*` é uma perda de tempo, mas alguém talvez deve estar pensando: "Bem, se *um* objeto estático não é suficiente, talvez um *vetor* estático resolva o problema..."

Eu não consigo honrar esse projeto com um código de exemplo, mas posso explicar brevemente por que a notação deve fazê-lo corar de vergonha. Primeiro, você deve escolher *n*, o tamanho do vetor. Se *n* for muito pequeno, você pode ficar sem lugares para armazenar os valores de retorno da função; nesse caso, você não terá ganhado nada com o projeto com um único objeto estático que descartamos logo acima. Mas, se *n* for muito grande, você piora o desempenho de seu programa, porque *cada* objeto no vetor será construído na primeira vez que a função for chamada. Isso lhe custará *n* construtores e *n* destrutores*, mesmo que a função em questão seja chamada uma única vez. Se "otimização" é o processo de melhorar o desempenho de um software, esse tipo de coisa deveria se chamar "desotimização". Por fim, pense em como você colocaria os valores de que precisa nos objetos

* Os destrutores serão chamados uma vez no término do programa.

do vetor e o que custaria para você fazer isso. A maneira mais direta de mover um valor entre objetos é por meio da atribuição, mas qual é o custo de uma atribuição? Para muitos tipos, é mais ou menos o mesmo que uma chamada a um destrutor (para destruir o valor antigo) mais uma chamada para um construtor (para copiar o novo valor). Mas seu objetivo é evitar os custos da construção e da destruição! Admita: essa abordagem não renderá frutos. (Não, usar um `vector` em vez de um vetor predefinido não vai melhorar as coisas aqui.)

A maneira correta de escrever uma função que deve retornar um novo objeto é fazer a função retornar um novo objeto. Para o `operator*` de `Rational`, significa o código a seguir ou algo essencialmente equivalente:

```
inline const Rational operator*(const Rational& lhs, const Rational& rhs)
{
  return Rational(lhs.n * rhs.n, lhs.d * rhs.d);
}
```

Claro, você pode incorrer no custo de construir e de destruir o valor de retorno de `operator*`, mas, a longo prazo, esse é um preço pequeno a ser pago pelo comportamento correto. Além disso, a conta que você tanto teme pode nunca chegar. Como todas as linguagens de programação, C++ permite que os implementadores de compiladores apliquem otimizações para melhorar o desempenho do código gerado sem modificar seu comportamento observável e acaba que, em alguns casos, o valor de retorno da construção e da destruição de `operator*` pode ser eliminado com segurança. Quando os compiladores tiram vantagem desse fato (e eles em geral o fazem), seu programa continua a se comportar da maneira que supostamente deveria, apenas mais rapidamente do que você espera.

Tudo se resume ao seguinte: quando estiver decidindo entre retornar uma referência ou retornar um objeto, seu trabalho é escolher a alternativa que oferece o comportamento correto. Deixe que os implementadores de compiladores briguem para descobrir como fazer essa escolha ser a menos cara possível.

Lembrete

» Nunca retorne um ponteiro ou uma referência a um objeto de pilha local, uma referência a um objeto alocado no monte, ou um ponteiro ou referência a um objeto estático local se existir uma chance de mais de um desses objetos ser necessário. (O Item 4 fornece um exemplo de um projeto no qual é razoável retornar uma referência a uma variável local estática, pelo menos em ambientes com uma única linha de execução.)

Item 22: Declare os membros de dados como privados

Muito bem, este é o plano: primeiro, veremos por que os membros de dados não devem ser públicos. Depois, veremos que todos os argumentos contra os membros de dados públicos se aplicam igualmente aos membros protegidos. Isso nos levará à conclusão de que os membros de dados devem ser privados e, nesse ponto, teremos terminado.

Então, membros de dados públicos. Por que não?

Vamos iniciar pela consistência sintática (veja também o Item 18). Se os membros de dados não são públicos, a única maneira de os clientes acessarem um objeto é através das funções membro. Se tudo na interface púbica for uma função, os clientes não terão de coçar a cabeça tentando se lembrar se precisam usar parênteses quando quiserem acessar um membro da classe. Eles apenas os colocarão, porque tudo são funções. Ao longo de uma vida, isso pode economizar um monte de coçadas de cabeças.

Mas talvez você não ache convincente o argumento da consistência. E o que você acha então do fato de que usar funções dá um controle muito mais preciso sobre a acessibilidade dos membros de dados? Se você tornar um membro de dados público, todo mundo terá acesso de leitura e gravação a ele, mas, se você usar funções para obter ou modificar seu valor, pode impedir todos os acessos, implementar o acesso somente leitura e o acesso de leitura e gravação. E você pode até mesmo implementar o acesso de gravação apenas se quiser:

```
class AccessLevels {
public:
  ...
  int getReadOnly( ) const              { return readOnly; }

  void setReadWrite(int value)          { readWrite = value; }
  int getReadWrite( ) const             { return readWrite; }

  void setWriteOnly(int value)          { writeOnly = value; }

private:
  int noAccess;                              // nenhum acesso para esse int

  int readOnly;                              // acesso somente leitura para esse int

  int readWrite;                             // acesso de leitura e gravação para esse int

  int writeOnly;                             // acesso somente gravação para esse int
};
```

Esse controle de acesso de granularidade fina é importante, porque muitos membros de dados *devem* ser ocultos. Raramente cada um dos membros de dados precisa de um método de leitura e de gravação (*get* e *set*, respectivamente).

Ainda não está convencido? Então, é hora de trazer nossa arma secreta: o encapsulamento. Se você implementar o acesso a um membro de da-

dos por meio de uma função, pode substituir o membro de dados posteriormente com uma computação, e ninguém que está usando sua classe ficará ciente disso.

Por exemplo, suponhamos que você esteja escrevendo um aplicativo em que um equipamento automatizado está monitorando a velocidade dos carros que estão passando. À medida que os carros passam, sua velocidade é computada e o valor é adicionado em uma coleção de todos os dados de velocidade coletados até agora.

```
class SpeedDataCollection {
  ...
public:
  void addValue(int speed);                    // adiciona um novo valor de dados

  double averageSoFar() const;                 // retorna a velocidade média

  ...
};
```

Agora, considere a implementação da função membro `averageSoFar` (média até agora). Uma maneira de implementá-la é ter um membro de dados na classe que é uma média corrente de todos os dados de velocidades coletados até o momento. Em qualquer momento que `averageSoFar` for chamada, ela só retorna o valor desse membro de dados. Uma abordagem diferente é fazer `averageSoFar` computar seu valor cada vez que for chamada, algo que você pode fazer examinando todos os valores de dados na coleção.

A primeira abordagem (manter uma média corrente) aumenta cada objeto `SpeedDataCollection` (coleção de dados de velocidade), porque você precisa alocar espaço para os membros de dados que mantêm a média corrente, o total acumulado e o número de pontos de dados. Entretanto, `averageSoFar` pode ser implementada de maneira muito eficiente: é apenas uma função internalizada (veja o Item 30) que retorna o valor da média corrente. Em contrapartida, computar a média sempre que for requisitada faz `averageSoFar` ser executada mais lentamente, mas cada objeto `SpeedDataCollection` será menor.

Quem poderá dizer qual é a melhor? Em uma máquina em que a memória é restrita (por exemplo, um dispositivo embarcado ao lado da via), e em aplicações nas quais as médias não se fazem necessárias com frequência, computar a média todas as vezes é, provavelmente, uma solução melhor. Em uma aplicação em que as médias costumam ser necessárias, a velocidade é a essência, e a memória não é um problema, manter uma média corrente geralmente será preferível. O ponto importante é que, ao acessar a média por uma função membro (encapsulando-a), você pode trocar entre essas implementações diferentes (bem como por qualquer outra que você imaginar), e os clientes, no pior dos casos, só terão que ser compilados novamente. (Você pode eliminar até mesmo essa inconveniência seguindo as técnicas descritas no Item 31.)

Ocultar os membros de dados atrás de interfaces funcionais pode oferecer todos os tipos de flexibilidade de implementação. Por exemplo, desse

modo fica mais fácil notificar outros objetos quando os membros de dados são lidos ou gravados, verificar invariantes de classe e pré e pós-condições de funções, realizar sincronização em ambientes com múltiplas linhas de execução, etc. Os programadores que chegam a C++ vindos de linguagens como Delphi e C# reconhecerão essas capacidades como o equivalente às "propriedades" nessas outras linguagens, embora seja preciso digitar um conjunto extra de parênteses.

A questão sobre o encapsulamento é mais importante do que pode parecer inicialmente. Se você ocultar os membros de dados de seus clientes (ou seja, encapsulá-los), pode garantir que as invariantes de classe sejam sempre mantidas, porque apenas as funções membro podem afetá-las. Além disso, você se reserva o direito de modificar suas decisões de implementação posteriormente. Se você não ocultar essas decisões, logo descobrirá que, mesmo que seja o proprietário do código-fonte de uma classe, sua habilidade de modificar algo público é extremamente restrita, porque muito código cliente será quebrado. Público significa não encapsulado e, na prática, significa imutável, em especial para classes que são amplamente usadas. Ainda assim, as classes amplamente usadas têm uma imensa necessidade de encapsulamento, pois são as que mais podem se beneficiar da habilidade de substituir uma implementação por outra melhor.

O argumento contra os membros de dados protegidos é similar. Na verdade, é idêntico, apesar de não parecer à primeira vista. O raciocínio sobre a consistência sintática e sobre o controle de acesso de granularidade fina é claramente tão aplicável aos dados protegidos quanto aos dados públicos. Mas e o encapsulamento? Os membros de dados protegidos não são mais encapsulados do que os públicos? Na prática, a resposta é, surpreendentemente, não.

O Item 23 explica que o encapsulamento de algo é inversamente proporcional à quantidade de código que pode ser quebrado se algo for modificado. Assim, o grau de encapsulamento de um membro de dados é inversamente proporcional à quantidade de código que pode ser quebrado se os membros de dados mudarem, por exemplo, se um membro de dados for removido da classe (possivelmente em favor de uma computação, como em `averageSoFar`, acima).

Suponhamos que tivéssemos um membro de dados público e o eliminássemos. Quanto do código seria quebrado? Todo o código cliente, que é geralmente uma quantidade *desconhecidamente grande*. Então, os membros de dados públicos são completamente não encapsulados. Mas suponhamos que tivéssemos um membro de dados protegido e o eliminássemos. Quanto código seria quebrado agora? Todas as classes derivadas que o usam, o que é, mais uma vez, uma quantidade *desconhecidamente grande*. Os membros de dados protegidos são tão não encapsulados quanto os públicos, porque, em ambos os casos, se os membros de dados são modificados, uma quantidade desconhecidamente grande de código cliente estará quebrada. Isso não faz sentido, mas, como dirão os implementadores experientes de bibliotecas, é verdade mesmo assim. Uma vez que você declarou um membro de dados

como público ou protegido e os clientes começaram a usá-lo, é muito difícil mudar qualquer coisa sobre esse membro de dados. Muito código precisa ser reescrito, retestado ou recompilado. Do ponto de vista do encapsulamento, existem apenas dois níveis de acesso: privado (que oferece encapulamento) e todo o resto (que não oferece).

Lembretes

» Declare os membros de dados como privados (`private`). Isso dá aos clientes um acesso sintaticamente uniforme para os dados, permite um controle de acesso de granularidade mais fina, permite que as invariantes sejam satisfeitas e oferece flexibilidade à implementação das classes por parte de seus autores.

» Os dados protegidos (`protected`) não são mais encapsulados do que os dados públicos (`public`).

Item 23: Prefira funções não membro e não amigas a funções membro

Imagine uma classe para representar os navegadores da Web. Dentre as muitas funções que essa classe pode oferecer estão aquelas para limpar o histórico dos elementos baixados, limpar o histórico de URLs visitadas e remover todos os cookies do sistema:

```
class WebBrowser {
public:
  ...
  void clearCache( );
  void clearHistory( );
  void removeCookies( );
  ...
};
```

Muitos usuários vão querer realizar todas essas ações de uma vez só, então `WebBrowser` (navegador Web) pode também oferecer uma função para fazer justamente isso:

```
class WebBrowser {
public:
  ...
  void clearEverything( );                    // chama clearCache, clearHistory
                                              // e removeCookies
  ...
};
```

É claro, essa funcionalidade também poderia ser fornecida por uma função não membro que chama as funções membro apropriadas:

```
void clearBrowser(WebBrowser& wb)
{
  wb.clearCache( );
  wb.clearHistory( );
```

```
    wb.removeCookies( );
}
```

Então, o que é melhor, a função membro `clearEverything` (limpa tudo) ou a função não membro `clearBrowser` (limpa o navegador)?

Os princípios de orientação a objetos determinam que os dados e as funções que operam sobre eles devem ser empacotados juntos, e isso sugere que a função membro é a melhor escolha. Infelizmente, essa sugestão é incorreta. Ela se baseia em um entendimento errôneo sobre o que significa "orientado a objetos". Os princípios de orientação a objetos estabelecem que os dados devem ser tão *encapsulados* quanto possível. De maneira ilógica, a função membro `clearEverything` na verdade fornece *menos* encapsulamento do que a função não membro `clearBrowser`. Além disso, oferecer a função não membro dá maior flexibilidade de empacotamento para funcionalidades relacionadas à classe `WebBrowser`, e isso, por sua vez, leva a menos dependências de compilação e a um aumento na capacidade de extensão de `WebBrowser`. A abordagem não membro é então melhor do que uma função membro de diversas maneiras. É importante entender por quê.

Iniciaremos com o encapsulamento. Se algo está encapsulado, está oculto da visão. Quanto mais estiver encapsulado, menos coisas podem vê-lo. Quanto menos coisas puderem vê-lo, maior flexibilidade temos para mudá-lo, porque nossas mudanças afetam diretamente apenas aquelas coisas que veem aquilo que estamos mudando. Quanto mais algo é encapsulado, maior nossa habilidade de modificá-lo. Essa é exatamente a razão pela qual damos tanto valor ao encapsulamento: ele nos dá a flexibilidade de modificar as coisas de uma maneira que afete apenas um número limitado de clientes.

Considere os dados associados com um objeto. Quanto menos código puder ver os dados (ou seja, acessá-los), mais os dados serão encapsulados, e mais livremente poderemos mudar as características dos dados de um objeto, como o número de membros de dados, seus tipos, etc. Como medida de granularidade grossa de quanto um código pode ver um dado, podemos contar o número de funções que podem acessar esse dado: quanto mais funções puderem acessá-lo, menos encapsulado estará o dado.

O Item 22 explica que os membros de dados devem ser privados, porque, se não forem, um número ilimitado de funções poderá acessá-los. Eles não possuem encapsulamento algum. Para os membros de dados que *são* privados, o número de funções que podem acessá-los é o número de funções membro da classe mais o número de funções amigas, uma vez que apenas membros e amigos possuem acesso aos dados privados. Dada uma escolha entre uma função membro (que pode acessar não apenas os dados privados de uma classe, mas também funções privadas, enumerações, definições de tipo [`typedefs`], etc) e uma função não mem-

bro e não amiga (a qual não pode acessar nada dessas coisas) fornecendo a mesma funcionalidade, a escolha que leva a um maior encapsulamento é a função não membro não amiga, porque ela não aumenta o número de funções que podem acessar as partes privadas da classe. Isso explica porque `clearBrowser` (a função não membro não amiga) é preferível a `clearEverything` (a função membro): ela leva a um encapsulamento maior na classe `WebBrowser`.

Nesse ponto, vale a pena perceber duas coisas. Primeiro, esse raciocínio se aplica apenas a funções não membro *não amigas*. Os amigos possuem o mesmo acesso aos membros privados de uma classe que as funções membro, logo, o impacto é o mesmo no encapsulamento. Do ponto de vista do encapsulamento, a escolha não é entre funções membro e funções não membro, e sim entre funções membro e funções não membro não amigas. (O encapsulamento não é o único ponto de vista, é claro. O Item 24 explica que, quando a questão envolve conversões de tipo implícitas, a escolha *é* entre funções membro e não membro.)

A segunda coisa a observar é que, só porque as preocupações sobre o encapsulamento estabelecem que uma função seja uma função não membro de uma classe, não significa que ela não possa ser membro de outra classe. Isso pode se mostrar um problema considerável para os programadores acostumados com linguagens nas quais todas as funções *devem* estar em classes (por exemplo, Eiffel, Java, C#, etc). Por exemplo, poderíamos fazer de `clearBrowser` uma função membro estática de uma classe utilitária. Desde que ela não seja parte de (ou amiga de) `WebBrowser`, ela não afeta o encapsulamento dos membros privados de `WebBrowser`.

Em C++, uma abordagem mais natural seria tornar `clearBrowser` uma função não membro no mesmo espaço de nomes de `WebBrowser`:

```
namespace WebBrowserStuff {
    class WebBrowser { ... };
    void clearBrowser(WebBrowser& wb);
    ...
}
```

Existem, entretanto, mais coisas acontecendo além da naturalidade, pois os espaços de nomes, diferentemente das classes, podem se espalhar por múltiplos arquivos-fonte. Isso é importante, já que funções como `clearBrowser` são *funções de conveniência*. Não sendo nem membros nem amigas, elas não possuem acesso especial a `WebBrowser`, então elas não podem oferecer funcionalidade alguma que um cliente de `WebBrowser` já não poderia obter de alguma outra forma. Por exemplo, se `clearBrowser` não existisse, os clientes poderiam simplesmente chamar `clearCache` (limpar cache), `clearHistory` (limpar histórico) e `removeCookies` (remover cookies).

Uma classe como `WebBrowser` pode ter um grande número de funções de conveniência, algumas relacionadas aos itens favoritos, outras relacionadas

à impressão e ainda outras relacionadas ao gerenciamento de cookies, etc. Como regra geral, a maioria dos clientes estará interessada em apenas alguns desses conjuntos de funções de conveniência. Não há motivo para que um cliente interessado apenas em funções de conveniência relacionadas aos itens favoritos dependa da compilação de funções de conveniência relacionadas aos cookies, por exemplo. A maneira mais simples e direta de separá-las é declarar as funções de conveniência relacionadas aos itens favoritos em um arquivo de cabeçalho, as funções de conveniência relacionadas aos cookies em outro, as funções de conveniência relacionadas à impressão em um terceiro, etc.:

```
// cabeçalho "webbrowser.h" – cabeçalho para a classe WebBrowser propriamente dita
// bem como funcionalidades relacionadas às funcionalidades principais de WebBrowser
namespace WebBrowserStuff {

   class WebBrowser { ... };
      ...                                  // funcionalidades relacionadas ao "principal"
                                           // da classe, por exemplo, funções não membro de
                                           // que praticamente todos os clientes precisam
}

// cabeçalho "webbrowserbookmarks.h"
namespace WebBrowserStuff {
   ...                                     // funções de conveniência relacionadas
}                                          // a itens favoritos

// cabeçalho "webbrowsercookies.h"
namespace WebBrowserStuff {
   ...                                     // funções de conveniência relacionadas
}                                          // a cookies
...
```

Observe que é exatamente assim que a biblioteca padrão de C++ é organizada. Em vez de ter um só cabeçalho monolítico `<C++StandardLibrary>` contendo tudo no espaço de nomes `std` (padrão), existem dezenas de cabeçalhos (como `<vector>`, `<algorithm>`, `<memory>`, etc), em que cada um declara parte da funcionalidade em `std`. Os clientes que usam apenas funcionalidades relacionadas a vetores não precisam incluir o cabeçalho `<memory>`; os clientes que não usam listas (`list`) não precisam incluir `<list>`. Isso permite que os clientes dependam da compilação apenas nas partes do sistema que realmente usam. (Consulte o Item 31 para ver uma discussão sobre outras maneiras de reduzir as dependências de compilação.) Dividir a funcionalidade dessa forma não é possível quando ela vem de funções de uma classe membro, porque uma classe deve ser inteiramente definida; ela não pode ser dividida em pedaços.

Colocar todas as funções de conveniência em múltiplos arquivos de cabeçalho – mas em um único espaço de nomes – também significa que os clientes podem facilmente *estender* o conjunto de funções de conveniência. Tudo o que eles têm a fazer é adicionar mais funções não membro não amigas para o espaço de nomes. Por exemplo, se um cliente de `WebBrowser` decide escrever funções de conveniência relacionadas ao download de imagens, ele só precisa criar um arquivo de

cabeçalho contendo as declarações dessas funções no espaço de nomes `WebBrowserStuff` (funcionalidades do navegador da Web). As novas funções estão agora disponíveis e integradas, bem como todas as outras funções de conveniência. Esse é outro recurso que as classes não podem oferecer, porque as definições de classe são fechadas para a estensão por parte dos clientes. Claro, os clientes podem gerar novas classes, mas as classes derivadas não têm acesso aos membros encapsulados (ou seja, privados) na classe-base; então, as "funcionalidades estendidas" têm um status de segunda classe. Além disso, como explica o Item 7, nem todas as classes são projetadas para ser classes-base.

Lembrete

» Prefira funções não membro não amigas a funções membro. Isso aumenta o encapsulamento, a flexibilidade de empacotamento e a capacidade de extensão funcional.

Item 24: Declare funções não membro quando as conversões de tipo tiverem de ser aplicadas a todos os parâmetros

Comentei, na introdução deste livro, que fazer as classes oferecerem conversões de tipo implícitas em geral é uma má ideia. É claro, existem exceções a essa regra, e uma das mais comuns é quando estamos criando tipos numéricos. Por exemplo, se você está projetando uma classe para representar números racionais, permitir conversões implícitas de inteiros para racionais não parece ruim. Certamente não é menos razoável que as conversões predefinidas de C++ de `int` para `double` (e é muito mais razoável do que as conversões predefinidas de C++ de `double` para `int`). Sendo esse o caso, você deve iniciar sua classe `Rational` da seguinte forma:

```
class Rational {
public:
   Rational(int numerator = 0,          // o construtor é deliberadamente não explícito,
            int denominator = 1);       // permite conversões implícitas de int
                                        // para Rational

   int numerator( ) const;              // métodos de acesso para numerator e
   int denominator( ) const;            // denominator – veja o Item 22

private:
   ...
};
```

Você sabe que vai querer oferecer suporte para operações aritméticas como adição, multiplicação, etc., mas não tem certeza se deve implementá-las como funções membro, funções não membro ou, possivelmente, funções membro amigas. Seus instintos dizem que, quando está em dúvida, você deve ser orientado a objetos. Você sabe disso, sabe que a multiplicação de números racionais está relacionada à classe `Rational`, então parece natural implementar `operator*` para números racionais

dentro da classe `Rational`. Sem seguir a lógica, o Item 23 argumenta que a ideia de colocar funções dentro da classe a que estão associadas é, algumas vezes, *contrária* aos princípios de orientação a objetos, mas vamos colocar isso de lado e investigar a ideia de tornar `operator*` uma função membro de `Rational`:

```
class Rational {
public:
  ...

  const Rational operator*(const Rational& rhs) const;
};
```

(Se você não está certo de porque essa função é declarada dessa forma – retornando um resultado constante por valor, mas recebendo uma referência para uma constante como seu argumento – consulte os Itens 3, 20 e 21.)

Esse projeto permite multiplicar os racionais com a maior facilidade:

```
Rational oneEighth(1, 8);
Rational oneHalf(1, 2);

Rational result = oneHalf * oneEighth;                    // muito bem

result = result * oneEighth;                              // muito bem
```

Mas você não está satisfeito. Você também gostaria de oferecer suporte para operações de modo misto, nas quais os números racionais (`Rational`) possam ser multiplicados, por exemplo, com números inteiros (`int`). Afinal, poucas coisas são tão naturais quanto multiplicar dois números entre si, mesmo que, por acaso, sejam tipos diferentes de números.

Quando você tenta fazer aritmética de modo misto, entretanto, descobre que isso funciona apenas metade das vezes:

```
result = oneHalf * 2;                                     // tudo bem

result = 2 * oneHalf;                                     // erro!
```

Esse é um mau presságio. A multiplicação supostamente deveria ser comutativa, lembra-se?

A fonte do problema torna-se visível quando você reescreve os dois últimos exemplos em sua forma funcional equivalente:

```
result = oneHalf.operator*(2);                            // muito bem

result = 2.operator*(oneHalf);                            // erro!
```

O objeto `oneHalf` (uma metade) é uma instância de uma classe que contém um `operator*`, então os compiladores chamam essa função. Entretanto, o inteiro 2 não possui uma classe associada, então não existe uma função membro `operator*`. Os compiladores também procurarão por `operator*`s não membro (por exemplo, aqueles no espaço de nomes ou no escopo global) que podem ser chamados como a seguir:

```
result = operator*(2, oneHalf);                           // erro!
```

Mas, nesse exemplo, não existe uma função não membro `operator*` que receba um `int` e um `Rational`, então há falha na busca.

Olhe mais uma vez para a chamada que é bem-sucedida. Você verá que o seu segundo parâmetro é o inteiro 2, embora `Rational::operator*` receba um objeto `Rational` como parâmetro. O que está acontecendo aqui? Por que o 2 funciona em uma posição e não em outra?

O que está acontecendo é a conversão de tipo implícita. Os compiladores sabem que você está passando um `int` e que a função requer um `Rational`, mas eles também sabem que podem criar um `Rational` adequado chamando o construtor de `Rational` com o `int` que você forneceu, então é isso o que eles fazem. Ou seja, eles tratam a chamada como se ela fosse escrita mais ou menos da seguinte forma:

```
const Rational temp(2);                 // cria um objeto Rational
                                        // objeto a partir de 2
result = oneHalf * temp;                // o mesmo que oneHalf.operator*(temp);
```

É claro que os compiladores fazem isso apenas porque um construtor não explícito está envolvido. Se o construtor de `Rational` fosse explícito (`explicit`), nenhuma dessas sentenças seria compilada:

```
result = oneHalf * 2;                   // erro! (com construtor explícito)
                                        // não pode converter 2 para Rational
result = 2 * oneHalf;                   // mesmo erro, mesmo problema
```

Essa abordagem não suporta aritmética de modo misto, mas, no mínimo, o comportamento das duas sentenças seria consistente.

Seu objetivo, entretanto, é tanto a consistência quanto o suporte para aritmética, ou seja, um projeto no qual ambas as sentenças acima seriam compiladas. Isso nos traz de volta a essas duas sentenças e ao motivo pelo qual, mesmo que o construtor de `Rational` não seja explícito, uma é compilada e a outra não:

```
result = oneHalf * 2;                   // muito bem (com construtor não explícito)
result = 2 * oneHalf;                   // erro! (mesmo com construtor não explícito)
```

Acontece que os parâmetros são elegíveis para a conversão implícita de tipos *apenas se forem listados na lista de parâmetros*. O parâmetro implícito correspondente ao objeto no qual a função membro é invocada – aquele para o qual `this` aponta – *nunca* é elegível para conversões implícitas. É por isso que a primeira chamada é compilada e a segunda não. O primeiro caso envolve um parâmetro listado na lista de parâmetros, mas o segundo caso não.

Você ainda gostaria oferecer suporte para a aritmética de modo misto, entretanto, e a maneira para fazer isso talvez esteja clara agora: torne `operator*` uma função não membro, permitindo, então, que os compiladores realizem conversões de tipo implícitas em *todos* os argumentos:

```
class Rational {
  ...                                                  // não contém operator*
};
const Rational operator*(const Rational& lhs,          // agora uma função não membro
                         const Rational& rhs)
{
  return Rational(lhs.numerator( ) * rhs.numerator( ),
                  lhs.denominator( ) * rhs.denominator( ));
}
Rational oneFourth(1, 4);
Rational result;
result = oneFourth * 2;                                // muito bem
result = 2 * oneFourth;                                // agora funciona!
```

Esse é, certamente, um final feliz para a história, mas existe uma preocupação irritante. A função `operator*` deveria ser amiga da classe `Rational`?

Neste caso, a resposta é não, porque `operator*` pode ser implementada completamente em termos da interface pública de `Rational`. O código acima mostra uma maneira de fazer isso. Assim, temos uma observação importante a fazer: o oposto de uma função membro é uma *função não membro*, e não uma função amiga. Muitos programadores de C++ partem do pressuposto de que, se uma função é relacionada a uma classe e não deve ser um membro (por exemplo, por causa de uma necessidade de conversão de tipos em todos os argumentos), ela deve ser uma função amiga. Esse exemplo demonstra que esse raciocínio é falho. Sempre que você puder, evite funções amigas. Você deve fazer isso porque, bem como na vida real, os amigos frequentemente trazem mais problemas do que valem. Algumas vezes, a amizade é garantida, é claro, mas permanece o fato de que, só porque uma função não deve ser um membro, não quer dizer que ela deva ser uma amiga.

Este item contém a verdade e nada mais que a verdade, mas não é a verdade completa. Quando você cruzar a linha de C++ Orientado a Objetos para C++ com Templates (veja o Item 1) e fizer `Rational` ser um template de classe em vez de uma classe, existirão novas questões a serem consideradas, novas maneiras de resolvê-las e algumas implicações de projeto surpreendentes. Essas questões, resoluções e implicações são o tópico do Item 46.

Lembrete

» Se você precisar de conversões de tipo em todos os parâmetros de uma função (incluindo aquele que, de outra forma, seria apontado pelo ponteiro `this`), a função deve ser não membro.

Item 25: Considere o suporte para um `swap` que não lance exceções

A função `swap` (trocar) é interessante. Originalmente introduzida como parte da STL, se tornou pilar da programação segura em relação às exceções (veja o Item 29) e um mecanismo comum para lidar com a possibilidade de autoatribuição (veja o Item 11). Como `swap` é tão útil, é importante implementá-la apropriadamente; mas, juntamente com sua importância única, vem um conjunto único de complicações. Neste item, exploramos quais são elas e como tratá-las.

Trocar os valores de dois objetos significa dar a cada um deles o valor do outro. Por padrão, a troca é feita por meio do algoritmo padrão `swap`. Sua implementação típica é justamente aquilo que você esperaria:

```
namespace std {

  template<typename T>                    // implementação típica de std::swap;
  void swap(T& a, T& b)                   // troca os valores entre a e b
  {
    T temp(a);
    a = b;
    b = temp;
  }

}
```

Como seus tipos oferecem suporte para cópias (através do construtor de cópias ou do operador de atribuição por cópia), a implementação padrão de `swap` deixará que os objetos dos tipos que você criar sejam trocados sem que você se preocupe em ter de fazer quaisquer trabalhos especiais para oferecer suporte a isso.

Entretanto, a implementação padrão de `swap` pode não empolgá-lo. Ela envolve copiar três objetos: `a` para `temp`, `b` para `a` e `temp` para `b`. Para alguns tipos, nenhuma dessas cópias é realmente necessária; para eles, o `swap` padrão é uma rota direta para a pista lenta.

Os mais proeminentes dentre esses tipos são aqueles que consistem principalmente em um ponteiro para outro tipo que contém os dados propriamente ditos. Uma manifestação comum dessa abordagem de projeto é o "dialeto, expressão, linguagem (idiomática) pimpl" ("ponteiro para implementação" – veja o Item 31). Uma classe `Widget` que emprega esse projeto se pareceria com o seguinte:

```
class WidgetImpl {                        // classe para dados de Widget
```

```
public:                                    // os detalhes não são importantes
   ...
private:
   int a, b, c;                            // possivelmente muitos dados
   std::vector<double> v;                  // caro para copiar!
   ...
};
class Widget {                                    // classe usando o idioma pimpl
public:
   Widget(const Widget& rhs);
   Widget& operator=(const Widget& rhs)           // para copiar um Widget, copie seu
   {                                              // objeto WidgetImpl. Para saber mais
      ...                                         // detalhes sobre como implementar
      *pImpl = *(rhs.pImpl);                      // operator= de um modo geral, veja
      ...                                         // os Itens 10, 11 e 12
   }
   ...
private:
   WidgetImpl *pImpl;                             // ponteiro para o objeto
};                                                // com os dados desse Widget
```

Para trocar o valor de dois objetos `Widget`, tudo o que precisamos fazer é trocar seus ponteiros `pimpl`, mas o algoritmo `swap` padrão não tem como saber isso, então ele copiaria não apenas três `Widgets`, mas também três objetos `WidgetImpl` (implementação de um `Widget`). Muito ineficiente. Nada empolgante.

O que gostaríamos de fazer é dizer à `std::swap` que, quando os objetos `Widgets` estão sendo trocados, a maneira de realizar a troca é por meio da aplicação de `swap` a seus ponteiros internos `pimpl`. Existe uma maneira de dizer exatamente isso: especializar `std::swap` para `Widget`. Aqui está a ideia básica, apesar de ela não ser compilada nesta forma:

```
namespace std {
   template<>                                     // essa é uma versão especializada
   void swap<Widget>(Widget& a,                   // de std::swap quando T for
                     Widget& b)                   // Widget; não será compilado
   {
      swap(a.pImpl, b.pImpl);                     // para trocar Widgets, apenas troque
   }                                              // seus ponteiros pImpl
}
```

O "`template<>`" no início dessa função diz que essa é uma especialização total de template para `std::swap`, e o "`<Widget>`", após o nome da função, diz que a especialização é para quando `T` for `Widget`. Em outras palavras, quando o template geral `swap` for aplicado a `Widgets`, essa é a implementação que deve ser usada. Em geral, não é permitido que alteremos os conteúdos do espaço de nomes `std`, mas nos é permitido especializar totalmente templates padrão (tal como `swap`) para tipos de nossa própria criação (como `Widget`). É isso o que estamos fazendo aqui.

Como disse, essa função não é compilada, porque ela está tentando acessar os ponteiros `pimpl` dentro de `a` e de `b`, e eles são privados. Poderíamos declarar nossa especialização como uma função amiga, mas a convenção é diferente: é fazer `Widget` declarar uma função membro pública chamada `swap` que realmente faz a troca, então especializar `std::swap` para chamar a função membro:

```
class Widget {                                  // o mesmo que acima, exceto pela
public:                                         // adição da função membro swap
  ...
  void swap(Widget& other)
  {
    using std::swap;                            // a necessidade para essa declaração
                                                // é explicada posteriormente neste Item

    swap(pImpl, other.pImpl);                   // para trocar Widgets, troque seus
  }                                             // ponteiros pImpl
  ...
};

namespace std {

  template<>                                    // especialização revisada de
  void swap<Widget>(Widget& a,                  // std::swap
                    Widget& b)
  {
    a.swap(b);                                  // para trocar Widgets, chame suas
  }                                             // funções membro swap
}
```

Isso é compilado não apenas, como também é consistente com os contêineres STL, os quais fornecem funções membro `swap` públicas e especializações de `std::swap` que chamam essas funções membro.

Suponhamos, entretanto, que `Widget` e `WidgetImpl` fossem templates de classes em vez de classes: possivelmente, poderíamos parametrizar o tipo de dados armazenado em `WidgetImpl`:

```
template<typename T>
class WidgetImpl { ... };

template<typename T>
class Widget { ... };
```

Colocar uma função membro `swap` em `Widget` (e, se precisássemos, em `WidgetImpl`) é tão fácil quanto antes, mas teremos problemas com a especialização para `std::swap`. Isso é o que queremos escrever:

```
namespace std {

  template<typename T>
  void swap<Widget<T> >(Widget<T>& a,           // erro! código ilegal!
                        Widget<T>& b)
  { a.swap(b); }

}
```

Isso parece perfeitamente racional, mas não é legal. Estamos tentando especializar parcialmente um template de função (`std::swap`), mas, apesar de

C++ permitir a especialização parcial de templates de classe, a linguagem não permite essa ação para templates de funções. Esse código não deve ser compilado (apesar de alguns compiladores o aceitarem erroneamente).

Quando você quer "especializar parcialmente" um template de função, a abordagem usual é simplesmente adicionar uma sobrecarga. Ela se pareceria com o código a seguir:

```
namespace std {

  template<typename T>                          // uma sobrecarga de std::swap
  void swap(Widget<T>& a,                       // (observe a falta de "<...>" após
            Widget<T>& b)                       // "swap"), mas veja abaixo
  { a.swap(b); }                                // por que esse não é um código válido
}
```

Em geral, é permitido sobrescrever templates de classes, mas `std` é um espaço de nomes especial, e as regras que o governam são especiais também. É permitido especializar totalmente templates em `std`, mas não é permitido adicionar *novos* templates (ou classes, ou funções, ou qualquer outra coisa) em `std`. Os conteúdos de `std` são determinados unicamente pelo comitê de padronização de C++, e estamos proibidos de modificar o que ele decidiu que deve ir lá. Infelizmente, a forma da proibição pode desestimulá-lo. Os programas que cruzam essa linha certamente compilarão e serão executados, mas seu comportamento será indefinido. Se você quiser que seu aplicativo tenha comportamento previsível, não vai adicionar novas coisas a `std`.

Então, o que fazer? Ainda precisamos que outras pessoas possam chamar `swap` e obter nossa versão específica do template mais eficiente. A resposta é simples. Continuamos declarando um `swap` não membro que chama o `swap` membro, só não declaramos o não membro como uma especialização ou uma sobrecarga de `std::swap`. Por exemplo, se todas as nossas funcionalidades relacionadas a `Widget` estão no espaço de nomes `WidgetStuff` (coisas de `Widget`), ele se pareceria com o seguinte:

```
namespace WidgetStuff {

  ...                                           // WidgetImpl com templates, etc
  template<typename T>                          // como antes, incluindo o swap para
  class Widget { ... };                         // função membro
  ...

  template<typename T>                          // função swap não membro
  void swap(Widget<T>& a,                       // não faz parte do espaço de nomes std
            Widget<T>& b)
  {
    a.swap(b);
  }
}
```

Agora, se qualquer código em qualquer lugar chamar `swap` em dois objetos `Widget`, as regras de busca de nomes em C++ (especificamente as regras conhecidas como busca dependente de argumentos ou *Koenig lookup*) en-

contrarão a versão específica de `Widget` em `WidgetStuff`. Isso é exatamente o que queremos.

Essa abordagem funciona bem tanto para classes quanto para templates de classes, então parece que devemos usá-la todas as vezes. Infelizmente, existe uma razão para especializar `std::swap` para as classes (eu a descreverei brevemente); então, se você quer que sua versão específica de classe de `swap` seja chamada em tantos contextos quantos forem possíveis (e você quer isso), precisa escrever tanto uma versão não membro no mesmo espaço de nomes de sua classe quanto uma especialização de `std::swap`.

A propósito, se você não está usando espaço de nomes, tudo acima continua sendo aplicável (ou seja, você ainda precisa de um `swap` não membro que chame o `swap` membro), mas por que você está inflando o espaço de nomes global como todas as suas classes, templates, funções, enumerações e nomes de definições de tipos? Você não tem senso de propriedade?

Tudo o que já escrevi até agora pertence aos autores de `swap`, mas vale a pena dar uma olhada em uma situação do ponto de vista do cliente. Suponhamos que você esteja escrevendo um template de função no qual precisa trocar os valores de dois objetos:

```
template<typename T>
void doSomething(T& obj1, T& obj2)
{
  ...
  swap(obj1, obj2);
  ...
}
```

Que `swap` deve ser chamado? Aquele geral em `std`, que você sabe que existe; uma especialização do `swap` geral em `std`, que pode ou não existir; ou um `swap` específico de `T`, que pode ou não existir e que pode ou não estar em um espaço de nomes (mas certamente não deveria estar em `std`)? O que você deseja é chamar uma versão específica de `T`, se existir uma, mas cair novamente de volta na versão geral em `std`, se não existir uma. Veja como você pode satisfazer seu desejo:

```
template<typename T>
void doSomething(T& obj1, T& obj2)
{
  using std::swap;                    // disponibilize std::swap nessa função
  ...
  swap(obj1, obj2);                   // chame a melhor troca para objetos do tipo T
  ...
}
```

Quando os compiladores veem a chamada a `swap`, eles buscam o `swap` correto a ser invocado. As regras de resolução de nomes garantem que serão encontradas quaisquer funções `swap` específicas de `T` no escopo global ou

no mesmo espaço de nomes do tipo `T`. (Por exemplo, se `T` for `Widget` no espaço de nomes `WidgetStuff`, os compiladores usarão a resolução que depende do argumento para encontrar `swap` em `WidgetStuff`.) Se não existir nenhum `swap` específico de `T`, os compiladores usarão `swap` em `std` graças à declaração `using` que torna `std::swap` visível nessa função. Mesmo assim, os compiladores preferirão uma especialização específica para `T` de `std::swap` em vez do template geral; então, se `std::swap` tiver sido especializada para `T`, será usada a versão especializada.

Dessa forma, é fácil obter o `swap` correto a ser chamado. A única coisa que você precisa ter cuidado é de não qualificar a chamada, porque isso afetará a forma como C++ determina a função a ser invocada. Por exemplo, se você escrevesse a chamada a `swap` desta maneira

```
std::swap(obj1, obj2);                    // a maneira errada de chamar swap
```

estaria forçando os compiladores a considerar apenas o `swap` em `std` (incluindo quaisquer especializações de template), eliminando a possibilidade de obter uma versão específica para `T` mais apropriada definida em algum outro lugar. Infelizmente, alguns programadores equivocados *realmente* qualificam as chamadas a `swap` dessa maneira, e é por isso que é importante especializar totalmente `std::swap` para suas classes: isso disponibiliza as implementações de `swap` específicas de tipo para o código escrito dessa maneira errônea. (Esse código está presente em algumas implementações da biblioteca padrão, então é de seu interesse ajudar esse código a trabalhar de forma tão eficiente quanto possível.)

Nesse ponto, já discutimos `swap` padrão, `swap` membros, `swap` não membros, especializações de `std::swap` e chamadas a `swap`; então, vamos resumir a situação.

Primeiro, se a implementação padrão de `swap` oferecer uma eficiência aceitável para a sua classe ou template de classe, você não precisa fazer nada. Qualquer um que tente trocar objetos de seu tipo obterá a versão padrão, e isso funcionará bem.

Segundo, se a implementação padrão de `swap` não for eficiente o suficiente (o que quase sempre significa que sua classe ou seu template está usando alguma variação do idioma pimpl), faça o seguinte:

1. Ofereça uma função membro `swap` pública que troque, de maneira eficiente, o valor de dois objetos de seu tipo. Por razões que explicarei em seguida, essa função nunca deve lançar uma exceção.

2. Ofereça uma função `swap` não membro no mesmo espaço de nomes de sua classe ou de seu template. Faça a função chamar o `swap` membro.

3. Se você estiver escrevendo uma classe (e não um template de classe), especialize `std::swap` para a sua classe. Faça ela também chamar o `swap` que é uma função membro.

Por fim, se você estiver chamando `swap`, certifique-se de incluir uma declaração `using` para tornar `std::swap` visível em sua função, então chame `swap` sem qualquer qualificação de espaço de nomes.

A única ponta solta é meu aviso de fazer a versão membro de `swap` nunca lançar exceções. Isso ocorre porque uma das aplicações mais úteis de `swap` é ajudar classes (e templates de classes) a oferecer a mais forte garantia de segurança de exceções. O Item 29 fornece todos os detalhes, mas a técnica se baseia na premissa de que a versão membro de `swap` nunca lança exceções. Essa restrição se aplica apenas à versão membro! Ela não pode ser aplicada para a versão não membro, porque a versão padrão de `swap` se baseia em construção de cópia e em atribuição por cópia e, em geral, é permitido a essas duas funções lançarem exceções. Quando você escrever uma versão personalizada de `swap`, aí então você estará oferecendo, em geral, mais do que apenas uma maneira eficiente de trocar valores; estará também oferecendo uma versão que não lança exceções. Como regra geral, essas duas características de `swap` andam de mãos dadas, porque `swap`s altamente eficazes são quase sempre baseadas em operações sobre tipos predefinidos (como nos ponteiros subjacentes ao idioma pimpl), e as operações em tipos predefinidos nunca lançam exceções.

Lembretes

» Forneça uma função membro `swap` quando `std::swap` for ineficiente para o seu tipo. Certifique-se de que seu `swap` não lance exceções.

» Se você oferecer um `swap` membro, ofereça também um `swap` não membro que chama o membro. Para as classes (não templates), especialize `std::swap` também.

» Quando estiver chamando `swap`, use uma declaração `using` para `std::swap` e chame `swap` sem qualificações de espaços de nomes.

» É aceitável especializar totalmente templates de `std` para tipos definidos pelo usuário, mas nunca tente adicionar algo completamente novo a `std`.

CAPÍTULO 5

IMPLEMENTAÇÕES

Na maioria das situações, o essencial é chegar às definições adequadas para suas classes (e templates de classes) e às declarações apropriadas para suas funções (e templates de funções). Quando adequadas, as implementações correspondentes são, em sua maioria, diretas. Mas você ainda deve prestar atenção em alguns pontos. Definir variáveis muito cedo pode causar problemas de desempenho. Usar conversões implícitas em demasia pode levar a códigos que são lentos, difíceis de manter e infectados com bugs sutis. Retornar manipuladores para os membros internos de um objeto pode vencer o encapsulamento e deixar os clientes com manipuladores soltos. Não considerar o impacto das exceções pode levar a recursos vazados e estruturas de dados corrompidas. As internalizações fervorosas podem causar um inchaço no código. O acoplamento excessivo pode resultar em tempos de compilação inaceitavelmente longos.

Todos esses problemas podem ser evitados. Este capítulo explica como.

Item 26: Postergue a definição de variáveis tanto quanto possível

Sempre que você define uma variável de um tipo com um construtor ou um destrutor, incorre no custo de construção quando o controle alcança a definição da variável e no custo de destruição sempre que a variável sai de escopo. Existe um custo associado a variáveis não utilizadas, então você deve evitá-las sempre que puder.

Você provavelmente está pensando que nunca definiu variáveis que não são usadas, mas talvez precise pensar nisso novamente. Considere a seguinte função, que retorna uma versão criptografada de uma senha, desde que a senha seja longa o suficiente. Se a senha for muito pequena, a função lança uma exceção do tipo `logic_error` (erro lógico), que é definida na biblioteca padrão de C++ (veja o Item 54):

```
// esta função define a variável "encrypted" muito cedo
std::string encryptPassword(const std::string& password)
{
   using namespace std;

   string encrypted;

   if (password.length( ) < MinimumPasswordLength) {
      throw logic_error("Password is too short");
   }
   ...                                  // faça o que for necessário para colocar uma
                                        // versão criptografada da senha em encrypted
   return encrypted;
}
```

O objeto `encrypted` (criptografado) não é *completamente* não usado nessa função, mas não é usado se for lançada uma exceção. Ou seja, você pagará pela construção e pela destruição de `encrypted` mesmo que `encryptPassword` (criptografar senha) lance uma exceção. Como resultado, você estaria melhor se tivesse postergado a definição de `encrypted` até que *soubesse* que precisaria dele:

```
// esta função posterga a definição de encrypted até que ela seja realmente necessária
std::string encryptPassword(const std::string& password)
{
   using namespace std;

   if (password.length( ) < MinimumPasswordLength) {
      throw logic_error("Password is too short");
   }

   string encrypted;
   ...                                  // faça o que for necessário para colocar uma
                                        // versão criptografada da senha em encrypted
   return encrypted;
}
```

Esse código ainda não está tão enxuto quanto poderia, porque `encrypted` é definido sem nenhum argumento de inicialização. Ou seja, seu construtor padrão será usado. Em muitos casos, a primeira coisa que você fará a um objeto é lhe dar algum valor, frequentemente por meio de uma atribuição. O Item 4 explica porque construir um objeto com seu construtor padrão e depois atribuir valores a ele é menos eficiente do que inicializá-lo com o valor que você realmente quer que ele tenha. Essa análise se aplica aqui também. Por exemplo, suponhamos que a parte difícil de `encryptPassword` seja realizada nesta função:

```
void encrypt(std::string& s);           // encripts está no lugar
```

A função `encryptPassword` poderia ser implementada da seguinte forma, apesar de não ser a melhor maneira de fazer isso:

```
// esta função posterga a definição de encrypted até que
// seja necessária, mas mesmo assim ela é desnecessariamente ineficiente
std::string encryptPassword(const std::string& password)
{
   ...                                  // importa std e verifica o tamanho como acima

   std::string encrypted;               // encrypted criado com o construtor padrão
   encrypted = password;                // atribui a encrypted

   encrypt(encrypted);
   return encrypted;
}
```

Uma abordagem preferível é inicializar `encrypted` com `password`, evitando a desnecessária, e potencialmente cara, construção padrão:

```
// por fim, a melhor maneira de definir e inicializar encrypted
std::string encryptPassword(const std::string& password)
{
   ...                                  // verifica o tamanho

   string encrypted(password);          // define e inicializa
                                        // através do construtor de cópia
   encrypt(encrypted);
   return encrypted;
}
```

Isso sugere o real significado de "tanto quanto possível" no título deste item. Você deve não apenas postergar a definição de uma variável até precisar dela, como também tentar postergar a definição até que você tenha argumentos de inicialização para ela. Ao fazer isso, você evita construir e destruir objetos desnecessários e evita construções padrão desnecessárias. Além disso, ajuda a documentar o propósito das variáveis ao inicializá-las em contextos nos quais seu propósito é claro.

"Mas e os laços?", você deve estar se perguntando. Se uma variável é usada apenas dentro de um laço, é melhor defini-la fora do laço e fazer uma atribuição a ela em cada iteração do laço, ou definir a variável dentro do laço? Ou seja, qual dessas estruturas gerais é melhor?

```
// Abordagem A: definir fora do laço

Widget w;
for (int i = 0; i < n; ++i) {
   w = algum valor dependente de i;
   ...
}
```

```
// Abordagem B: definir dentro do laço

for (int i = 0; i < n; ++i) {
   Widget w(some value dependent on i);
   ...
}
```

Aqui, troquei um objeto do tipo `string` para um objeto do tipo `Widget` para evitar quaisquer preconceitos com o custo de realizar uma construção, uma destruição ou uma atribuição para o objeto.

Em termos de operações de `Widget`, o custo dessas duas abordagens são os seguintes:

- Abordagem A: 1 construtor + 1 destrutor + *n* atribuições
- Abordagem B: *n* construtores + *n* destrutores

Para as classes em que uma atribuição custa menos do que um par construtor/destrutor, a Abordagem A é geralmente mais eficiente. Esse é o caso em que *n* se torna grande. Caso contrário, a Abordagem B é provavelmente melhor. Além disso, a Abordagem A faz o nome `w` ser visível em um escopo maior (aquele que contém o laço) do que a Abordagem B, algo que é contrário à inteligibilidade e à capacidade de manutenção. Como resultado, a menos que você saiba (1) que a atribuição é mais barata do que um par construtor/destrutor e (2) que está lidando com uma parte sensível em relação ao desempenho de seu código, você deve usar a Abordagem B como padrão.

Lembrete

» Postergue as definições de variáveis tanto quanto possível. Isso aumenta a clareza dos programas e melhora sua eficiência.

Item 27: Minimize as conversões explícitas

As regras de C++ são projetadas para garantir que os erros de tipos sejam impossíveis. Em teoria, se seu programa é compilado corretamente, ele não está tentando realizar nenhuma operação insegura ou sem sentido em quaisquer objetos. Essa é uma garantia valiosa. Você não deve perdê-la por pouca coisa.

Infelizmente, as conversões explícitas (castings) subvertem o sistema de tipos. Isso pode levar a todos os tipos de problemas, e alguns deles são fáceis de serem reconhecidos, outros extraordinariamente sutis. Se você está chegando a C++ vindo de C, de Java ou de C#, tome nota, pois a conversão explícita nessas linguagens é algo mais necessário e menos perigoso do que em C++. Mas C++ não é C. Não é Java. Não é C#. Nessa linguagem, a conversão explícita é um recurso que você deve abordar com grande respeito.

Vamos começar com uma revisão da sintaxe de conversão explícita, porque existem, normalmente, três maneiras diferentes de escrever a mesma conversão explícita. As conversões explícitas no estilo de C se parecem com o seguinte:

```
(T) expressão                    // converte explicitamente expressão como do tipo T
```

As conversões explícitas no estilo de funções usam a seguinte sintaxe:

```
T(expressão)                     // converte explicitamente expressão como do tipo T
```

Não existe diferença no significado entre essas duas formas; é puramente uma questão de onde você coloca os parênteses. Eu chamo essas duas formas de *conversões explícitas no estilo antigo*.

C++ também oferece quatro novas formas de conversão explícita (frequentemente chamadas de *conversões explícitas no novo estilo* ou *no estilo de C++*):

```
const_cast<T>(expressão)
dynamic_cast<T>(expressão)
reinterpret_cast<T>(expressão)
static_cast<T>(expressão)
```

Cada uma delas serve a um propósito distinto:

- Em geral, usa-se `const_cast` para descartar a constância dos objetos. É a única conversão explícita no estilo de C++ que pode fazer isso.
- Usa-se `dynamic_cast` principalmente para realizar conversões explícitas seguras para subtipos (downcastings), ou seja, para determinar se um objeto é de um tipo em particular em uma hierarquia de herança. É a única conversão explícita que não pode ser realizada usando-se a sintaxe do estilo antigo. Também é a única conversão explícita que pode ter um custo significativo em tempo de execução. (Fornecerei os detalhes sobre isso daqui a pouco.)
- Usa-se `reinterpret_cast` para conversões explícitas de baixo nível que levam a resultados dependentes de implementação (ou seja, não portáveis), como converter explicitamente um ponteiro em um inteiro. Essas conversões explícitas devem ser raras fora de código de baixo nível. Eu as uso apenas uma vez neste livro, e apenas quando estou discutindo como você deveria escrever um alocador de depuração para memória bruta (veja o Item 50).
- Pode-se usar `static_cast` para forçar as conversões implícitas (por exemplo, objetos não constantes em objetos constantes – como no Item 3, `int` em `double`, etc). Também pode ser usada para realizar o inverso de muitas conversões como essas (por exemplo, ponteiros `void*` em ponteiros com tipo, ponteiros para tipos base em ponteiros para tipos derivados), apesar de ela não poder converter explicitamente de constantes para objetos não constantes (apenas `const_cast` pode fazer isso).

As conversões explícitas no estilo antigo continuam sendo válidas, mas é preferível usar as novas formas. Primeiro, elas são muito mais fáceis de serem identificadas no código (tanto por seres humanos quanto por ferramentas como o `grep`), simplificando o processo de encontrar locais no código em que o sistema de tipos esteja sendo subvertido. Segundo, o propósito mais direcionado especificado por cada uma das conversões explícitas faz com que seja possível para os compiladores diagnosticarem os erros de uso. Por exemplo, se você tentar descartar a constância usando uma conversão explícita no novo estilo que não seja `const_cast`, seu código não será compilado.

Praticamente, a única vez que uso uma conversão explícita no estilo antigo é quando quero chamar um construtor explícito (`explicit`) de forma a passar um objeto para uma função. Por exemplo:

```
class Widget {
public:
  explicit Widget(int size);
  ...
};

void doSomeWork(const Widget& w);

doSomeWork(Widget(15));                 // cria um Widget a partir de um int
                                        // com conversão explícita no estilo de função

doSomeWork(static_cast<Widget>(15));    // cria um Widget a partir de um int
                                        // com conversão explícita no estilo de C++
```

De certa forma, a criação deliberada de objetos não "se parece" com uma conversão explícita; então, eu, provavelmente, usaria uma conversão explícita no estilo de função em vez de `static_cast` nesse caso. Assim, mais uma vez, um código que leva a um descarte normalmente parece bastante razoável quando você o escreve; talvez seja melhor ignorar os sentimentos e usar sempre as conversões explícitas no novo estilo.

Muitos programadores acreditam que as conversões explícitas não fazem nada além de dizer aos compiladores que tratem um tipo como outro, mas isso é um engano. As conversões de tipo de qualquer natureza (sejam elas explícitas por meio de conversões explícitas ou implícitas por meio dos compiladores) frequentemente levam a códigos que são aplicados em tempo de execução. Por exemplo, no trecho de código a seguir,

```
int x, y;
...
double d = static_cast<double>(x)/y;    // divide x por y, mas usa
                                        // divisão de ponto flutuante
```

é quase certo que a conversão explícita de `int x` para um `double` gerará código, porque, na maioria das arquiteturas, a representação subjacente para um `int` é diferente da de um `double`. Isso talvez não seja surpreendente, mas este exemplo pode abrir um pouco seus olhos:

```
class Base { ... };

class Derived: public Base { ... };

Derived d;

Base *pb = &d;                          // implicitamente convertido Derived* ⇒ Base*
```

Aqui estamos apenas criando um ponteiro da classe-base para um objeto da classe derivada, mas, algumas vezes, os valores dos dois ponteiros não serão iguais. Quando esse for o caso, é aplicado um deslocamento em *tempo de execução* para o ponteiro `Derived*` para se obter o valor correto do ponteiro `Base*`.

Esse último exemplo demonstra que um único objeto (por exemplo, um objeto do tipo `Derived`) pode ter mais de um endereço (ou seja, seu endereço quando apontado por um ponteiro `Base*` e seu endereço quando apontado por um ponteiro `Derived*`). Isso não pode acontecer em C. Não pode acontecer em Java. Não pode acontecer em C#. Mas *acontece* em C++. Na verdade, quando se está usando herança múltipla, acontece quase o tempo todo, mas pode acontecer com herança simples também. Dentre outras coisas, isso quer dizer que, em geral, você deve evitar fazer pressupostos sobre a forma como as coisas ocorrem em C++, e, certamente, não deve realizar conversões explícitas com base nesses pressupostos. Por exemplo, converter endereços de objetos em ponteiros `char*` e depois usar aritmética de ponteiros sobre eles quase sempre leva a comportamentos indefinidos.

Mas observe que eu disse que um deslocamento é "às vezes" necessário. A maneira como os objetos são criados e a maneira como seus endereços são calculados variam de um compilador para outro. Significa que, só porque suas conversões explícitas do tipo "eu sei como as coisas ocorrem" funcionam em uma plataforma, não significa que funcionarão em outras. O mundo está repleto de programadores que aprenderam essa lição da maneira mais difícil.

Um fato interessante sobre as conversões explícitas é que é fácil escrever algo que parece correto (e pode ser correto em outras linguagens), mas que está errado. Muitos frameworks de aplicação, por exemplo, exigem que as implementações de funções membros virtuais em classes derivadas chamem suas correspondentes da classe-base primeiro. Suponhamos que tivéssemos uma classe-base `Window` (janela) e uma classe derivada `SpecialWindow` (janela especial), e ambas definissem a função virtual `onResize` (ao redimensionar). Suponhamos, ainda, que a função `onResize` de `SpecialWindow` espere invocar `onResize` de `Window` primeiro. Veja uma maneira de implementar isso de modo que pareça correto, mas que não é:

```
class Window {                                  // classe-base
public:
   virtual void onResize( ) { ... }             // impl. de onResize na classe-base
   ...
};

class SpecialWindow: public Window {            // classe derivada
public:
   virtual void onResize( ) {                   // impl. de onResize na classe derivada;
      static_cast<Window>(*this).onResize( );   // converte *this para Window
                                                // então chama seu onResize
                                                // não funciona!

      ...                                       // realiza tarefas específicas
   }                                            // de SpecialWindow

   ...
};
```

Destaquei a conversão explícita no código. (É um novo estilo de conversão explícita, mas usar uma conversão explícita no estilo antigo não mudaria nada.) Como você deveria esperar, o código converte explicitamente

`*this` em um `Window`. A chamada resultante para `onResize` então invoca `Window::onResize`. O que você não deve esperar é que ela não invoque a função no objeto atual! Em vez disso, a conversão explícita cria uma nova *cópia* temporária da parte da classe-base de `*this`, e depois invoca `onResize` na cópia! O código acima não chama `Window::onResize` no objeto atual e então realiza ações específicas de `SpecialWindow` no objeto atual – ele chama `Window::onResize` em uma *cópia da parte da classe-base* do objeto atual antes de realizar ações específicas de `SpecialWindow` no objeto atual. Se `Window::onResize` modificar o objeto atual (uma possibilidade bastante remota, considerando que `onResize` é uma função membro não constante), o objeto atual não será modificado. Em vez disso, uma *cópia* desse objeto será modificada. Se `SpecialWindow::onResize` modificar o objeto atual, no entanto, o objeto atual será modificado, levando à possibilidade de que o código deixará o objeto atual em um estado inválido, em que não foram feitas modificações na classe-base, mas foram nas classes derivadas.

A solução é eliminar a conversão explícita, substituindo-a por aquilo que você realmente quer dizer. Você não quer enganar os compiladores para que tratem `*this` como objeto da classe-base; você quer chamar a versão da classe-base de `onResize` no objeto atual. Então diga o seguinte:

```
class SpecialWindow: public Window {
public:
   virtual void onResize( ) {
      Window::onResize( );                    // chama Window::onResize
      ...                                     // em *this
   }

   ...

};
```

Esse exemplo também demonstra que, se você quiser realizar uma conversão explícita, é sinal de que pode estar tratando as coisas da maneira errada. Esse é especialmente o caso se seu desejo for usar `dynamic_cast`.

Antes de mergulharmos nas implicações de projeto de `dynamic_cast`, vale a pena observar que muitas implementações de `dynamic_cast` podem ser bastante lentas. Por exemplo, pelo menos uma implementação comum se baseia, em parte, em comparações de cadeias nos nomes das classes. Se você está realizando um `dynamic_cast` em um objeto em uma hierarquia de classes com quatro níveis de profundidade, cada `dynamic_cast` nessa implementação poderia lhe custar até quatro chamadas a `strcmp` para comparar nomes de classes. Uma hierarquia mais profunda ou uma que use herança múltipla seria mais cara. Existem razões para algumas implementações funcionarem dessa forma (precisam dar suporte à ligação dinâmica). Independentemente disso, além de ser cauteloso sobre as conversões explícitas de uma maneira geral, você deve ser especialmente cauteloso em relação a `dynamic_cast`s em código em que o desempenho é extremamente importante.

A necessidade do uso de `dynamic_casts` geralmente aparece porque você quer realizar operações da classe derivada em algo que acredita ser um objeto da classe derivada, mas você tem apenas um ponteiro ou uma referência para a base por meio dos quais você manipula o objeto. Existem duas maneiras gerais de evitar esse problema.

Primeiro, use contêineres para armazenar ponteiros (em geral, ponteiros espertos – veja o Item 13) diretamente para objetos da classe derivada, eliminando a necessidade de manipular esses objetos por meio de interfaces da classe-base. Por exemplo, se em nossa hierarquia `Window/SpecialWindow` apenas `SpecialWindow` possibilita que a janela realize a operação de piscar, em vez de fazer o seguinte,

```
class Window { ... };

class SpecialWindow: public Window {
public:
  void blink( );
  ...
};

typedef                                                  // veja o Item 13 para info.
  std::vector<std::tr1::shared_ptr<Window> > VPW;        // em tr1::shared_ptr

VPW winPtrs;

...

for (VPW::iterator iter = winPtrs.begin( );              // código indesejável:
     iter != winPtrs.end( );                             // usa dynamic_cast
     ++iter) {
  if (SpecialWindow *psw = dynamic_cast<SpecialWindow*>(iter->get( )))
    psw->blink();
}
```

tente fazer isto:

```
typedef std::vector<std::tr1::shared_ptr<SpecialWindow> > VPSW;

VPSW winPtrs;

...

for (VPSW::iterator iter = winPtrs.begin( );             // código melhor: não
     iter != winPtrs.end( );                             // usa dynamic_cast
     ++iter)
  (*iter)->blink( );
```

Obviamente, essa abordagem não permitirá armazenar ponteiros para todos os tipos derivados de `Window` no mesmo contêiner. Para trabalhar com tipos diferentes de janelas, você pode precisar de contêineres seguros em relação a múltiplos tipos.

Uma alternativa que deixará manipular todos os tipos derivados de `Window` por meio de uma interface da classe-base é fornecer funções virtuais na classe-base que deixem que você faça o que precisa. Por exemplo, apesar de só `SpecialWindows` poder piscar, talvez faça sentido declarar a função na classe-base, oferecendo uma implementação padrão que não faz nada:

```
class Window {
public:
   virtual void blink( ) { }                    // a impl. padrão é vazia
   ...                                          // veja o Item 34 para ver porque
};                                              // uma implementação padrão
                                                // pode ser uma má ideia

class SpecialWindow: public Window {
public:
   virtual void blink( ) { ... }                // nesta classe, blink
   ...                                          // faz algo
};

typedef std::vector<std::tr1::shared_ptr<Window> > VPW;

VPW winPtrs;                                    // o contêiner mantém
                                                // (ponteiros para) todos os possíveis
...                                             // tipos de janela

for (VPW::iterator iter = winPtrs.begin( );
     iter != winPtrs.end( );
     ++iter)                                    // note a falta de
   (*iter)->blink( );                           // dynamic_cast
```

Nenhuma dessas abordagens – usar contêineres seguros em relação a tipos ou mover funções virtuais para cima na hierarquia – é universalmente aplicável, mas, em muitos casos, fornecem uma alternativa viável à conversão explícita usando `dynamic_cast`. Quando elas fazem isso, você deve utilizá-las.

Algo que você definitivamente deve evitar são projetos que envolvam `dynamic_cast`s em cascata, ou seja, qualquer coisa parecida com:

```
class Window { ... };

...                                             // as classes dinâmicas são definidas aqui

typedef std::vector<std::tr1::shared_ptr<Window> > VPW;

VPW winPtrs;

...

for (VPW::iterator iter = winPtrs.begin( ); iter != winPtrs.end( ); ++iter)
{
   if (SpecialWindow1 *psw1 =
         dynamic_cast<SpecialWindow1*>(iter->get( ))) { ... }

   else if (SpecialWindow2 *psw2 =
               dynamic_cast<SpecialWindow2*>(iter->get( ))) { ... }

   else if (SpecialWindow3 *psw3 =
               dynamic_cast<SpecialWindow3*>(iter->get( ))) { ... }

   ...
}
```

Esse código C++ gera código que é grande e lento; além disso, é inflexível, pois, cada vez que a hierarquia de classes de `Window` muda, todo o código da hierarquia deve ser examinado para ver se ele precisa ser atualizado. (Por exemplo, se for adicionada uma nova classe derivada, um novo desvio condicional deve ser adicionado à cascata acima.) O código que se parece com esse deve, quase sempre, ser substituído por algo baseado em chamadas a funções virtuais.

Um bom código C++ usa pouquíssimas conversões explícitas, mas, geralmente, não é prático se livrar de todas elas. A conversão explícita de `int` para `double` na página 118, por exemplo, é um uso razoável de uma conversão explícita, apesar de não ser estritamente necessária. (O código poderia ser reescrito para declarar uma nova variável do tipo `double` que é inicializada com o valor de `x`.) Como a maioria das construções suspeitas, as conversões explícitas devem ser isoladas tanto quanto possível, em geral ocultas dentro de funções cujas interfaces isolem os chamadores do trabalho sujo que está sendo feito dentro delas.

Lembretes

» Evite conversões explícitas sempre que isso for prático, principalmente `dynamic_casts` em código sensível ao desempenho. Se um projeto exigir conversões explícitas, tente desenvolver uma alternativa livre de conversões explícitas.

» Quando o uso de conversões explícitas for necessário, tente ocultá-las dentro de uma função. Os clientes podem então chamar a função em vez de colocar conversões explícitas em seus próprios códigos.

» Prefira as conversões explícitas no estilo de C++ às conversões explícitas no estilo antigo. Elas são mais fáceis de serem vistas, e são mais específicas em relação ao que fazem.

Item 28: Evite retornar "manipuladores" para objetos internos

Suponhamos que você esteja trabalhando em uma aplicação que envolva retângulos. Cada retângulo pode ser representado pelo lado superior esquerdo e pelo lado inferior direito. Para manter um objeto da classe `Rectangle` (retângulo) pequeno, você pode decidir que os pontos que definem sua extensão não devem ser armazenados na classe `Rectangle` propriamente dita, mas em uma estrutura auxiliar para a qual `Rectangle` aponte:

```
class Point {                                   // classe para representar pontos
public:
   Point(int x, int y);
   ...

   void setX(int newVal);
   void setY(int newVal);
   ...
};
```

```
struct RectData {                                   // dados de Point para um Rectangle
   Point ulhc;                                      // ulhc = "canto superior esquerdo"
   Point lrhc;                                      // lrhc = "canto inferior direito"
};

class Rectangle {
   ...

private:
   std::tr1::shared_ptr<RectData> pData;            // veja o Item 13 para obter mais informações
};                                                  // sobre tr1::shared_ptr
```

Como os clientes de `Rectangle` precisarão ser capazes de determinar a extensão de um `Rectangle`, a classe fornece as funções `upperLeft` (canto superior esquerdo) e `lowerRight` (canto inferior direito). Entretanto, `Point` (ponto) é um tipo definido pelo usuário; então, ciente da observação do Item 20, de que, em geral, passar tipos definidos pelo usuário por referência é mais eficiente do que passá-los por valor, essas funções retornam referências aos objetos `Point` subjacentes:

```
class Rectangle {
public:
   ...
   Point& upperLeft( ) const { return pData->ulhc; }
   Point& lowerRight( ) const { return pData->lrhc; }
   ...
};
```

Esse projeto será compilado, mas está errado. Na verdade, ele é contraditório. Por um lado, `upperLeft` e `lowerRight` são declaradas como funções membro constantes, pois são projetadas apenas para oferecer aos clientes uma maneira de aprender quais são os pontos de um `Rectangle`, e não para deixar que os clientes modifiquem o retângulo (veja o Item 3). Por outro lado, ambas as funções retornam referências para os dados internos privados – referências que os chamadores podem usar para modificar os dados internos! Por exemplo:

```
Point coord1(0, 0);
Point coord2(100, 100);

const Rectangle rec(coord1, coord2);                // rec é um retângulo constante de
                                                    // (0, 0) a (100, 100)

rec.upperLeft().setX(50);                           // agora rec vai de
                                                    // (50, 0) a (100, 100)!
```

Aqui, observe como o chamador de `upperLeft` consegue usar a referência retornada para um dos membros de dados `Point` de `rec` para modificar esse membro. Mas `rec` é supostamente constante (`const`)!

Isso imediatamente nos leva a duas lições. Primeiro, um membro de dados só é encapsulado na mesma medida que a função mais acessível que retorna uma referência a ele. Nesse caso, apesar de `ulhc` e `lrhc` serem declarados como privados, eles são públicos, pois as funções públicas `upperLeft` e `lowerRight` retornam referências a eles. Segundo, se uma função membro constante retorna uma referência a dados associados com um objeto

que são armazenados fora do objeto propriamente dito, o chamador da função pode modificar esses dados (essa é apenas uma das limitações da constância bit a bit – veja o Item 3).

Tudo o que fizemos envolvia funções membro retornando referências, mas, se retornassem ponteiros ou iteradores, os mesmos problemas existiriam pelas mesmas razões. Referências, ponteiros e iteradores são manipuladores (maneiras para chegar a outros objetos), e retornar um manipulador para os membros internos de um objeto sempre apresenta o risco de comprometer o encapsulamento de um objeto. Como vimos, isso também pode levar a funções membro constantes que permitem que o estado de um objeto seja modificado.

Geralmente, pensamos nos membros "internos" como seus membros de dados, mas as funções membro não acessíveis ao público geral (ou seja, que são protegidas ou privadas) são parte dos membros internos de um objeto também. Dessa forma, é importante não retornar manipuladores para eles. Isso significa que você nunca deve ter uma função membro retornando um ponteiro para uma função membro menos acessível. Se fizer isso, o nível de acesso será o da função mais acessível, porque os clientes vão conseguir obter um ponteiro para a função menos acessível e chamar essa função pelo ponteiro.

As funções que retornam ponteiros para funções membro não são comuns; entretanto, vamos voltar nossa atenção novamente para a classe `Rectangle` e suas funções membro `upperLeft` e `lowerRight`. Ambos os problemas que identificamos para essas funções poderiam ser eliminados simplesmente aplicando `const` para seus tipos de retorno:

```
class Rectangle {
public:
   ...
   const Point& upperLeft( ) const { return pData->ulhc; }
   const Point& lowerRight( ) const { return pData->lrhc; }
   ...
};
```

Com esse projeto alterado, um cliente pode ler os pontos definindo um retângulo, mas não pode escrevê-los. Ou seja, declarar `upperLeft` e `lowerRight` como `const` não é mais uma mentira, porque essas funções não permitem mais que seus chamadores modifiquem o estado do objeto. Em relação ao problema de encapsulamento, nossa intenção sempre foi deixar que os clientes vejam os pontos que fazem parte de um retângulo, então esse é um relaxamento deliberado do encapsulamento. Mais importante, esse é um relaxamento *limitado*: é fornecido apenas acesso somente leitura para essas funções. O acesso de gravação ainda é proibido.

Mesmo assim, `upperLeft` e `lowerRight` ainda retornam manipuladores para os membros internos de um objeto, e isso pode ser problemático de outras formas. Em particular, pode levar a *manipuladores perdidos*: manipuladores que se referem a partes dos objetos que não existem mais. A

fonte mais comum desses objetos que desaparecem são os valores de retorno de funções. Por exemplo, considere uma função que retorna a caixa delimitadora de um objeto GUI na forma de um retângulo:

```
class GUIObject { ... };

const Rectangle                          // retorna um retângulo por
   boundingBox(const GUIObject& obj);    // valor; veja o Item 3 para saber por que
                                         // o tipo de retorno é constante
```

Agora, considere como um cliente poderia usar essa função:

```
GUIObject *pgo;                          // faça com que pgo aponte para
...                                      // algum GUIObject

const Point *pUpperLeft =                // obtém um ponteiro para o
   &(boundingBox(*pgo).upperLeft( ));    // ponto superior esquerdo de sua
                                         // caixa ao redor
```

A chamada a `boundingBox` (caixa delimitadora) retornará um objeto novo e temporário `Rectangle`. Esse objeto não tem nome, então vamos chamá-lo de *temp*. `upperLeft` será, então, chamada em `temp`, e essa chamada retornará uma referência a uma parte interna de `temp`, em particular, a um dos pontos (`Point`) que a compõe. `pUpperLeft` (um ponteiro para o canto superior esquerdo) apontará para esse objeto da classe `Point`. Até agora tudo bem, mas não terminamos ainda, porque; no final da sentença, o valor de retorno de `boundingBox` – *temp* – será destruído, e isso indiretamente levará à destruição dos pontos de *temp*. Isso, por sua vez, deixará `pUpperLeft` apontando para um objeto que já não existe mais; `pUpperLeft` estará perdido no final da sentença que o criou!

É por isso que qualquer função que retorna um manipulador a uma parte interna do objeto é perigosa. Não importa se o manipulador é um ponteiro, uma referência ou um iterador. Não importa se ele é qualificado com `const`. Não importa se a função membro que retorna o manipulador é, ela própria, constante. Tudo o que importa é que um manipulador está sendo retornado, pois, uma vez que isso for feito, você corre o risco de o manipulador viver mais que o objeto a que ele faz referência.

Isso não significa que você *nunca* deva ter uma função membro que retorne um manipulador. Algumas vezes, você precisa. Por exemplo, `operator[]` permite que você pegue caracteres individuais de cadeias (`string`) e de vetores (`vector`), e esse operador funciona retornando referências aos dados nos contêineres (veja o Item 3) – dados que são destruídos quando da destruição dos contêineres. Mesmo assim, essas funções são a exceção, e não a regra.

Lembrete

» Evite retornar manipuladores (referências, ponteiros, ou iteradores) para objetos internos. Isso aumenta o encapsulamento, ajuda as funções membro constantes a agir como constantes e minimiza a criação de manipuladores perdidos.

Item 29: Busque a criação de código seguro em relação a exceções

A segurança em relação a exceções é parecida com a gravidez... mas guarde essa ideia por um momento. Não podemos realmente falar sobre reprodução até que estejamos familiarizados com o cortejo.

Suponhamos que tivéssemos uma classe para representar menus GUI com imagens de fundo. A classe foi projetada para ser usada em um ambiente com múltiplas linhas de execução, então tem um objeto de exclusão mútua (mutex) para o controle de concorrência:

```
class PrettyMenu {
public:
  ...
  void changeBackground(std::istream& imgSrc);      // troca a imagem
  ...                                               // de fundo

private:
  Mutex mutex;                                      // mutex para esse objeto

  Image *bgImage;                                   // imagem de fundo atual
  int imageChanges;                                 // nº de vezes que a imagem foi alterada
};
```

Considere a seguinte implementação da função `changeBackground` (troca o fundo) de `PrettyMenu` (menu bonito):

```
void PrettyMenu::changeBackground(std::istream& imgSrc)
{
  lock(&mutex);                                     // adquire mutex (como no Item 14)

  delete bgImage;                                   // livra-se do fundo antigo
  ++imageChanges;                                   // atualiza o contador de mudanças de imagens
  bgImage = new Image(imgSrc);                      // instala o novo fundo

  unlock(&mutex);                                   // libera o mutex
}
```

Da perspectiva da segurança das exceções, essa função é tão ruim quanto poderia ser. Existem dois requisitos para a segurança das exceções, e ela não satisfaz nenhum deles.

Quando uma exceção é lançada, as funções seguras em relação a exceções:

- **Não vazam recursos.** O código acima falha nesse teste, porque, se a expressão "`new Image(imgSrc)`" levar a uma exceção, a chamada a `unlock` (destrancar) nunca será executada e o mutex será mantido para sempre.

- **Não permitem que as estruturas de dados fiquem corrompidas.** Se "`new Image(imgSrc)`" lançar uma exceção, `bgImage` (imagem de fundo) ficará apontando para um objeto apagado. Além disso, `imageChanges` (mudanças na imagem) foi incrementada, mesmo que não seja verdade que foi instalada uma nova imagem. (Por outro lado, a imagem antiga foi,

definitivamente, eliminada, então suponho que você poderia argumentar que a imagem "mudou".)

É fácil tratar da questão do vazamento de recursos, pois o Item 13 explica como usar objetos para gerenciar recursos e o Item 14 introduz a classe `Lock` (cadeado) como forma de garantir que os objetos de exclusão mútua sejam liberados de uma maneira temporalmente adequada:

```
void PrettyMenu::changeBackground(std::istream& imgSrc)
{
  Lock ml(&mutex);                    // do Item 14: adquira o objeto de exclusão
                                      // mútua e garanta sua posterior liberação
delete bgImage;
++imageChanges;
bgImage = new Image(imgSrc);
}
```

Uma das melhores coisas sobre as classes de gerenciamento de recursos como `Lock` é que elas normalmente tornam as funções menores. Você consegue ver como a chamada a `unlock` não é mais necessária? Como regra geral, menos código é código melhor, porque existe menos para dar errado e menos para ser entendido quando forem efetuadas mudanças.

Deixado o vazamento de recursos para trás, podemos focar nossa atenção na questão das estruturas de dados corrompidas. Aqui, temos uma escolha, mas, antes de escolhermos, precisamos confrontar a terminologia que define nossas escolhas.

As funções seguras em relação a exceções oferecem três garantias:

- As funções que oferecem **a garantia básica** prometem que, se uma exceção for lançada, tudo no programa permanece em um estado válido. Nenhum objeto ou estrutura de dados fica corrompido, e todos os objetos estão em um estado consistente internamente (ou seja, todas as invariantes de classes são satisfeitas). Entretanto, o estado exato do programa pode não ser previsível. Por exemplo, poderíamos escrever `changeBackground` de forma que, se uma exceção fosse lançada, o objeto `PrettyMenu` continuaria a ter a imagem de fundo antiga, ou poderia ter alguma imagem padrão de fundo, mas os clientes não conseguiriam prever qual. (Para descobrir, eles poderiam, supostamente, chamar alguma função membro que dissesse qual era a imagem de fundo.)

- As funções que oferecem **garantia forte** prometem que, se uma exceção for lançada, o estado do programa não é modificado. As chamadas a essas funções são consideradas *atômicas*, no sentido de que, se forem bem-sucedidas, são completamente bem-sucedidas, e, se falharem, o estado do programa é idêntico àquele em que essas funções nunca foram chamadas.

  Trabalhar com funções que oferecem garantia forte é mais fácil do que trabalhar com funções que oferecem apenas a garantia básica, pois,

após chamar uma função que oferece garantia forte, existem apenas duas possibilidades para os estados do programa: o esperado após a execução bem-sucedida da função, ou o estado que existia no momento em que a função foi chamada. Em contraste a isso, se uma chamada para uma função que oferece apenas a garantia básica levar a uma exceção, o programa pode estar em *qualquer* estado válido.

- As funções que oferecem **garantia de não lançar exceções** prometem nunca lançar exceções, porque sempre fazem o que prometem. Todas as operações em tipos predefinidos (por exemplo, inteiros, ponteiros, etc) são não lançadoras (ou seja, oferecem a garantia de não lançar exceções). Esse é um bloco de construção crucial para código seguro em relação a exceções.

  Pode parecer razoável considerar que as funções com uma especificação de exceções vazia sejam não lançadoras, mas isso não é necessariamente verdade. Por exemplo, considere a seguinte função:

  ```
  int doSomething( ) throw( );          // observe a especificação de exceção vazia
  ```

  Isso não diz que `doSomething` (faz algo) nunca lançará uma exceção, e sim diz que, *se* `doSomething` lança uma exceção, é um erro sério, e que a função `unexpected` (inesperada) deve ser chamada.* Na verdade, `doSomething` pode não oferecer garantia alguma em relação a exceções. A declaração de uma função (incluindo sua especificação de exceções, se ela tiver uma) não diz se uma função é correta ou portável ou eficiente, e também não diz qual, se existir alguma, garantia de segurança em relação a exceções ela oferece. Todas essas características são determinadas pela implementação da função, e não pela sua declaração.

O código seguro em relação a exceções deve oferecer uma das três garantias acima. Se ele não o faz, não é seguro em relação a exceções. A escolha, então, é determinar que garantia oferecer para cada uma das funções que você escrever. Além de quando você estiver lidando com código legado inseguro em relação a exceções (que discutiremos posteriormente neste item), não oferecer nenhuma garantia em relação às exceções deve ser uma opção apenas se sua equipe de analistas de requisitos identificar a necessidade de que suas aplicações vazem recursos e corrompam as estruturas de dados.

Como regra geral, você deve oferecer a garantia mais forte praticável. Do ponto de vista da segurança em relação a exceções, as funções não lançadoras são excelentes, mas é difícil escalar a parte C de C++ sem chamar funções que possam lançar exceções. Qualquer coisa que use memória dinamicamente alocada (por exemplo, todos os contêineres da STL) geralmente lança uma exceção `bad_alloc` se não puder encontrar memória suficiente para satisfazer a uma requisição (veja o Item 49). Ofereça a garantia de não

* Para mais informações sobre a função `unexpected`, consulte o seu mecanismo de busca favorito ou algum texto mais completo sobre C++. (Você provavelmente terá mais sorte buscando por `set_unexpected`, a função que especifica a função `unexpected`).

lançamento quando puder, mas, para a maioria das funções, a escolha é entre garantias básicas e fortes.

No caso de `changeBackground`, *quase* oferecer a garantia forte não é difícil. Primeiro, modificamos o tipo do membro de dados `bgImage` de `PrettyMenu` de um ponteiro `Image*` (imagem) predefinido e o transformamos em um dos ponteiros espertos de gerenciamento de recursos descritos no Item 13. Para ser franco, essa é uma boa ideia simplesmente com base na prevenção de vazamentos de recursos. O fato de ela nos ajudar a oferecer a garantia forte de segurança em relação a exceções reforça o argumento do Item 13 de que usar objetos (como ponteiros espertos) para gerenciar recursos é fundamental para um bom projeto. No código abaixo, mostro o uso de `tr1::shared_ptr` porque seu comportamento mais intuitivo, quando copiado, é, em geral, preferível ao `auto_ptr`.

Segundo, reordenamos as sentenças em `changeBackground` de forma que não incrementaremos `imageChanges` até que a imagem tenha sido modificada. Como regra geral, é uma boa política não modificar o estado de um objeto para identificar que algo ocorreu até que esse algo realmente tenha ocorrido.

Veja o código resultante:

```
class PrettyMenu {
  ...
  std::tr1::shared_ptr<Image> bgImage;
  ...
};

void PrettyMenu::changeBackground(std::istream& imgSrc)
{
  Lock ml(&mutex);
  bgImage.reset(new Image(imgSrc));          // substitui a parte interna de bgImage
                                             // ponteiro com o resultado da
                                             // expressão "new Image"
  ++imageChanges;
}
```

Observe que não existe mais uma necessidade de apagar manualmente a imagem antiga porque isso está sendo tratado internamente pelo ponteiro esperto. Além disso, a remoção ocorre apenas se a nova imagem for criada com sucesso. Para ser mais preciso, a função `tr1::shared_ptr::reset` será chamada apenas se o seu parâmetro (o resultado de "`new Image(imgSrc)`") for criado com sucesso. Usa-se `delete` apenas dentro da chamada a `reset`, então, se a função nunca for executada, `delete` nunca será usado. Observe também que o uso de um objeto (o `tr1::shared_ptr`) para gerenciar um recurso (a instância de `Image` dinamicamente alocada) mais uma vez diminuiu o tamanho de `changeBackground`.

Como eu disse, essas duas mudanças *quase* são suficientes para permitir que `changeBackground` ofereça a garantia forte de segurança em relação a exceções. Qual é o problema? O parâmetro `imgSrc` (fonte da imagem). Se

o construtor de `Image` lança uma exceção, é possível que o marcador de entrada para o fluxo correspondente tenha sido movido, e esse movimento seria uma mudança no estado visível para o resto do programa. Até que `changeBackground` trate essa questão, ela oferece apenas a garantia básica de segurança em relação a exceções.

Vamos colocar isso de lado, entretanto, e fingir que `changeBackground` realmente oferece a garantia forte. (Estou confiante de que você descobrirá uma maneira de fazer isso, talvez trocando o tipo de seu parâmetro de um `istream` [fluxo de entrada] para o nome do arquivo contendo os dados da imagem.) Existe uma estratégia geral de projeto que, em geral, leva à garantia forte, e é importante conhecê-la. A estratégia é conhecida como "copiar e trocar". Em princípio, é bastante simples. Faça uma cópia do objeto que você quer modificar e depois faça todas as mudanças necessárias na cópia. Se qualquer uma das operações de modificação lançar uma exceção, o objeto original permanecerá intocado. Depois que todas as mudanças forem completadas com sucesso, troque o objeto modificado pelo original em uma operação não lançadora.

Em geral, implementa-se essa ação colocando todos os dados por objeto do objeto "real" em um objeto de implementação separado, dando, então, ao objeto real um ponteiro para seu objeto de implementação. Isso costuma ser conhecido como "idioma pImpl", e o Item 31 o descreve mais detalhadamente. Para `PrettyMenu`, ele geralmente se pareceria com:

```
struct PMImpl {                                  // PMImpl = "Impl. de PrettyMenu";
   std::tr1::shared_ptr<Image> bgImage;          // vê abaixo para saber
   int imageChanges;                             // por que é uma estrutura
};

class PrettyMenu {
   ...

private:
   Mutex mutex;
   std::tr1::shared_ptr<PMImpl> pImpl;
};

void PrettyMenu::changeBackground(std::istream& imgSrc)
{
   using std::swap;                              // vê o Item 25

   Lock ml(&mutex);                              // adquire o mutex

   std::tr1::shared_ptr<PMImpl>                  // copia os dados do objeto
      pNew(new PMImpl(*pImpl));

   pNew->bgImage.reset(new Image(imgSrc));       // modifica a cópia
   ++pNew->imageChanges;

   swap(pImpl, pNew);                            // troca os novos
                                                 // dados no local
}                                                // libera o mutex
```

Nesse exemplo, escolhi fazer de `PMImpl` (implementação de `PrettyMenu`) uma estrutura em vez de uma classe, porque o encapsulamento dos dados

de `PrettyMenu` é garantido pelo fato de `pImpl` ser privado. Tornar `PMImpl` uma classe seria, no mínimo, tão bom quanto, torná-lo uma estrutura, mas menos conveniente. (Também ajuda a manter os puristas da orientação a objetos satisfeitos.) Se desejado, `PMImpl` poderia ser aninhada dentro de `PrettyMenu`, mas questões de empacotamento como essa não dependem da escrita de código seguro em relação a exceções, que é a nossa preocupação aqui.

A estratégia "copiar e trocar" é uma maneira excelente de fazer mudanças do tipo "tudo ou nada" ao estado de um objeto, mas, em geral, ela não garante que a função como um todo seja fortemente segura em relação a exceções. Para ver por que, considere uma abstração de `changeBackground`, chamada `someFunc` (uma função), que usa copiar e trocar, mas que inclui chamadas para duas outras funções, `f1` e `f2`:

```
void someFunc( )
{
  ...                                   // faz uma cópia do estado local
  f1( );
  f2( );
  ...                                   // troca o estado modificado
}
```

Deve ficar claro que, se `f1` ou `f2` forem menos do que fortemente seguras em relação a exceções, será difícil para `someFunc` ser fortemente segura em relação a exceções. Por exemplo, suponhamos que `f1` ofereça apenas a garantia básica. Para que `someFunc` ofereça a garantia forte, ela teria que escrever código para determinar o estado do programa inteiro antes de chamar `f1`, capturar todas as exceções de `f1` e, então, restaurar o estado original.

As coisas não são realmente nada melhores se `f1` e `f2` forem fortemente seguras em relação a exceções. Afinal, se `f1` for executada até seu término, o estado do programa pode ter mudado arbitrariamente; assim, se `f2` lançar uma exceção, o estado do programa não será o mesmo que era quando `someFunc` foi chamada, mesmo que `f2` não tenha mudado nada.

O problema são os efeitos colaterais. Enquanto as funções operarem apenas no estado local (ou seja, `someFunc` afetar apenas o estado do objeto no qual está sendo invocada), é relativamente fácil oferecer a garantia forte. Quando as funções possuem efeitos colaterais em dados não locais, é muito mais difícil. Se um dos efeitos colaterais de chamar `f1` for, por exemplo, a modificação de uma base de dados, será difícil fazer `someFunc` ser fortemente segura em relação a exceções. De modo geral, não é possível desfazer uma modificação em uma base de dados que já tenha sido efetivada (por meio de `commit`); outros clientes da base de dados já podem ter visto o novo estado da base de dados.

Questões como essa podem impedi-lo de oferecer a garantia forte para uma função, mesmo que você quisesse. Outra questão é a eficiência. O cerne da questão copiar e trocar é a ideia de modificar uma cópia dos dados de um

objeto e depois trocar os dados modificados pelo original em uma operação não lançadora. Isso requer que seja feita uma cópia de cada objeto que será modificado, o que custa tempo e espaço que você pode não ser capaz, ou não desejar, de disponibilizar. A garantia forte é altamente desejável, e você deve oferecê-la quando for praticável, mas ela não é sempre praticável.

Quando não for, você precisará oferecer a garantia básica. Na prática, você provavelmente vai descobrir que pode oferecer a garantia forte para algumas funções, mas que o custo, em termos de eficiência ou de complexidade, será inviável para muitas outras. Desde que você tenha feito um esforço razoável para oferecer a garantia forte sempre que praticável, ninguém poderá criticá-lo quando você oferecer apenas a garantia básica. Para muitas funções, a garantia básica é uma escolha perfeitamente razoável.

As coisas são diferentes se você escrever uma função que não oferece garantia alguma de segurança em relação a exceções, porque, nesse caso, é razoável considerar que você seja culpado até que se prove sua inocência. Você *deve* escrever código seguro em relação a exceções, mas pode ter uma defesa convincente. Considere novamente a implementação de `someFunc`, que chama as funções `f1` e `f2`. Suponhamos que `f2` não ofereça segurança em relação às exceções, nem mesmo a garantia básica. Isso significa que, se `f2` emitir uma exceção, é porque o programa pode ter vazado recursos dentro de `f2`. Também que `f2` pode ter estruturas de dados corrompidas, ou seja, os vetores ordenados podem não estar mais ordenados, os objetos que estavam sendo transferidos de uma estrutura de dados para outra podem ter sido perdidos, etc. Não é possível para `someFunc` compensar esses problemas. Se as funções que `someFunc` chama não oferecem garantias sobre a segurança em relação a exceções, `someFunc` também não pode oferecer garantia alguma.

O que nos traz de volta à gravidez. Uma mulher ou está grávida, ou não está. Não é possível estar meio grávida. De maneira similar, ou um sistema de software é seguro em relação a exceções, ou não é. Não existe um sistema parcialmente seguro em relação às exceções. Se um sistema tem uma só função que não seja segura em relação às exceções, o sistema como um todo não é seguro em relação a exceções, porque as chamadas para essa única função podem vazar recursos e corromper as estruturas de dados. Infelizmente, muito código legado em C++ foi escrito sem segurança em relação às exceções em mente, então muitos sistemas atualmente não são seguros em relação a exceções – eles incorporam código que foi escrito de uma maneira não segura em relação a exceções.

Não há motivos para perpetuar esse estado das coisas. Quando estiver escrevendo código novo ou modificando o código existente, pense cuidadosamente em como torná-lo seguro em relação a exceções. Inicie usando objetos para gerenciar recursos. (Mais uma vez, veja o Item 13.) Isso impedirá vazamentos de recursos. Depois, determine quais das três garantias de segurança em relação a exceções é a mais forte que você pode oferecer de maneira prática para cada função que escrever, optando por não oferecer ne-

nhuma garantia apenas se as chamadas a código legado não deixarem outra escolha. Documente suas decisões, tanto para os clientes de suas funções, quanto para os mantenedores futuros. A garantia de segurança em relação a exceções de uma função é uma parte visível de sua interface, então você deve escolhê-la tão deliberadamente quanto escolhe todos os outros aspectos da interface de uma função.

Quarenta anos atrás, os códigos repletos de `goto`s eram considerados uma prática perfeitamente boa. Agora, buscamos escrever fluxos de controle estruturados. Vinte anos atrás, os dados acessíveis globalmente eram considerados uma prática perfeitamente boa. Agora, buscamos encapsular os dados. Dez anos atrás, escrever funções sem pensar sobre o impacto das exceções era considerado uma prática perfeitamente boa. Agora, buscamos escrever código seguro em relação a exceções. O tempo passa. Vivemos. Aprendemos.

Lembretes

» As funções seguras em relação a exceções não vazam recursos e não permitem que as estruturas de dados sejam corrompidas, mesmo quando as exceções forem lançadas. Essas funções oferecem as seguintes garantias: básica, forte ou de não lançamento.

» A garantia forte pode, muitas vezes, ser implementada por meio de "cópia e troca", mas ela não é praticável para todas as funções.

» Uma função pode, normalmente, oferecer uma garantia que não é mais forte do que a mais fraca das garantias das funções que ela chama.

Item 30: Entenda as vantagens e desvantagens da internalização

Funções internalizadas – que ideia *maravilhosa*! Elas se parecem com funções, agem como funções, são até muito melhores do que as macros (veja o Item 2), e você pode chamá-las sem incorrer no custo de uma chamada a função. O que mais poderíamos querer?

Você realmente obtém mais do que pode estar pensando, porque evitar o custo de uma chamada a função é apenas uma parte da história. As otimizações de compilação em geral são projetadas para trechos de código que não possuem chamadas a funções; assim, quando você internaliza uma função, pode possibilitar que os compiladores realizem otimizações específicas de contexto no corpo da função. A maioria dos compiladores nunca realiza essas otimizações em chamadas a funções "externalizadas".

Em programação, entretanto, como na vida, nada é de graça, e as funções internalizadas não são uma exceção. A ideia por trás de uma função internalizada é substituir cada chamada a essa função com seu corpo de código, e não é necessário ter doutorado em estatística para ver que isso provavelmente aumenta o tamanho do código objeto. Em máquinas com memórias

limitadas, muitas internalizações podem gerar programas que são muito grandes para o espaço disponível. Mesmo com memória virtual, o inchaço de código induzido pela internalização pode levar à paginação adicional, a uma taxa reduzida de acessos a cache de instruções e às penalidades de desempenho que acompanham essas coisas.

Por outro lado, se uma função internalizada é *muito* pequena, o código gerado para o corpo da função pode ser menor do que o código gerado para uma chamada à função. Se esse for o caso, internalizar a função pode, na verdade, levar a um código objeto *menor* e a uma taxa de acessos a cache de instruções maior!

Tenha em mente que `inline` é uma *requisição* para os compiladores, e não um comando. A requisição pode ser dada implícita ou explicitamente. A maneira implícita é definir uma função dentro de uma definição de classe:

```
class Person {
public:
  ...
  int age( ) const { return theAge; }          // uma requisição implícita de internalização
  ...                                          // definida em uma definição de classe

private:
  int theAge;
};
```

Essas funções são normalmente funções membro, mas o Item 46 explica que as funções amigas também podem ser definidas dentro de classes. Quando são, também são declaradas implicitamente como internalizadas.

A maneira explícita de declarar uma função internalizada é preceder sua definição com a palavra-chave `inline`. Por exemplo, essa é a maneira pela qual o template `max` (de `<algorithm>`) é frequentemente implementado:

```
template<typename T>                           // uma internalização explícita
inline const T& std::max(const T& a, const T& b)   // request: std::max é
{ return a < b ? b : a; }                      // precedida por "inline"
```

O fato de `max` ser um template traz a observação que tanto as funções internalizadas quanto os templates, em geral, são definidos em arquivos de cabeçalho. Isso faz alguns programadores concluírem que as funções template devem ser internalizadas. Essa conclusão é tanto inválida quanto potencialmente prejudicial, então, vale a pena darmos uma breve olhada nela.

As funções internalizadas devem estar em arquivos de cabeçalho, porque a maioria dos ambientes de construção faz internalização durante a compilação. A fim de substituir a chamada à função com o corpo da função chamada, os compiladores devem saber como é a função. (Alguns ambientes de construção podem internalizar durante a ligação, e alguns – como os ambientes gerenciados baseados na Infraestrutura de Linguagem Comum [CLI – *Common Language Infrastructure*] do .NET – podem, na verdade, internalizar em tempo de execução. Esses ambientes são a exceção, não a

regra. Internalizar, na maioria dos programas em C++, é uma atividade em tempo de compilação.)

Os templates geralmente estão em arquivos de cabeçalhos, pois os compiladores precisam saber como é um template para que o instancie quando for usado. (Mais uma vez, isso não é universal. Alguns ambientes de construção realizam a instanciação de templates durante a ligação. Entretanto, a instanciação em tempo de compilação é mais comum.)

A instanciação de templates independe da internalização. Se você está escrevendo um template e acredita que todas as funções instanciadas a partir dele devam ser internalizadas, declare o template como `inline`; isso é o que se faz com a implementação de `std::max` acima. Mas, se você estiver escrevendo um template para funções sem necessidade de serem internalizadas, evite declará-los internalizado (explícita ou implicitamente). A internalização tem custos, e você não quer arcar com eles sem ter pensado antes. Já mencionamos como a internalização pode causar um inchaço de código (uma consideração especialmente importante para autores de templates – veja o Item 44), mas existem outros custos também, que discutiremos daqui a pouco.

Antes de fazermos isso, vamos terminar a observação de que `inline` é uma requisição que os compiladores podem ignorar. A maioria dos compiladores se recusa a internalizar funções que lhes parecem muito complicadas (por exemplo, aquelas que contêm laços ou que são recursivas), e todas, exceto as chamadas mais triviais às funções virtuais, desafiam a internalização. Essa última observação não deve ser uma surpresa. A palavra-chave `virtual` significa "espere até a execução para descobrir qual função chamar", e `inline` significa "antes da execução, substitua o local da chamada pela função chamada". Se os compiladores não sabem qual função será chamada, você não pode culpá-los por se recusarem a internalizar o corpo da função.

Tudo leva ao seguinte: saber se uma determinada função internalizada é realmente internalizada é algo que depende do ambiente de construção que você está usando – principalmente do compilador. Felizmente, a maioria dos compiladores possui um nível de diagnóstico que resultará em um aviso (veja o Item 53) caso eles não consigam internalizar uma função que você pediu.

Algumas vezes, os compiladores geram um corpo de função para uma função internalizada mesmo quando estão perfeitamente dispostos a internalizar a função. Por exemplo, se o seu programa recebe o endereço de uma função internalizada, os compiladores devem gerar um corpo de função externalizado para ela. Como eles podem criar um ponteiro para uma função que não existe? Juntamente com o fato de que os compiladores não realizam internalizações entre chamadas por meio de ponteiros, isso significa que as chamadas para uma função internalizada podem ser internalizadas ou não, dependendo de como são feitas:

```
inline void f( ) {...}          // considera que os compiladores estarão dispostos
                                // a internalizar as chamadas a f
void (*pf )( ) = f;             // pf aponta para f
...
f( );                           // esta chamada será internalizada, porque é uma chamada "normal"
pf( );                          // esta chamada provavelmente não, porque é feita por
                                // um ponteiro para função
```

O espectro de funções internalizadas "não internalizadas" pode assombrá-lo, mesmo que você nunca use ponteiros para funções, porque os programadores não são necessariamente os únicos que pedem ponteiros para funções. Algumas vezes, os compiladores geram cópias de construtores e destrutores de forma que possam obter ponteiros para essas funções para uso durante a construção e a destruição de objetos em vetores e matrizes.

Na verdade, os construtores e os destrutores são, frequentemente, candidatos piores para a internalização do que um exame casual poderia indicar. Por exemplo, considere o construtor para a classe `Derived` (derivada) abaixo:

```
class Base {
public:
   ...
private:
   std::string bm1, bm2;            // membros base 1 e 2
};
class Derived: public Base {
public:
   Derived( ) { }                   // O construtor de Derived está vazio – ou não está?
   ...
private:
   std::string dm1, dm2, dm3;       // membros derivados 1-3
};
```

Esse construtor parece um candidato excelente para a internalização, uma vez que ele não contém código. Mas as aparências podem enganar.

C++ faz várias garantias sobre as coisas que acontecem quando os objetos são criados e destruídos. Quando você usa `new`, por exemplo, seus objetos criados dinamicamente são automaticamente inicializados por seus construtores, e, quando usa `delete`, os destrutores correspondentes são invocados. Quando você cria um objeto, cada classe-base e cada um dos membros de dados nesse objeto são automaticamente construídos, e o processo inverso ocorre automaticamente na destruição de um objeto. Se uma exceção é lançada durante a construção de um objeto, quaisquer partes do objeto que já tenham sido construídas completamente são destruídas automaticamente. Em todos esses cenários, C++ diz *o que* deve acontecer, mas não *como*. Isso é tarefa para os implementadores de compiladores, mas deve ficar claro que essas coisas não acontecem por si mesmas. Deve existir algum código em seu programa para fazer essas coisas acontecerem, e esse código – o código escrito pelos compiladores e inseridos em seus programas durante a compilação – precisa ir em

algum lugar. Algumas vezes, ele termina nos construtores e nos destrutores, então podemos imaginar implementações gerando código equivalente ao seguinte, para o construtor alegadamente vazio de `Derived`:

```
Derived::Derived( )                          // implementação conceitual do
{                                            // construtor "vazio" de Derived
   Base::Base( );                            // inicializa a parte de base

   try { dm1.std::string::string( ); }       // tenta construir dm1
   catch (...) {                             // se lançar uma exceção,
      Base::~Base( );                        // destrói a parte da classe-base e
      throw;                                 // propaga a exceção
   }
   try { dm2.std::string::string( ); }       // tenta construir dm1
   catch(...) {                              // se lançar uma exceção,
      dm1.std::string::~string( );           // destrói dm1,
      Base::~Base( );                        // destrói a parte da classe-base, e
      throw;                                 // propaga a exceção
   }
   try { dm3.std::string::string( ); }       // constrói dm3
   catch(...) {                              // se lançar uma exceção
      dm2.std::string::~string( );           // destrói dm2
      dm1.std::string::~string( );           // destroy dm1,
      Base::~Base( );                        // destrói a parte da classe-base, e
      throw;                                 // propaga a exceção
   }
}
```

Esse código não representa o que os compiladores reais emitiriam, porque os compiladores reais tratam as exceções de maneiras mais sofisticadas. Mas, mesmo assim, reflete precisamente o comportamento que o construtor "vazio" de `Derived` deve oferecer. Independentemente do grau de sofisticação da implementação das exceções em um compilador, o construtor de `Derived` deve, pelo menos, chamar os construtores para os seus membros de dados e para a sua classe-base, e essas chamadas (as quais podem, elas próprias, ser internalizadas) podem afetar sua atratividade para a internalização.

O mesmo raciocínio se aplica ao construtor de `Base` (classe-base); então, se ele for internalizado, todo o código inserido nele também será inserido no construtor de `Derived` (por meio da chamada do construtor de `Derived` para o construtor de `Base`). E, se o construtor de `string` também for internalizado, o construtor de `Derived` ganhará *cinco cópias* do código dessa função, uma para cada uma das cinco cadeias em um objeto `Derived` (as duas que ele herda, mais as três que ele mesmo declara). Talvez agora esteja claro porque não internalizar `Derived` é uma decisão que não precisa ser pensada. São aplicadas considerações similares ao destrutor de `Derived`, o qual, de uma maneira ou de outra, deve garantir que todos os objetos inicializados pelo construtor de `Derived` sejam destruídos apropriadamente.

Os projetistas de bibliotecas devem avaliar o impacto de declarar funções como internalizadas, porque é impossível fornecer atualizações binárias

para as funções internalizadas visíveis para os clientes em uma biblioteca. Em outras palavras, se `f` é uma função internalizada em uma biblioteca, os clientes da biblioteca compilam o corpo de `f` em suas aplicações. Se um implementador de biblioteca posteriormente decidir mudar `f`, todos os clientes que tenham usado `f` devem ser recompilados. Isso em geral é indesejável. Por outro lado, se `f` é uma função não internalizada, uma modificação a `f` requer apenas que os clientes façam uma religação. Essa é uma atividade substancialmente menos onerosa do que recompilar, se a biblioteca que contém a função for dinamicamente ligada, e pode ser absorvida de uma maneira que é completamente transparente para os clientes.

Para propósitos de desenvolvimento de programas, é importante manter todas essas considerações em mente, mas de um ponto de vista prático durante a codificação, um fato domina todos os outros: a maioria dos depuradores tem problemas com funções internalizadas. Isso pode não ser uma grande revelação. Mas como você configura um ponto de quebra (breakpoint) em uma função que não está lá? Apesar de alguns ambientes de construção gerenciarem o suporte para a depuração de funções internalizadas, muitos ambientes simplesmente desabilitam a internalização em construções de depuração.

Isso leva a uma estratégia lógica para determinar quais funções devem ser declaradas como internalizadas e quais não devem. Inicialmente, não internalize nada, ou pelo menos limite-se a internalizar apenas aquelas funções que devem ser internalizadas (veja o Item 46) ou que são realmente triviais (como `Person::age` – idade de uma pessoa – na página 155). Ao empregar internalizações cautelosamente, você facilita o uso de um depurador, mas também coloca a internalização em seu local apropriado: como uma otimização aplicada manualmente. Não se esqueça da regra empiricamente determinada de 80-20, que diz que um programa típico gasta 80% de seu tempo executando apenas 20% do código. É uma regra importante, porque ela lembra que seu objetivo como desenvolvedor de software é identificar os 20% do código que podem aumentar o desempenho geral do programa. Você pode internalizar e otimizar suas funções até não ter mais como, mas esse é um esforço perdido, a menos que você esteja se focando nas funções *certas*.

Lembretes

» Limite a maioria das internalizações às funções pequenas frequentemente chamadas. Isso facilita a depuração e a facilidade de atualização binária, minimiza um inchaço de código em potencial e maximiza as chances de uma melhor velocidade de execução de seus programas.

» Não declare templates de funções como `inline` apenas porque aparecem em arquivos de cabeçalho.

Item 31: Minimize as dependências de compilação entre os arquivos

Então, você vai para seu programa C++ e faz uma pequena mudança na implementação de sua classe. Não na interface da classe, você pensa, apenas na implementação; apenas nas coisas privadas. Então você reconstrói o programa, pensando que esse exercício deveria levar apenas alguns segundos. Afinal, só uma classe foi modificada. Você clica em *Compilar* ou digita `make` (ou algum equivalente), e fica atônito, e, depois, mortificado, à medida que se dá conta de que todo o *mundo* está sendo recompilado e religado! Você não *odeia* quando isso acontece?

O problema é que C++ não faz um bom trabalho em separar interfaces de implementações. Uma definição de classe especifica não apenas uma interface de classe, mas também um número razoável de detalhes de implementação. Por exemplo:

```
class Person {
public:
   Person(const std::string& name, const Date& birthday,
          const Address& addr);
   std::string name( ) const;
   std::string birthDate( ) const;
   std::string address( ) const;
   ...

private:
   std::string theName;                    // detalhe de implementação
   Date theBirthDate;                      // detalhe de implementação
   Address theAddress;                     // detalhe de implementação
};
```

Aqui, a classe `Person` (pessoa) não pode ser compilada sem o acesso às definições para as classes que a implementação de `Person` utiliza, que são `string` (cadeia de caracteres), `Date` (data) e `Address` (endereço). Em geral, essas definições são fornecidas pelas diretivas `#include`; assim, no arquivo que define a classe `Person`, provavelmente você encontrará algo como:

```
#include <string>
#include "date.h"
#include "address.h"
```

Infelizmente, isso gera uma dependência de compilação entre o arquivo que define `Person` e esses arquivos de cabeçalho. Se quaisquer desses arquivos de cabeçalho forem modificados, ou se qualquer um dos arquivos de cabeçalhos de que eles dependem for modificado, o arquivo que contém a classe `Person` precisará ser recompilado, assim como todos os arquivos que usam `Person`. Essas dependências de compilação em cascata já causaram a morte de muitos projetos.

Você deve ficar imaginando por que C++ insiste em colocar os detalhes de implementação de uma classe na definição da classe. Por exemplo, por que você não pode definir `Person` da seguinte maneira, especificando os detalhes de implementação da classe separadamente?

```
namespace std {
   class string;                          // declaração à frente (incorreta
}                                         // – veja abaixo)

class Date;                               // declaração à frente
class Address;                            // declaração à frente

class Person {
public:
   Person(const std::string& name, const Date& birthday,
          const Address& addr);
   std::string name( ) const;
   std::string birthDate( ) const;
   std::string address( ) const;
   ...
};
```

Se isso fosse possível, os clientes de `Person` teriam de ser recompilados apenas se a interface da classe fosse modificada.

Existem dois problemas com essa ideia. Primeiro, `string` não é uma classe, é uma definição de tipo (`typedef`) para `basic_string<char>`. Por conseguinte, a declaração à frente para `string` é incorreta. A declaração à frente apropriada é substancialmente mais complexa, porque envolve templates adicionais. Isso não importa, porém, porque você não deve tentar declarar manualmente partes da biblioteca padrão. Em vez disso, simplesmente use os `#includes` apropriados e termine a história. É improvável que ocorram gargalos de compilação por causa de cabeçalhos padrão, especialmente se seu ambiente de construção deixar que você tire vantagem de cabeçalhos pré-compilados. Se analisar sintaticamente cabeçalhos padrão for realmente um problema, você pode precisar modificar o projeto de sua interface para não usar as partes da biblioteca padrão que fazem surgir inclusões indesejáveis. A segunda (e mais significativa) dificuldade ao se declarar à frente tem a ver com a necessidade que os compiladores têm de conhecer o tamanho dos objetos durante a compilação. Considere:

```
int main( )
{
   int x;                                 // define um int

   Person p( params );                    // define um objeto Person
   ...
}
```

Quando os compiladores veem a definição para `x`, sabem que precisam alocar espaço suficiente (geralmente na pilha) para manter um `int`. Sem problemas. Cada compilador sabe que um `int` é grande. Quando os compila-

dores veem a definição para `p`, sabem que precisam alocar espaço suficiente para um objeto `Person`, mas como, supostamente, saberão o quão grande é um objeto `Person`? A única maneira por meio da qual podem obter essa informação é consultando a definição da classe, mas, se fosse permitido que uma definição de classe omitisse os detalhes de implementação, como os compiladores saberiam quanto espaço alocar? Essa questão não surge em linguagens como Smalltalk ou Java, porque, quando um objeto é definido nessas linguagens, os compiladores alocam apenas espaço suficiente para um *ponteiro* para um objeto. Ou seja, elas tratam o código acima como se ele tivesse sido escrito da seguinte forma:

```
int main( )
{
   int x;                                  // define um int

   Person *p;                              // define ponteiro para um objeto Person
   ...
}
```

Isso, é claro, é C++ válido, então você mesmo pode entrar no jogo "esconda a implementação do objeto atrás de um ponteiro". Uma maneira de fazer isso para `Person` é separá-la em duas classes, uma oferecendo apenas uma interface, e outra implementando essa interface. Se a classe de implementação for chamada de `PersonImpl` (implementação de `Person`), `Person` seria definida como segue:

```
#include <string>                          // componentes da biblioteca padrão
                                           // não devem ser declarados à frente

#include <memory>                          // para tr1::shared_ptr; veja abaixo

class PersonImpl;                          // declaração à frente da classe
                                           // de implementação de Person

class Date;                                // declarações à frente de classes usadas na
class Address;                             // interface Person

class Person {
public:
   Person(const std::string& name, const Date& birthday,
          const Address& addr);
   std::string name( ) const;
   std::string birthDate( ) const;
   std::string address( ) const;
   ...
private:                                   // ponteiro para a implementação;
   std::tr1::shared_ptr<PersonImpl> pImpl; // veja o Item 13 para obter mais info. sobre
};                                         // std::tr1::shared_ptr
```

Aqui, a classe principal (`Person`) contém como membro de dados nada além de um ponteiro (aqui, um `tr1::shared_ptr` – veja o Item 13) para sua classe de implementação (`PersonImpl`). Em geral, diz-se que esse projeto usa o *idioma pimpl* ("ponteiro para implementação"). Dentro dessas classes, o nome do ponteiro é frequentemente `pImpl`, como mostrado acima.

Com esse projeto, os clientes de `Person` são separados dos detalhes de datas, endereços e pessoas. A implementação dessas classes pode ser modificada à vontade, mas os clientes de `Person` não precisam ser recompilados. Além disso, como são incapazes de ver os detalhes da implementação de `Person`, os clientes provavelmente não escreverão código que, de alguma forma, dependa desses detalhes. Essa é uma separação verdadeira entre a interface e a implementação.

A chave para essa separação é a substituição de dependências em *definições* por dependências em *declarações*. Essa é a essência da minimização das dependências de compilação: torne seus arquivos de cabeçalho autossuficientes sempre que isso for prático, e, quando não for, dependa de declarações em outros arquivos, e não de definições. Todo o resto flui a partir dessa simples estratégia de projeto. Logo:

- **Evite usar objetos quando referências e ponteiros forem suficientes.** Você pode definir referências e ponteiros a um tipo com apenas uma declaração para o tipo. Definir *objetos* de um tipo exige presença da definição do tipo.

- **Dependa de declarações de classes em vez de definições de classe sempre que isso for possível.** Observe que você *nunca* precisa de uma definição de classe para declarar uma função que usa essa classe, nem mesmo se a função passar ou retornar o tipo da classe por valor:

```
class Date;                                  // declaração de classe

Date today( );                               // ok – nenhuma definição
void clearAppointments(Date d);              // de Date é necessária
```

  É claro, passar por valor é geralmente uma má ideia (veja o Item 20), mas você pode acabar fazendo isso por alguma razão, e, ainda assim, não se justifica introduzir dependências de compilação desnecessárias.

  A habilidade de declarar `today` (hoje) e `clearAppointments` (limpar compromissos) sem definir `Date` (data) pode surpreendê-lo, mas isso não é tão curioso quanto parece. Se qualquer um *chamar* essas funções, a definição de `Date` deve ter sido vista antes da chamada. Por que se preocupar em declarar funções que ninguém chama, você deve estar imaginando. Simples. Não é que *ninguém* as chame, é que *nem todo mundo* as chama. Se você tem uma biblioteca contendo dezenas de declarações de funções, é improvável que cada um dos clientes chame cada uma das funções. Ao mover o ônus de fornecer definições de classes a partir de seu arquivo de cabeçalho de *declarações* de funções para os arquivos clientes contendo *chamadas* a funções, você elimina dependências artificiais dos clientes com definições de tipo de que eles não precisam realmente.

- **Forneça arquivos de cabeçalho separados para declarações e definições.** A fim de facilitar as adesões às recomendações mencionadas, os arquivos de cabeçalho precisam vir em pares: um para declarações, o outro para definições. Esses arquivos devem ser mantidos de forma consistente, é claro. Se uma declaração muda em um lugar, ela deve ser mudada em ambos. Como resultado, os clientes de bibliotecas devem sempre incluir um arquivo de declaração em vez de declararem sozinhos algo à frente, e os autores de bibliotecas devem fornecer ambos os arquivos de cabeçalho. Por exemplo, o cliente `Date` que quer declarar `today` e `clearAppointments` não deve declarar manualmente, à frente, `Date` como mostrado acima. Em vez disso, deve incluir o cabeçalho de declarações adequado:

```
#include "datefwd.h"                    // arquivo de cabeçalho declarando
                                        // (mas não definindo) a classe Date

Date today( );                          // como antes
void clearAppointments(Date d);
```

  O nome do arquivo de cabeçalho contendo apenas declarações "`datefwd.h`" baseia-se no cabeçalho `<iosfwd>` da biblioteca padrão de C++ (veja o Item 54). O cabeçalho `<iosfwd>` contém declarações de componentes de fluxo de entrada e saída (`iostream`) cujas definições correspondentes estão em diversos cabeçalhos diferentes, incluindo `<sstream>`, `<streambuf>`, `<fstream>` e `<iostream>`.

  O cabeçalho `<iosfwd>` é instrutivo por outra razão: tornar claro que a recomendação deste item se aplica também a templates e a não templates. Apesar de o Item 30 explicar que, em muitos ambientes de construção, as definições de templates em geral são encontradas em arquivos de cabeçalho, alguns ambientes de construção permitem que as definições de template estejam em arquivos que não são de cabeçalho; então, ainda faz sentido fornecer cabeçalhos contendo somente declarações para templates. O cabeçalho `<iosfwd>` é um desses cabeçalhos.

  C++ também oferece a palavra-chave `export` para permitir a separação de declarações de template de definições de template. Infelizmente, o suporte de compilação para `export` é escasso, e experiências do mundo real com `export` são ainda mais escassas. Como resultado, é muito cedo para dizer que papel `export` terá na programação eficaz em C++.

Classes como `Person`, que empregam o idioma pimpl, são frequentemente chamadas de *classes manipuladoras* (*Handle*). Se você está tentando imaginar como essas classes fazem alguma coisa, uma maneira é encaminhar todas as suas chamadas funções para as classes de implementação correspondentes e fazer essas classes realizarem o trabalho real. Por exemplo, veja como as funções membro de `Person` poderiam ser implementadas:

```
#include "Person.h"                 // estamos implementando a classe Person,
                                    // então devemos incluir sua definição de classe

#include "PersonImpl.h"             // devemos também incluir a definição da classe PersonImpl,
                                    // caso contrário não poderemos chamar
                                    // suas funções membro; observe que
                                    // PersonImpl tem exatamente as mesmas
                                    // funções membro que Person – suas
                                    // interfaces são idênticas

Person::Person(const std::string& name, const Date& birthday,
               const Address& addr)
: pImpl(new PersonImpl(name, birthday, addr))
{}

std::string Person::name( ) const
{
   return pImpl->name( );
}
```

Observe como as chamadas ao construtor de `Person` chamam o construtor de `PersonImpl` (usando `new` – veja o Item 16) e como `Person::name` chama `PersonImpl::name`. Isso é importante. Tornar `Person` uma classe manipuladora não modifica o que `Person` faz, apenas modifica a maneira como faz as coisas.

Uma alternativa à abordagem que usa uma classe manipuladora é tornar `Person` um tipo especial de classe-base abstrata chamada de *classe de Interface*. O objetivo dessa classe é especificar uma interface para classes derivadas (veja o Item 34). Como resultado, em geral, ela não contém membros de dados, não tem construtores, tem um destrutor virtual (veja o Item 7) e um conjunto de funções virtuais puras que especificam a interface.

As classes de Interface são semelhantes às interfaces de Java e do .NET, mas C++ não impõe as restrições em classes de Interface que Java e o .NET impõem às suas interfaces. Nem Java, nem .NET permitem membros de dados ou implementações de funções em interfaces, por exemplo, mas C++ não proíbe nenhuma dessas coisas. A maior flexibilidade de C++ pode ser útil. Como o Item 36 explica, a implementação de funções não virtuais deve ser igual a todas as classes em uma hierarquia, então faz sentido implementar essas funções como parte da classe Interface que as declara.

Uma classe de Interface para `Person` poderia se parecer com o seguinte:

```
class Person {
public:
   virtual ~Person( );

   virtual std::string name( ) const = 0;
   virtual std::string birthDate( ) const = 0;
   virtual std::string address( ) const = 0;
   ...
};
```

Os clientes dessa classe devem programar em termos de ponteiros e de referências a `Person`, porque não é possível instanciar as classes que contêm funções puramente virtuais. (Entretanto, é possível instanciar classes *derivadas* de `Person` – veja abaixo). Tais como os clientes das classes manipuladoras, os clientes das classes de Interface não precisam recompilar, a menos que a interface da classe de Interface seja modificada.

Os clientes de uma classe Interface devem poder criar novos objetos. Em geral, eles fazem isso chamando uma função que desempenha o papel do construtor para as classes derivadas que estão sendo realmente instanciadas. Essas funções normalmente são chamadas de funções fábrica (veja o Item 13), ou *construtores virtuais*. Elas retornam ponteiros (preferencialmente ponteiros espertos – veja o Item 18) para objetos alocados dinamicamente que suportem a interface da classe de Interface. Essas funções são frequentemente declaradas como estáticas (através de `static`) dentro da classe de Interface:

```
class Person {
public:
   ...
   static std::tr1::shared_ptr<Person>          // retorna um tr1::shared_ptr para um novo
      create(const std::string& name,           // objeto Person inicializado com os
             const Date& birthday,              // parâmetros dados; veja o Item 18 para
             const Address& addr);              // saber por que um tr1::shared_ptr é retornado
   ...
};
```

Os clientes as usam tal como:

```
std::string name;
Date dateOfBirth;
Address address;
...
                                                // cria um objeto que suporta a interface Person
std::tr1::shared_ptr<Person> pp(Person::create(name, dateOfBirth, address));
...
std::cout << pp->name( )                        // usa o objeto através da
          << " was born on "                    // interface Person
          << pp->birthDate( )
          << " and now lives at "
          << pp->address( );
...                                             // o objeto é automaticamente
                                                // apagado quando pp sai de
                                                // escopo – veja o Item 13
```

Em algum momento, é claro, devem ser definidas classes concretas que suportam a interface da classe de Interface, e devem ser chamados construtores reais. Tudo ocorre nos bastidores, dentro dos arquivos que contêm as implementações dos construtores virtuais. Por exemplo, a classe de Interface `Person` pode ter uma classe derivada concreta `RealPerson` (pessoa real) que fornece implementações para as funções virtuais que ela herda:

```
class RealPerson: public Person {
public:
   RealPerson(const std::string& name, const Date& birthday,
              const Address& addr)
   : theName(name), theBirthDate(birthday), theAddress(addr)
   { }

   virtual ~RealPerson( ) { }

   std::string name( ) const;                    // as implementações dessas
   std::string birthDate( ) const;               // funções não são mostradas, mas
   std::string address( ) const;                 // são fáceis de imaginar

private:
   std::string theName;
   Date theBirthDate;
   Address theAddress;
};
```

Dada a classe `RealPerson`, é realmente simples escrever `Person::create` (criar pessoa):

```
std::tr1::shared_ptr<Person> Person::create(const std::string& name,
                                            const Date& birthday,
                                            const Address& addr)
{
   return std::tr1::shared_ptr<Person>(new RealPerson(name, birthday,
                                                      addr));
}
```

Uma implementação mais realista de `Person::create` criaria diferentes tipos de objetos de classes derivadas, dependendo, por exemplo, dos valores de parâmetros de função adicionais, de dados lidos de um arquivo ou de uma base de dados, de variáveis de ambiente, etc.

`RealPerson` demonstra um dos dois mecanismos mais comuns para implementar uma classe de Interface: ela herda sua especificação de interface da classe de Interface (`Person`), e depois implementa as funções na interface. Uma segunda maneira de implementar uma classe de interface envolve herança múltipla, um tópico explorado no Item 40.

As classes manipuladoras e as classes de Interface separam interfaces de implementações, reduzindo, dessa forma, as dependências de implementação entre arquivos. Desconfiado como você é, sei que está esperando para fazer a seguinte pergunta: "Qual é o custo toda essa mágica?" A resposta é bastante comum em ciência da computação: custa alguma velocidade em tempo de execução, mais alguma memória adicional por objeto.

No caso das classes manipuladoras, as funções membro devem passar pelo ponteiro de implementação para obter os dados do objeto. Isso adiciona um nível de indireção por acesso, e você deve adicionar o tamanho desse ponteiro de implementação à quantidade de memória necessária para armazenar cada objeto. Por fim, o ponteiro de implementação precisa ser inicializado (nos construtores da classe manipuladora) para apontar para um objeto de implementação alocado dinamicamente, então você incorre na sobrecarga

inerente da alocação dinâmica de memória (e subsequente liberação) e na possibilidade de encontrar exceções de falta de memória (`bad_alloc`).

Para classes de Interface, cada chamada à função é virtual, então você paga o custo de um salto indireto cada vez que faz uma chamada à função (veja o Item 7). Além disso, os objetos derivados da classe de Interface devem conter um ponteiro de tabela virtual (mais uma vez, veja o Item 7). Esse ponteiro pode aumentar a quantidade de memória necessária para armazenar um objeto, dependendo se a classe de Interface é a fonte exclusiva de funções virtuais para o objeto.

Por fim, nem as classes manipuladoras, nem as classes de Interface podem fazer muito uso de funções internalizadas. O Item 30 explica por que os corpos de funções em geral devem estar nos arquivos de cabeçalho para serem internalizados, mas as classes manipuladoras e de Interface são projetadas especificamente para ocultar detalhes de implementação, como corpos de funções.

Seria um erro sério, entretanto, não considerar as classes manipuladoras e as classes de Interface simplesmente porque possuem um custo associado a elas. As funções virtuais também possuem um custo, e você não quer esquecê-las, certo? (Se quiser, você está lendo o livro errado). Em vez disso, pense em usar essas técnicas de uma maneira revolucionária. Use as classes manipuladoras e as classes de Interface durante o desenvolvimento para minimizar o impacto nos clientes quando as implementações mudam. Substitua as classes manipuladoras e as classes de Interface por classes concretas para uso em produção quando puder ser mostrado que a diferença em termos de velocidade ou de tamanho é significativa o suficiente para justificar o aumento do acoplamento entre as classes.

Lembretes

» A ideia geral por trás da minimização das dependências de compilação é depender de declarações e não depender de definições. Duas abordagens baseadas nessa ideia são as classes manipuladoras e as classes de Interface.

» Os arquivos de cabeçalho de bibliotecas devem existir apenas nas formas completa e de somente declaração. Isso se aplica independentemente de os templates estarem, ou não, envolvidos.

CAPÍTULO 6

HERANÇA E PROJETO ORIENTADO A OBJETOS

A programação orientada a objetos (POO) já é moda há quase duas décadas; portanto, é provável que você tenha alguma noção de herança, derivação e funções virtuais. Mesmo que você programe apenas em C, certamente não deve ter escapado ao frenesi da POO.

A POO em C++ talvez seja um pouco diferente do que você está acostumado. A herança pode ser simples ou múltipla, e cada ligação de herança pode ser pública, protegida ou privada. Cada ligação também pode ser virtual ou não virtual. Existem também as opções das funções membro. Virtual? Não virtual? Puramente virtual? E as interações com outros recursos de linguagens. Como os valores padrão de parâmetros interagem com funções virtuais? Como a herança afeta as regras de busca de nomes de C++? E em relação às opções de projeto? Se o comportamento de uma classe precisa ser modificável, uma função virtual é a melhor maneira de fazer isso?

Este capítulo discute todos esses aspectos. Além disso, explico o que os diferentes recursos de C++ realmente *significam* – o que você está *expressando* de fato quando usa uma construção específica. Por exemplo, a herança pública significa um relacionamento do tipo "é um(a)"; se você tentar fazê-la significar outra coisa, terá problemas. De maneira similar, uma função virtual significa que "a interface deve ser herdada", enquanto uma função não virtual significa que "tanto a interface quanto a implementação devem ser herdadas". Não conseguir distinguir entre esses significados tem causado consideráveis problemas aos programadores de C++.

Se você entender os significados dos vários recursos de C++, verá que sua visão sobre POO mudará. Em vez de ser um exercício de diferenciar recursos de linguagem, se tornará uma questão de determinar o que você quer dizer sobre o seu sistema de software. E, uma vez que você sabe o que quer dizer, a tradução para C++ não é terrivelmente exigente.

Item 32: Certifique-se de que a herança pública modele um relacionamento "é um(a)"

Em seu livro *Some Must Watch While Some Must Sleep* (W. H. Freeman e Company, 1974), William Dement relata a história de sua tentativa de fixar na mente de seus alunos as lições mais importantes de seu curso. Afirma-se, ele dizia em aula, que, na Inglaterra, a criança média em idade escolar lembra-se pouco mais de história além do fato de a Batalha de Hastings ter sido em 1066. Se uma criança lembra pouco mais que isso, Dement enfatizava, ela se lembra da data 1066. Para os alunos na *sua* disciplina, Dement continuava, existiam apenas algumas mensagens centrais, incluindo, de um modo interessante, o fato de que as pílulas para dormir causam insônia. Ele implorava para que seus alunos lembrassem desses poucos fatos cruciais, mesmo que esquecessem todo o resto que foi discutido na disciplina, e voltava a esses preceitos fundamentais repetidamente durante o curso.

No final, a última questão no exame final foi: "Escreva uma coisa do curso que você certamente se lembrará pelo resto da vida". Quando Dement corrigiu os exames, ficou atônito. Praticamente todo o mundo escreveu "1066".

Então, é com grande apreensão que digo a você, agora, que a regra mais importante na programação orientada a objetos com C++ é a seguinte: herança pública significa "é um(a)". Grave essa regra em sua memória.

Se você escrever que a classe `D` ("Derivada") herda publicamente da classe `B` ("Base"), está dizendo aos compiladores C++ (assim como aos leitores humanos de seu código) que todo objeto do tipo `D` também é um objeto do tipo `B`, mas *não vice-versa*. Você está dizendo que `B` representa um conceito mais geral que `D`, que `D` representa um conceito mais especializado que `B`. Você está se certificando que, em qualquer lugar que um objeto do tipo `B` puder ser usado, também é possível usar um objeto do tipo `D`, porque todo objeto do tipo `D` *é* um objeto do tipo `B`. Por outro lado, se você precisa de um objeto do tipo `D`, um objeto do tipo `B` não servirá: todo `D` é um `B`, mas não vice-versa.

C++ força essa interpretação de herança pública. Considere o seguinte exemplo:

```
class Person { ... };

class Student: public Person { ... };
```

Sabemos de nossa experiência do dia a dia que todo aluno é uma pessoa, mas nem toda pessoa é um aluno. É exatamente isso que essa hierarquia afirma. Esperamos que tudo que seja verdade para uma pessoa – por exemplo, que ela tenha uma data de nascimento – também seja verdade para um aluno. Não esperamos que tudo o que seja verdade para um aluno – que ele esteja matriculado em uma determinada escola, por exemplo – seja verdade

para pessoas de um modo geral. A noção de "pessoa" é mais geral que a de "aluno"; um aluno é um tipo especializado de pessoa.

Dentro do mundo de C++, qualquer função que espera um argumento do tipo `Person` (pessoa – ou um ponteiro ou referência a `Person`) também receberá um objeto `Student` (aluno – ou um ponteiro ou referência a `Student`):

```
void eat(const Person& p);          // todo o mundo pode comer

void study(const Student& s);       // apenas alunos estudam

Person p;                           // p é uma pessoa (Person)
Student s;                          // s é um aluno (Student)

eat(p);                             // ok, p é uma pessoa (Person)

eat(s);                             // ok, s é um aluno (Student),
                                    // e um aluno (Student) é uma pessoa (Person)

study(s);                           // ótimo

study(p);                           // erro! p não é um aluno (Student)
```

Isso é verdade apenas para a herança *pública*. C++ se comportará como descrevi apenas se `Student` for publicamente derivada de `Person`. A herança privada significa algo inteiramente diferente (veja o Item 39), e a herança protegida é algo cujo significado me foge à compreensão até hoje.

A equivalência da herança pública e "é um(a)" parece simples, mas algumas vezes sua intuição pode enganar. Por exemplo, é fato que um pinguim é um pássaro, e é fato que os pássaros voam. Se, inocentemente tentarmos expressar isso em C++, nossos esforços nos levam a:

```
class Bird {
public:
   virtual void fly( );             // pássaros podem voar

   ...
};

class Penguin: public Bird {        // pinguins são pássaros

   ...
};
```

De repente, estamos encrencados, porque essa hierarquia diz que os pinguins podem voar, o que sabemos que não é verdade. O que aconteceu?

Nesse caso, somos vítimas de uma linguagem imprecisa: a língua falada. Quando dizemos que os pássaros podem voar, não queremos dizer que *todos* os tipos de pássaros podem voar; apenas que, de um modo geral, os pássaros possuem a habilidade de voar. Se fôssemos mais precisos, reconheceríamos que existem diversos tipos de pássaros não voadores, e chegaríamos à seguinte hierarquia, que modela a realidade muito melhor:

```
class Bird {
   ...                                          // nenhuma função voar (fly) é declarada
};
class FlyingBird: public Bird {
public:
   virtual void fly( );
   ...
};
class Penguin: public Bird {
   ...                                          // nenhuma função voar (fly) é declarada
};
```

Essa hierarquia é muito mais fiel ao que realmente sabemos do que era o projeto original.

Mesmo assim, não terminamos com essas questões, porque, para alguns sistemas de software, pode não existir a necessidade de distinguir entre pássaros que voam e pássaros que não voam. Se sua aplicação tem a ver com bicos e asas e nada a ver com voar, a hierarquia de duas classes seria bastante satisfatória. Essa é uma reflexão simples sobre o fato de que não existe um projeto ideal para todos os aplicativos de software. O melhor projeto depende daquilo que se espera que o sistema faça, tanto agora quanto no futuro. Se sua aplicação não possui conhecimento de voo e não se espera que um dia precise disso, não diferenciar pássaros que voam e que não voam pode ser uma decisão de projeto perfeitamente válida. Na verdade, ela pode ser preferível a um projeto que diferencie os dois, porque tal distinção não existe no mundo que você está tentando modelar.

Existe outra escola de pensamento sobre como tratar o que chamo de problema "todos os pássaros podem voar, os pinguins são pássaros, pinguins não podem voar, xi". Ela propõe a redefinição da função `fly` (voar) para pinguins de forma que seja gerado um erro em tempo de execução:

```
void error(const std::string& msg);                 // definida em outro lugar
class Penguin: public Bird {
public:
   virtual void fly( ) { error("Attempt to make a penguin fly!"); }
   ...
};
```

É importante reconhecer que essa abordagem diz algo diferente do que você possa estar pensando. Ela *não* diz "os pinguins não podem voar". Ela diz que "os pinguins podem voar, mas é um erro para eles tentarem fazê-lo".

Como saber qual é a diferença? A diferença está no momento em que o erro é detectado. A definição "pinguins não podem voar" pode ser verificada pelos compiladores, mas a violação da regra "os pinguins podem voar, mas é um erro para eles tentarem fazê-lo" pode ser detectada apenas em tempo de execução.

Para expressar a restrição "pinguins não podem voar – *ponto*", você se certifica de que nenhuma função de voo seja definida para objetos da classe `Penguin` (Pinguim):

```
class Bird {
   ...                                          // nenhuma função fly é declarada
};
class Penguin: public Bird {
   ...                                          // nenhuma função fly é declarada
};
```

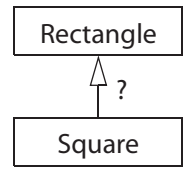
Se você tentar, agora, fazer um pinguim voar, os compiladores o repreenderão por sua transgressão:

```
Penguin p;
p.fly( );                                       // erro!
```

Essa abordagem é bastante diferente do comportamento que você obtém se adotar a abordagem que gera erros em tempo de execução. Com essa metodologia, os compiladores não dirão uma palavra se quer sobre a chamada a `p.fly`. O Item 18 explica que boas interfaces impedem que código inválido seja compilado, então você deve preferir o projeto que é rejeitado durante a compilação que os pinguins tentem voar do que aquela que detecta os voos apenas em tempo de execução.

Talvez você admita que sua intuição ornitológica deixe a desejar, mas pode confiar em seu conhecimento de geometria elementar, certo? Quer dizer, quão complicados podem ser retângulos e quadrados?

Bem, responda à seguinte questão simples: a classe que representa quadrados (`Square`) deve herdar publicamente da classe que representa retângulos (`Rectangle`)?



"Dã!", você diria. "É claro que deve! Todo mundo sabe que um quadrado é um retângulo, mas o contrário geralmente não se aplica". Isso é suficiente, pelo menos na escola, mas acho que não estamos mais na escola.

Considere o código a seguir:

```
class Rectangle {
public:
  virtual void setHeight(int newHeight);
  virtual void setWidth(int newWidth);

  virtual int height( ) const;                    // retorna os valores atuais
  virtual int width( ) const;
  ...
};

void makeBigger(Rectangle& r)                     // função para aumentar a área de r
{
  int oldHeight = r.height( );

  r.setWidth(r.width( ) + 10);                    // adiciona 10 à largura de r

  assert(r.height( ) == oldHeight);               // garanta que a altura de r
}                                                 // permaneça sem ser modificada
```

Claramente, a afirmação nunca deve falhar. A função `makeBigger` (aumentar) apenas modifica a largura de `r`. Sua altura nunca é modificada.

Agora, considere o seguinte código, que usa herança pública para permitir que quadrados sejam tratados como retângulos:

```
class Square: public Rectangle { ... };

Square s;

...

assert(s.width( ) == s.height( ));          // isso deve ser verdade para todos os quadrados

makeBigger(s);                              // por herança, s é um retângulo
                                            // então podemos aumentar sua área

assert(s.width( ) == s.height( ));          // isso ainda pode ser verdade
                                            // para todos os quadrados
```

Também fica claro que essa segunda asserção nunca deve falhar. Por definição, a largura de um quadrado é a mesma que sua altura.

Mas agora temos um problema. Como podemos reconciliar as seguintes afirmações?

- Antes de chamarmos `makeBigger`, a altura de `s` é a mesma que sua largura;
- Dentro de `makeBigger`, a largura de `s` é modificada, mas sua altura não;

- Após retornar de `makeBigger`, a altura de `s` é novamente a mesma de sua largura. (Observe que `s` é passado para `makeBigger` por referência, então `makeBigger` modifica `s`, não uma cópia de `s`.)

Então?

Bem-vindo ao mundo maravilhoso da herança pública, em que os instintos que você desenvolveu em outras áreas de estudo – incluindo a matemática – podem não servir tão bem quanto você esperaria. A dificuldade fundamental, nesse caso, é que algo aplicável a um retângulo (sua largura pode ser modificada independentemente de sua altura) não é aplicável a um quadrado (sua largura e altura devem ser iguais). Mas a herança pública garante que tudo o que se aplica a objetos da classe-base – *tudo!* – também se aplique aos objetos da classe derivada. No caso dos retângulos e quadrados (bem como no exemplo envolvendo conjuntos e listas no Item 38), essa asserção não é garantidamente verdadeira, então usar herança pública para modelar seu relacionamento é simplesmente incorreto. Os compiladores deixarão que você o faça, mas, como vimos agora, não existe uma garantia de que o código se comportará de maneira apropriada. Como todo programador deve aprender (alguns com mais frequência do que outros), apenas porque o código é compilado não significa que ele funcione.

Não tenha medo de a intuição de software que você desenvolveu ao longo dos anos falhar quando você usar a abordagem de projeto orientado a objetos. Esse conhecimento ainda é valioso, mas, agora que você adicionou a herança ao seu arsenal de alternativas de projeto, terá que atualizar sua intuição com as novas ideias aprendidas para orientá-lo na aplicação apropriada da herança. Em tempo, a noção de fazer Pinguim (`Penguin`) herdar de Pássaro (`Bird`) ou de Quadrado (`Square`) herdar de Retângulo (`Rectangle`) provocará o mesmo sentimento estranho que você provavelmente sente quando alguém lhe mostra uma função com diversas páginas. É, *possivelmente*, a maneira correta de abordar as coisas, só não é muito provável.

O relacionamento "é um(a)" não é o único que pode existir entre classes. Os dois outros relacionamentos interclasses comuns são "tem um(a)" e "é implementado em termos de". Esses relacionamentos são vistos nos Itens 38 e 39. Não é incomum que os projetos C++ tenham problemas porque um desses outros relacionamentos importantes foi incorretamente modelado como um relacionamento "é um(a)", então você deve ter certeza que entende as diferenças entre esses relacionamentos e que conhece como cada um deles é melhor modelado em C++.

Lembretes

» A herança pública significa "é um(a)". Tudo que se aplica para as classes-base também deve se aplicar às classes derivadas, porque cada objeto da classe derivada *é* um objeto da classe-base.

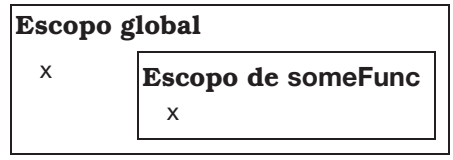
Item 33: Evite ocultar nomes herdados

Shakespeare tinha uma queda por nomes. "O que há num simples nome?", ele perguntou. "O que chamamos rosa com outro nome não teria igual perfume?". O bardo também escreveu, "o que me subtrai o meu bom nome... a mim me deixa na miséria". Certo. Isso nos traz aos nomes herdados em C++.

A questão realmente não tem nada a ver com herança. Tem a ver com escopos. Todos nós sabemos que em um código como o seguinte,

```
int x;                                  // variável global
void someFunc( )
{
double x;                               // variável local

std::cin >> x;                          // lê um novo valor para o x local
}
```

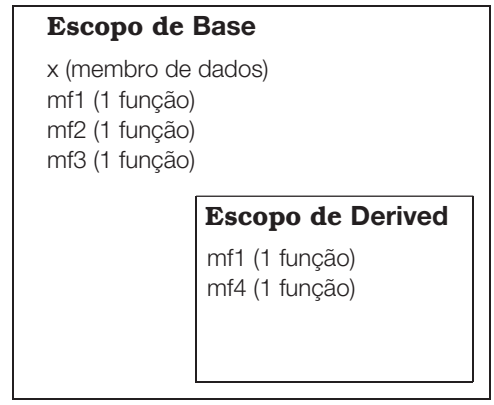
a sentença lendo `x` refere-se à variável local `x` em vez de à variável global `x`, porque os nomes em escopos internos ocultam nomes em escopos externos. Podemos visualizar a situação de escopo da seguinte forma:



Quando os compiladores estão no escopo de `someFunc` (uma função) e encontram o nome `x`, procuram no escopo local para ver se existe algo com aquele nome, pois, caso exista, eles nunca examinam outro escopo. Nesse caso, o `x` de `someFunc` é do tipo `double` e a variável global `x` é do tipo `int`, mas isso não importa. As regras de ocultação de nomes fazem justamente isso: ocultam *nomes*. O fato de os nomes corresponderem ao mesmo tipo ou a tipos diferentes é imaterial. Nesse caso, um `double` chamado `x` oculta um `int` chamado `x`.

Aqui entra a herança. Sabemos que, quando estamos dentro de uma função membro de uma classe derivada e nos referimos a algo na classe-base (por exemplo, uma função membro, uma definição de tipo, ou um membro de dados), os compiladores podem achar o item para o qual estamos nos referindo, porque as classes derivadas herdam as coisas declaradas nas classes-base. A maneira pela qual isso realmente funciona é que o escopo de uma classe derivada é aninhado dentro do escopo de sua classe-base. Por exemplo:

```
class Base {
private:
   int x;

public:
   virtual void mf1( ) = 0;
   virtual void mf2( );
   void mf3( );
   ...
};

class Derived: public Base {
public:
   virtual void mf1( );
   void mf4( );
   ...
};
```



Esse exemplo inclui um misto de nomes públicos e privados, bem como os nomes de membros de dados e de funções membro. As funções membro são puramente virtuais, virtuais simples (impuras) e não virtuais. Esse exemplo visa a enfatizar que estamos falando de *nomes*. O exemplo também poderia ter incluído nomes dos tipos, como, por exemplo, enumerações, classes aninhadas e definições de tipo. A única coisa que importa nessa discussão é que eles são nomes. *Do que* eles são nome é irrelevante. O exemplo usa herança simples, mas, uma vez que você tenha entendido o que está acontecendo na herança simples, o comportamento de C++ na herança múltipla é fácil de ser antecipado.

Suponhamos que `mf4` (função membro 4) na classe derivada seja implementada, em parte, como:

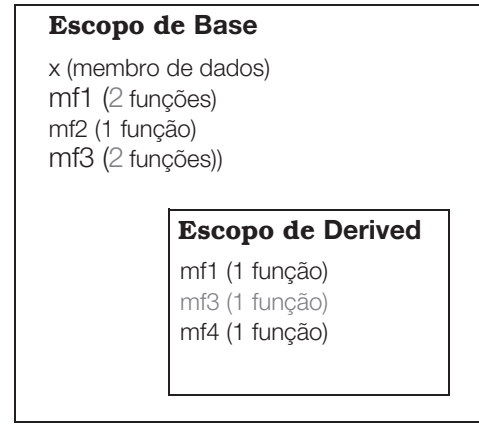
```
void Derived::mf4( )
{
   ...
   mf2( );
   ...
}
```

Quando os compiladores veem o uso do nome `mf2` (função membro 2) aqui, precisam descobrir a que ele se refere. Eles fazem isso procurando nos escopos por uma declaração de algo chamado `mf2`. Primeiro, eles olham no escopo local (aquele de `mf4`), mas não encontram declaração alguma para algo chamado `mf2`. Eles então buscam pelo escopo externo ao escopo atual, aquele da classe derivada (`Derived`). Eles continuam sem achar nada chamado `mf2`, então se movem para o próximo escopo externo, o da classe-base. Lá eles acham algo chamado `mf2`, então a busca para. Se não existisse `mf2` em `Base`, a busca continuaria, primeiro no(s) espaço(s) de nomes contendo `Base`, se houvesse algum, e, por fim, no escopo global.

O processo que acabei de descrever é preciso, mas não é uma descrição completa de como os nomes são encontrados em C++. Nosso objetivo, no entanto, não é conhecer o suficiente sobre resolução de nomes para escrever um compilador, mas conhecer o suficiente para evitar surpresas desagradáveis, e, para essa tarefa, já temos bastante informação.

Considere o exemplo anterior novamente, com a diferença de que, desta vez, vamos sobrecarregar `mf1` (função membro 1) e `mf3` (função membro 3) e adicionar uma versão de `mf3` a `Derived`. (Como explica o Item 36, a sobrecarga de `mf3` por `Derived` – uma função não virtual herdada – coloca esse projeto instantaneamente sob suspeita, mas, para entender a visibilidade de nomes sob herança, vamos ignorar isso.)

```
class Base {
private:
   int x;

public:
   virtual void mf1( ) = 0;
   virtual void mf1(int);

   virtual void mf2( );

   void mf3( );
   void mf3(double);
   ...
};

class Derived: public Base {
public:
   virtual void mf1( );
   void mf3( );
   void mf4( );
   ...
};
```



Esse código leva a um comportamento que surpreende todo o programador de C++ que o encontra pela primeira vez. A regra de ocultação de nome baseada em escopo não foi modificada, então *todas* as funções chamadas `mf1` e `mf3` na classe-base são ocultadas pelas funções chamadas `mf1` e `mf3` na classe derivada. Da perspectiva de resolução de nomes, `Base::mf1` e `Base::mf3` não são mais herdadas por `Derived`!

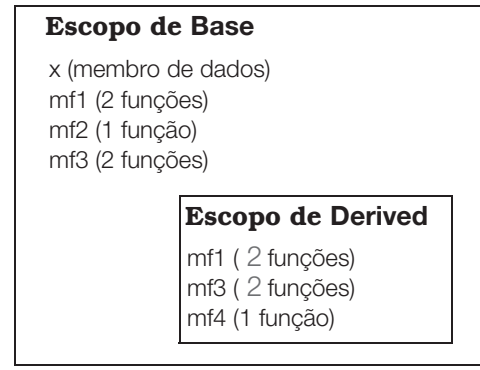
```
Derived d;
int x;
...

d.mf1( );                    // muito bem, chama Derived::mf1
d.mf1(x);                    // erro! Derived::mf1 oculta Base::mf1

d.mf2( );                    // ótimo, chama Base::mf2

d.mf3( );                    // ótimo, chama Derived::mf3
d.mf3(x);                    // erro! Derived::mf3 oculta Base::mf3
```

Como você pode ver, isso se aplica até mesmo se as funções nas classes-base e derivada recebem tipos de parâmetros diferentes, e também se aplica independentemente de as funções serem virtuais ou não virtuais. Da mesma maneira que, no início deste item, o `x` do tipo `double` na função `someFunc` ocultava o `x` do tipo `int` no escopo global, aqui a função `mf3` em `Derived` oculta uma função da classe-`base` chamada `mf3` que possui um tipo diferente.

A razão por trás desse comportamento é que ele impede que você herde acidentalmente sobrecargas de classes-base distantes quando você cria uma nova classe derivada em uma biblioteca ou em uma estrutura de aplicação. Infelizmente, em geral você *quer* herdar as sobrecargas. Na verdade, se você está usando herança pública e não quer herdar as sobrecargas, está violando o relacionamento "é um(a)" entre as classes-base e derivada, algo que o Item 32 explica ser fundamental em relação à herança pública. Sendo esse o caso, você quase sempre deve sobrescrever o comportamento padrão de C++ de ocultação de nomes herdados.

Isso é feito por meio de *declarações de uso* (`using`):

```
class Base {
private:
   int x;

public:
   virtual void mf1( ) = 0;
   virtual void mf1(int);

   virtual void mf2( );

   void mf3( );
   void mf3(double);
   ...
};

class Derived: public Base {
public:
   using Base::mf1;
   using Base::mf3;

   virtual void mf1( );
   void mf3( );
   void mf4( );
   ...
};
```




```
// torna tudo o que for chamado mf1 e mf3 em Base
// visível (e pública) no escopo de Derived
```

Agora, a herança funcionará como o esperado:

```
Derived d;
int x;

...
d.mf1( );                          // ainda bom, ainda chama Derived::mf1
d.mf1(x);                          // ok agora, chama Base::mf1

d.mf2( );                          // ainda bom, ainda chama Base::mf2

d.mf3( );                          // ainda bom, chama Derived::mf3
d.mf3(x);                          // ok agora, chama Base::mf3
```

Isso significa que, se você herda de uma classe-base com funções sobrecarregadas e quer redefinir ou sobrescrever apenas algumas delas, você precisa incluir uma declaração `using` para cada nome que, de outra forma, estaria ocultando. Se não fizer isso, alguns dos nomes que você gostaria de herdar serão ocultados.

É concebível que, algumas vezes, você não queira herdar todas as funções de suas classes-base. No entanto, sob herança pública, isso nunca deve ser o caso, porque, novamente, isso viola o relacionamento "é um(a)" entre as classes-base e as classes derivadas. (Esse é o motivo pelo qual as declarações `using` acima estão na parte pública da classe derivada: os nomes que são públicos na classe-base também devem ser públicos em uma classe derivada publicamente.) Sob herança privada (veja o Item 39), entretanto, pode fazer sentido. Por exemplo, suponhamos que `Derived` herde privadamente de `Base`, e a única versão de `mf1` que `Derived` quer herdar é aquela que não recebe parâmetro algum. Uma declaração `using` não servirá aqui, porque ela torna *todas* as funções herdadas com um dado nome visíveis na classe derivada. Não, esse é um caso para uma técnica diferente, que, nesse caso, é uma simples função delegadora:

```
class Base {
public:
  virtual void mf1( ) = 0;
  virtual void mf1(int);

  ...                              // como antes
};

class Derived: private Base {
public:
  virtual void mf1( )              // função delegadora; implicitamente
  { Base::mf1( ); }                // internalizada (veja o Item 30)
  ...
};

...

Derived d;
int x;

d.mf1( );                          // bem, chama Derived::mf1
d.mf1(x);                          // erro! Base::mf1( ) é ocultada
```

Outro uso para funções delegadoras internalizadas é tentar corrigir compiladores antigos que (incorretamente) não suportam o uso de declarações `using` para importar nomes herdados no escopo de uma classe derivada.

Essa é a história completa sobre herança e ocultação de nomes, mas, quando a herança é combinada com templates, surge uma forma inteiramente diferente de questões relacionadas a "nomes herdados são ocultados". Para saber todos os detalhes demarcados por sinais de menor e maior, veja o Item 43.

Lembretes

» Os nomes em classes derivadas ocultam os nomes em classes-base. Sob herança pública, isso nunca é desejável.

» Para tornar os nomes ocultos visíveis novamente, empregue declarações `using` ou funções delegadoras.

Item 34: Diferencie a herança de interface da herança de implementação

A noção aparentemente direta de herança (pública) compreende, após um exame mais detalhado, duas partes separáveis: a herança de interfaces de funções e a herança de implementações de funções. A diferença entre esses dois tipos de herança corresponde exatamente à diferença, discutida na Introdução deste livro, entre as declarações de funções e as definições de funções.

Como projetista de classes, às vezes você precisa que as classes derivadas herdem apenas a interface (declaração) de uma função membro. Às vezes, você precisa que as classes derivadas herdem tanto a interface quanto a implementação de uma função, mas precisa permitir que elas sobrescrevam a implementação que herdam. E, às vezes, você precisa que as classes derivadas herdem a interface e a implementação de uma função sem lhes permitir sobrescrever qualquer coisa.

Para entender melhor as diferenças entre essas opções, considere uma hierarquia de classes para representar formas geométricas em uma aplicação gráfica:

```
class Shape {
public:
   virtual void draw( ) const = 0;

   virtual void error(const std::string& msg);

   int objectID( ) const;
   ...
};

class Rectangle: public Shape { ... };

class Ellipse: public Shape { ... };
```

`Shape` (forma) é uma classe abstrata; sua função puramente virtual `draw` (desenhar) a marca como tal. Como resultado, os clientes não podem criar instâncias da classe `Shape`, apenas de classes derivadas dela. Independentemente disso, `Shape` imprime uma forte influência em todas as classes que herdam (publicamente) dela, porque

- As *interfaces* de funções membro *são sempre herdadas*. Conforme explicado no Item 32, a herança pública significa "é um(a)", então qualquer coisa que seja verdade para uma classe-base também deve ser verdade para suas classes derivadas. Logo, se uma função se aplica a uma classe, ela deve também se aplicar às suas classes derivadas.

Três funções são declaradas na classe `Shape`. A primeira, `draw`, desenha o objeto atual em uma visualização implícita. A segunda, `error` (erro), é chamada por funções membro se elas precisam informar um erro. A terceira, `objectID` (identidade do objeto), retorna um identificador inteiro único para o objeto atual. Cada função é declarada de uma maneira diferente: `draw` é uma função virtual pura; `error` é uma função virtual (impura?) simples; e `objectID` é uma função não virtual. Quais são as implicações dessas declarações diferentes?

Considere primeiro a função virtual pura `draw`:

```
class Shape {
public:
   virtual void draw( ) const = 0;
   ...
};
```

Os recursos mais proeminentes das funções virtuais puras é que elas *devem* ser redeclaradas por qualquer classe concreta que as herda, e geralmente não têm definições em classes abstratas. Coloque essas duas características juntas e você descobre que

- O propósito de declarar uma função virtual pura é fazer as classes derivadas herdarem *apenas a interface* de uma função.

Isso faz perfeito sentido para a função `Shape::draw`, porque é razoável exigir que todos os objetos `Shape` sejam desenháveis, mas a classe `Shape` não consegue fornecer nenhuma implementação padrão razoável para essa função. O algoritmo para desenhar uma elipse é muito diferente daquele para desenhar um retângulo, por exemplo. A declaração de `Shape::draw` diz aos projetistas das classes derivadas concretas: "Você deve fornecer uma função `draw`, mas não tenho ideia de como você vai implementá-la".

A propósito, *é* possível fornecer uma definição para uma função virtual pura. Ou seja, você pode fornecer uma implementação para `Shape::draw`, e C++ não reclamará, mas a única maneira de chamá-la seria por meio da qualificação da chamada com o nome da classe:

```
Shape *ps = new Shape;                  // erro! Shape é abstrata

Shape *ps1 = new Rectangle;             // ótimo
ps1->draw( );                           // chama Rectangle::draw

Shape *ps2 = new Ellipse;               // ótimo
ps2->draw( );                           // chama Ellipse::draw

ps1->Shape::draw( );                    // chama Shape::draw

ps2->Shape::draw( );                    // chama Shape::draw
```

Além de ajudá-lo a impressionar seus colegas programadores, a utilidade desse recurso é, geralmente, limitada. Como você verá abaixo, entretanto, ele pode ser empregado como mecanismo para fornecer uma implementação padrão mais segura do que a usual para funções virtuais (impuras) simples.

A história por trás de funções virtuais simples é um pouco diferente daquela das virtuais puras. Normalmente, as classes derivadas herdam a interface da função, mas as funções virtuais simples fornecem uma implementação do que as classes derivadas podem sobrescrever. Se você pensar sobre o assunto por um minuto, se dará conta de que

- O propósito de declarar uma função virtual simples é fazer as classes derivadas herdarem a *interface de uma função, bem como uma implementação padrão*.

Considere o caso se `Shape::error`:

```
class Shape {
public:
   virtual void error(const std::string& msg);
   ...
};
```

A interface diz que toda a classe deve suportar uma função que deve ser chamada quando é encontrado um erro, mas cada classe é livre para tratar erros da maneira que lhe for apropriada. Se uma classe não quiser fazer nada especial, ela pode simplesmente se basear no tratamento padrão de erros fornecido na classe `Shape`. Ou seja, a declaração de `Shape::error` diz aos projetistas das classes derivadas: "Você precisa oferecer suporte para uma função `error`, mas, se não quiser escrever a sua própria, pode se basear na versão padrão na classe `Shape`".

Acontece que pode ser perigoso permitir que funções virtuais simples especifiquem tanto uma interface de função quanto uma implementação padrão. Para ver por que, considere uma hierarquia de aeronaves para a Linhas Aéreas XYZ. A XYZ possui apenas dois tipos de aeronaves, o Modelo A e o Modelo B, e ambos voam exatamente da mesma forma. Logo, a XYZ projeta a seguinte hierarquia:

```
class Airport { ... };                                  // representa aeroportos

class Airplane {
public:
  virtual void fly(const Airport& destination);

  ...
};

void Airplane::fly(const Airport& destination)
{
  código padrão para pilotar uma aeronave até um dado destino
}

class ModelA: public Airplane { ... };

class ModelB: public Airplane { ... };
```

Para expressar que todas as aeronaves precisam oferecer suporte à função `fly` (voar), e em reconhecimento ao fato de que diferentes modelos de aeronave podem, em princípio, requerer implementações diferentes de `fly`, `Airplane::fly` (Aeronave::voar) é declarada como virtual. Entretanto, para evitar escrever código idêntico nas classes `ModelA` (Modelo A) e `ModelB` (Modelo B), o comportamento padrão de voo é fornecido como o corpo de `Airplane::fly`, o qual tanto `ModelA` quanto `ModelB` herdam.

Esse é um projeto orientado a objetos clássico. Duas classes compartilham um recurso comum (a maneira pela qual elas implementam `fly`), então o recurso comum é movido para a classe-base e o recurso é herdado pelas duas classes. Esse projeto faz os recursos comuns serem explícitos, evitando duplicação de código, facilitando melhorias futuras e tornando a manutenção de longo prazo mais fácil – todas as coisas pelas quais a tecnologia de orientação a objetos é tão conhecida. A Linhas Aéreas XYZ deveria ter orgulho disso.

Agora, suponhamos que, devido ao seu crescimento, a XYZ decida adquirir um novo tipo de aeronave, o Modelo C. O Modelo C difere de algumas maneiras do Modelo A e do Modelo B. Em particular, ele voa de maneira diferente. Os programadores da XYZ adicionam a classe para o Modelo C à hierarquia, mas, na pressa de colocar o novo modelo em serviço, esquecem de redefinir a função `fly`:

```
class ModelC: public Airplane {
  ...                                                   // nenhuma função fly é declarada
};
```

Em seu código, então, eles têm algo parecido com o seguinte:

```
Airport PDX(...);                                       // PDX é o aeroporto próximo a minha casa

Airplane *pa = new ModelC;

...

pa->fly(PDX);                                           // chama Airplane::fly!
```

É um desastre: o que se está tentando fazer é colocar um objeto do Modelo C (`ModelC`) a voar como se fosse um Modelo A ou um Modelo B. Esse não é o tipo de comportamento que inspira confiança nos passageiros.

O problema aqui não é `Airplane::fly` ter um comportamento padrão, mas permitir que `ModelC` herde esse comportamento sem dizer explicitamente que queria fazer isso. Felizmente, é fácil oferecer comportamento padrão para as classes derivadas, mas não dá-lo, a menos que peçam por ele. O truque é separar a conexão entre a *interface* da função virtual e sua *implementação* padrão. Veja uma maneira de fazer isso:

```
class Airplane {
public:
   virtual void fly(const Airport& destination) = 0;

   ...

protected:
   void defaultFly(const Airport& destination);
};

void Airplane::defaultFly(const Airport& destination)
{
   código padrão para pilotar uma aeronave até um dado destino
}
```

Observe como `Airplane::fly` foi modificada para ser uma função virtual *pura*, o que fornece a interface para voar. A implementação padrão está presente também na classe `Airplane`, mas agora na forma de uma função independente, chamada `defaultFly` (voo padrão). As classes como `ModelA` e `ModelB` que querem usar o comportamento padrão simplesmente fazem uma chamada internalizada para `defaultFly` dentro do corpo de `fly` (mas veja o Item 30 para mais informações sobre a interação da internalização com funções virtuais):

```
class ModelA: public Airplane {
public:
   virtual void fly(const Airport& destination)
   { defaultFly(destination); }

   ...
};

class ModelB: public Airplane {
public:
   virtual void fly(const Airport& destination)
   { defaultFly(destination); }

   ...
};
```

Para a classe `ModelC`, não existe a possibilidade de herdar acidentalmente a implementação incorreta de `fly`, porque a função virtual pura em `Airplane` força `ModelC` a fornecer sua própria versão de `fly`.

```
class ModelC: public Airplane {
public:
  virtual void fly(const Airport& destination);

  ...

};

void ModelC::fly(const Airport& destination)
{
  código padrão para pilotar uma aeronave ModelC até um dado destino
}
```

Esse método não é à prova de erros (os programadores ainda podem ter problemas com copiar e colar), mas é mais confiável que o projeto original. Assim como para `Airplane::defaultFly`, ele é protegido porque é realmente um detalhe de implementação de `Airplane` e suas classes derivadas. Os clientes que usam aeronaves devem se preocupar apenas com o fato de elas poderem voar, e não como o voo é implementado.

Também é importante o fato de `Airplane::defaultFly` ser uma função *não virtual*. Isso porque nenhuma classe derivada deve redefinir essa função, algo sobre o qual o Item 36 discute. Se `defaultFly` fosse virtual, você teria um problema circular: o que aconteceria se alguma classe derivada se esquecesse de redefinir `defaultFly` quando, supostamente, deveria fazer isso?

Algumas pessoas têm objeções à ideia de ter funções separadas para fornecer a interface e a implementação padrão, como `fly` e `defaultFly` acima. Primeiro, elas afirmam, isso polui o espaço de nomes da classe com uma proliferação de nomes de funções fortemente relacionados. Mesmo assim, elas concordam que a interface e a implementação padrão devem ser separadas. Como elas resolvem essa aparente contradição? Obtendo vantagem do fato de as funções virtuais puras poderem ser redeclaradas em classes derivadas concretas, mas elas também podem ter implementações próprias. Veja como a hierarquia de `Airplane` poderia tirar vantagem da habilidade de definir uma função virtual pura:

```
class Airplane {
public:
  virtual void fly(const Airport& destination) = 0;

  ...

};
```

```
void Airplane::fly(const Airport& destination)          // uma implementação de uma
{                                                       // função virtual pura
   código padrão para pilotar uma aeronave
   até um dado destino
}

class ModelA: public Airplane {
public:
   virtual void fly(const Airport& destination)
   { Airplane::fly(destination); }

   ...
};

class ModelB: public Airplane {
public:
   virtual void fly(const Airport& destination)
   { Airplane::fly(destination); }

   ...
};

class ModelC: public Airplane {
public:
   virtual void fly(const Airport& destination);

   ...
};

void ModelC::fly(const Airport& destination)
{
   código para pilotar uma aeronave ModelC até um dado destino
}
```

Esse é quase o mesmo projeto mostrado anteriormente, com a diferença de que o corpo da função virtual pura `Airplane::fly` pega o lugar da função independente `Airplane::defaultFly`. Em essência, `fly` foi quebrada em seus dois componentes fundamentais. Sua declaração especifica sua interface (que as classes derivadas *devem* usar), enquanto sua definição especifica seu comportamento padrão (que as classes derivadas *podem* usar, mas apenas se o requererem explicitamente). Ao mesclar `fly` com `defaultFly`, no entanto, você perde a habilidade de dar às duas funções diferentes níveis de proteção: o código que costumava ser protegido (estando em `defaultFly`) é agora público (porque agora ele está em `fly`).

Por fim, chegamos à função não virtual de `Shape`, chamada `objectID`:

```
class Shape {
public:
   int objectID( ) const;
   ...
};
```

Quando uma função membro é não virtual, supõe-se que ela se comporte de modo diferente nas classes derivadas. Na verdade, uma função membro não virtual especifica uma *invariante em relação à especialização*, porque ela identifica o comportamento que supostamente não deve ser modificado, independentemente de quão especializada uma classe se torne. Como tal,

- O objetivo de declarar uma função não virtual é fazer as classes derivadas herdarem *tanto a interface quanto uma implementação obrigatória* de uma função.

Você pode pensar na declaração de `Shape::objectID` assim: "todo objeto `Shape` possui uma função que retorna um identificador de objeto, e tal identificador de objeto é sempre computado da mesma maneira. Essa maneira é determinada pela definição de `Shape::objectID`, e nenhuma classe derivada deve tentar modificar como ela é feita". Como uma função não virtual identifica uma *invariante* em relação à especialização, ela nunca pode ser redefinida em uma classe derivada, um ponto que é discutido em detalhes no Item 36.

As diferenças nas declarações para funções virtuais puras, virtuais e não virtuais permitem que você especifique com precisão o que quer que as classes derivadas herdem: apenas a interface, a interface e uma implementação padrão, ou uma interface e uma implementação obrigatória. Como esses diferentes tipos de declarações significam coisas fundamentalmente diferentes, você deve escolher com cuidado entre eles quando declarar suas funções membro. Se fizer isso, você deve evitar os dois erros mais comuns feitos por projetistas de classe inexperientes.

O primeiro erro é declarar todas as funções como não virtuais. Isso não deixa espaço para a especialização em classes derivadas; os destrutores não virtuais são particularmente problemáticos (veja o Item 7). É claro, que é perfeitamente razoável projetar uma classe sem a intenção de que seja uma classe-base. Nesse caso, é mais apropriado um conjunto de funções membro exclusivamente não virtuais. Com muita frequência, porém, essas classes são declaradas ou por desconhecimento das diferenças entre funções virtuais e não virtuais ou como resultado de uma preocupação infundada quanto ao custo em termos de desempenho das funções virtuais. O que importa é que praticamente todas as classes que supostamente serão usadas como classes-base terão funções virtuais (mais uma vez, veja o Item 7).

Se você está preocupado com o custo das funções virtuais, permita-me lembrar a regra empiricamente baseada de 80/20 (veja também o Item 30), a qual afirma que, em um programa típico, 80% do tempo de execução será gasto em apenas 20% do código. Essa regra é importante, porque significa que, em média, 80% de suas chamadas a funções podem ser virtuais sem ter o mínimo impacto detectável no desempenho geral

de seu programa. Antes de você se preocupar em conseguir arcar com o custo de uma função virtual, certifique de que está focando nos 20% do programa em que a decisão pode realmente fazer alguma diferença.

O outro problema comum é declarar *todas* as funções membro como virtuais. Algumas vezes, é a coisa certa a fazer – veja as classes de Interface do Item 31. Entretanto, isso também pode indicar que o projetista de classes não entende o básico para poder tomar as decisões necessárias. Algumas funções *não* devem ser redefiníveis nas classes derivadas e, sempre que esse for o caso, você pode dizer isso tornando essas funções não virtuais. Não é útil para ninguém fingir que sua classe pode ser tudo para todos se eles tiverem que redefinir todas as suas funções. Se você tiver uma invariante em relação à especialização, não tenha medo de dizer isso!

Lembretes

» A herança de interface é diferente da herança de implementação. Sob herança pública, as classes derivadas sempre herdam as interfaces da classe-base.

» As funções virtuais puras especificam apenas herança da interface.

» As funções virtuais simples (impuras) especificam a herança de interface mais herança de uma implementação padrão.

» As funções não virtuais especificam herança de interface mais herança de uma implementação obrigatória.

Item 35: Considere alternativas ao uso de funções virtuais

Suponhamos que você esteja trabalhando em um vídeo game e esteja projetando uma hierarquia para os personagens do jogo. Sendo seu jogo de uma variedade violenta, é comum que os personagens estejam machucados ou em péssimo estado de vida. Logo, você decide oferecer uma função membro, `healthValue` (valor de saúde), que retorna um valor inteiro que indica a vida do personagem. Como os personagens podem calcular a vida de maneiras diferentes, declarar `healthValue` como virtual parece ser a maneira óbvia de projetar as coisas:

```
class GameCharacter {
public:
  virtual int healthValue( ) const;          // retorna a taxa de vida do personagem;
  ...                                        // as classes derivadas podem redefinir isso
};
```

O fato de `healthValue` não ser declarada como virtual pura sugere que existe um algoritmo padrão para calcular a saúde (veja o Item 34).

Essa é, na verdade, a maneira óbvia de projetar as coisas e, em certo sentido, seu ponto fraco. Como esse projeto é tão óbvio, você pode não dar a

atenção adequada às suas alternativas. Para ajudá-lo a escapar dos percalços do projeto orientado a objetos, vamos considerar outras maneiras de abordar esse problema.

O padrão método template com o idioma de interface não virtual

Começaremos com uma escola de pensamento interessante que argumenta que as funções virtuais devem, quase sempre, ser privadas. Os adeptos dessa escola sugerem que um projeto melhor reteria `healthValue` como função membro pública, mas a tornaria não virtual e faria com que ela chamasse uma função virtual privada para fazer o trabalho real, digamos, `doHealthValue` (calcular o valor de saúde):

```
class GameCharacter {
public:
  int healthValue( ) const                  // as classes derivadas não redefinem
  {                                         // isso – veja o Item 36

    ...                                     // faz coisas "antes" – veja abaixo

    int retVal = doHealthValue( );          // faz o trabalho real

    ...                                     // faz coisas "depois" – veja abaixo
    return retVal;
  }
  ...
private:
  virtual int doHealthValue( ) const        // as classes derivadas podem redefinir isso
  {
    ...                                     // algoritmo padrão para calcular
  }                                         // a saúde do personagem
};
```

Nesse código (e no resto deste item), mostro o corpo das funções membro em definições de classe. Como explica o Item 30, isso as declara explicitamente como internalizadas (`inline`). Mostro o código dessa maneira apenas para facilitar a visualização do que está acontecendo. Os projetos que descrevo são independentes de decisões de internalização; então, não pense que seja significativo o fato de as funções membro serem definidas dentro das classes. Não é.

Esse projeto básico – fazer os clientes chamarem funções virtuais privadas indiretamente através de funções membro não virtuais – é conhecido como o *idioma de interface não virtual (NVI)*. Essa é uma manifestação específica de um padrão de projeto mais geral chamado Método Template (um padrão que, infelizmente, não tem nada a ver com templates C++). Eu chamo a função não virtual (no caso, `healthValue`) de adaptador (wrapper) da função virtual.

Uma vantagem do idioma NVI é sugerida pelos comentários "fazer coisas "antes" e "fazer coisas 'depois'" no código. Esses comentários identificam segmentos de código que, com certeza, serão chamados antes e depois da função virtual que faz o trabalho real. Isso significa que o adaptador garante que, antes que uma função virtual seja chamada, o contexto apropriado seja configurado e que, depois de a chamada ter terminado, o contexto seja limpo. Por exemplo, as coisas "anteriores" poderiam incluir trancar um objeto de exclusão mútua, fazer uma entrada de log, verificar se as invariantes de classe e as pré-condições de funções são satisfeitas, etc. As coisas "posteriores" poderiam incluir a liberação de um objeto de exclusão mútua, a verificação de pós-condições de funções, a reverificação de invariantes de classes, etc. Não existe, realmente, uma boa maneira de fazer isso se você deixar que os clientes chamem as funções virtuais diretamente.

Pode-lhe ter ocorrido que o idioma NVI envolve classes derivadas que redefinem funções virtuais privadas – funções que elas não podem chamar! Não existe uma contradição de projeto aqui. Redefinir uma função virtual especifica *como* algo deve ser feito; chamar uma função virtual especifica *quando* isso será feito. Essas preocupações são independentes.

O idioma NVI permite que as classes derivadas redefinam uma função virtual, dando-nos controle sobre *como* as funcionalidades serão implementadas, mas a classe-base reserva-se o direito de dizer *quando* a função será camada. Isso pode parecer estranho a princípio, mas a regra que diz que as classes derivadas podem redefinir funções virtuais privadas herdadas é perfeitamente aceitável.

No idioma NVI, não é estritamente necessário que as funções virtuais sejam privadas. Em algumas hierarquias de classes, espera-se que as implementações de classes derivadas de uma função virtual invoquem suas funções correspondentes da classe-base (como no exemplo da página 120), e, para que essas chamadas sejam legais, as funções virtuais devem ser protegidas, e não privadas. Algumas vezes, uma função virtual precisa até mesmo ser pública (por exemplo, destrutores em classes-base polimórficas – veja o Item 7), mas então o idioma NVI não pode realmente ser aplicado.

O padrão Estratégia com ponteiros para funções

O idioma NVI é uma alternativa interessante às funções virtuais públicas, mas, do ponto de vista de projeto, é pouco mais do que uma decoração. Afinal, ainda estamos usando funções virtuais para calcular a vida de cada personagem. Uma asserção de projeto mais dramática seria dizer que o cálculo da vida de um personagem não depende do tipo de personagem – que esse cálculo não precisa, obrigatoriamente, fazer parte do personagem. Por exemplo, poderíamos requerer que seja passado um ponteiro para uma função de cálculo de vida para cada construtor de personagem, e poderíamos chamar tal função para realizar o cálculo propriamente dito:

```
class GameCharacter;                          // declaração mais a frente

// função para o algoritmo de cálculo de vida padrão
int defaultHealthCalc(const GameCharacter& gc);

class GameCharacter {
public:
  typedef int (*HealthCalcFunc)(const GameCharacter&);

  explicit GameCharacter(HealthCalcFunc hcf = defaultHealthCalc)
  : healthFunc(hcf )
  { }

  int healthValue( ) const
  { return healthFunc(*this); }

  ...

private:
  HealthCalcFunc healthFunc;
};
```

Essa abordagem é uma aplicação simples de outro padrão de projeto comum, chamado de Estratégia (Strategy). Comparada às abordagens baseadas em funções virtuais na hierarquia de `GameCharacter` (personagem do jogo), ela oferece uma flexibilidade interessante:

- Instâncias diferentes do mesmo tipo de personagem podem ter diferentes funções de cálculo de vida. Por exemplo:

```
class EvilBadGuy: public GameCharacter {
public:
  explicit EvilBadGuy(HealthCalcFunc hcf = defaultHealthCalc)
  : GameCharacter(hcf )
  { ... }

  ...
};

int loseHealthQuickly(const GameCharacter&);          // cálculo de vida
int loseHealthSlowly(const GameCharacter&);           // funções com diferentes
                                                      // comportamentos

EvilBadGuy ebg1(loseHealthQuickly);                   // personagens do mesmo
EvilBadGuy ebg2(loseHealthSlowly);                    // tipo com diferentes
                                                      // comportamentos
                                                      // relacionados à vida
```

- As funções de cálculo de vida para um personagem em particular podem ser modificadas em tempo de execução. Por exemplo, `GameCharacter` poderia oferecer uma função membro, `setHealthCalculator` (configura o calculador de saúde), que permitiria a substituição da função de cálculo de vida atual.

Por outro lado, o fato de a função de cálculo de vida não ser mais uma função membro da hierarquia de `GameCharacter` significa que ela não tem mais acesso especial algum às partes internas do objeto cuja vida está calculando. Por exemplo, `defaultHealthCalc` não possui acesso às partes não públicas de `EvilBadGuy` (bandido diabólico). Se a vida de um perso-

nagem pode ser calculada puramente por meio de informações disponíveis pela sua interface pública, isso não é um problema; mas, se o cálculo de vida precisa requerer informações não públicas, isso é um problema. Na verdade, é um problema em potencial que pode acontecer sempre que você substituir funcionalidades dentro de uma classe (por meio de uma função membro, por exemplo) por funcionalidade equivalente fora da classe (por meio de uma função não membro não amiga ou de uma função membro não amiga de outra classe). Essa questão continuará no restante deste item, porque todas as outras alternativas de projeto que consideraremos envolvem o uso de funções fora da hierarquia de `GameCharacter`.

Como regra geral, a única maneira de resolver a necessidade de acesso às partes não públicas de uma classe que as funções não membro têm é enfraquecendo o encapsulamento da classe. Por exemplo, a classe pode declarar as funções não membro como amigas (`friend`), ou pode oferecer funções de acesso públicas para partes de sua implementação que, de outra forma, preferiria manter ocultas. Se as vantagens de usar um ponteiro para função em vez de uma função virtual (como a habilidade de ter funções de cálculo de saúde por objeto e a habilidade de modificar essas funções em tempo de execução) forem maiores do que os prejuízos acarretados pela possibilidade de diminuir o encapsulamento de `GameCharacter` é algo que você deve decidir caso a caso.

O padrão estratégia (Strategy) com `tr1::function`

Uma vez que você se acostuma ao uso de templates e seu uso de interfaces implícitas (veja o Item 41), a abordagem baseada em ponteiros para funções se parece bastante rígida. Por que o calculador de vida precisaria ser uma função em vez de simplesmente ser algo que *age* como uma função (como um objeto função)? Se ele precisa ser uma função, por que não pode ser uma função membro? E por que ele deve retornar um `int` em vez de qualquer tipo que possa ser convertido para um `int`?

Essas restrições evaporam se substituirmos o uso de um ponteiro para função (como `healthFunc`) por um objeto do tipo `tr1::function`. Como o item 54 explica, esses objetos podem manter *qualquer entidade chamável* (como um ponteiro para função, um objeto função, ou um ponteiro para uma função membro) cuja assinatura seja compatível com o que é esperado. Veja o projeto que vimos há pouco, usando `tr1::function`:

```
class GameCharacter;                                    // como antes
int defaultHealthCalc(const GameCharacter& gc);         // como antes

class GameCharacter {
public:
  // HealthCalcFunc é qualquer entidade chamável que pode ser chamada com
  // qualquer coisa compatível com um GameCharacter e que retorna qualquer coisa
  // compatível com um int; veja abaixo para obter mais detalhes
  typedef std::tr1::function<int (const GameCharacter&)> HealthCalcFunc;
```

```
    explicit GameCharacter(HealthCalcFunc hcf = defaultHealthCalc)
    : healthFunc(hcf )
    { }

    int healthValue( ) const
    { return healthFunc(*this); }
    ...

private:
    HealthCalcFunc healthFunc;
};
```

Como você pode ver, `HealthCalcFunc` (função de cálculo de vida) é uma definição de tipo (`typedef`) para uma instanciação de `tr1::function`. Isso significa que ela age como um tipo generalizado de ponteiro para função. Olhe com cuidado para o que, especificamente, `HealthCalcFunc` é uma definição de tipo:

```
std::tr1::function<int (const GameCharacter&)>
```

Aqui destaquei a "assinatura alvo" dessa instanciação de `tr1::function`. Essa assinatura alvo é "uma função tomando uma referência a um `const GameCharacter` e retornando um `int`". Um objeto desse tipo `tr1::function` (ou seja, do tipo `HealthCalcFunc`) pode manter qualquer entidade chamável compatível com a assinatura alvo. Ser compatível significa que o parâmetro da entidade pode ser convertido implicitamente em `int`.

Comparado com o último projeto que vimos (no qual `GameCharacter` mantém um ponteiro para uma função), esse projeto é quase sempre o mesmo. A única diferença é que `GameCharacter` agora mantém um objeto `tr1::function` – um ponteiro *generalizado* para uma função. Essa mudança é tão pequena que eu que até diria inconsequente, não fosse o fato de que uma consequência sua ser que os clientes agora possuem uma flexibilidade muito maior ao especificar funções de cálculo de vida:

```
short calcHealth(const GameCharacter&);                 // função de cálculo
                                                        // de vida; observe
                                                        // o tipo de retorno não inteiro

struct HealthCalculator {                               // classe para objetos
    int operator( )(const GameCharacter&) const         // função para cálculo
    { ... }                                             // de vida
};

class GameLevel {
public:
    float health(const GameCharacter&) const;           // função membro
    ...                                                 // para cálculo de vida; observe
};                                                      // o tipo de retorno não inteiro


class EvilBadGuy: public GameCharacter {                // como antes
    ...
```

```
};
class EyeCandyCharacter: public GameCharacter {          // outro tipo
   ...                                                   // de personagem; assuma o mesmo
};                                                       // construtor de
                                                         // EvilBadGuy

EvilBadGuy ebg1(calcHealth);                             // personagem usando uma
                                                         // função de cálculo
                                                         // de vida

EyeCandyCharacter ecc1(HealthCalculator( ));             // personagem usando um
                                                         // objeto função
                                                         // de cálculo de vida
GameLevel currentLevel;
...
EvilBadGuy ebg2(                                         // personagem usando uma
   std::tr1::bind(&GameLevel::health,                    // função membro
                  currentLevel,                          // para cálculo de vida;
                  _1)                                    // veja abaixo para obter mais detalhes
);
```

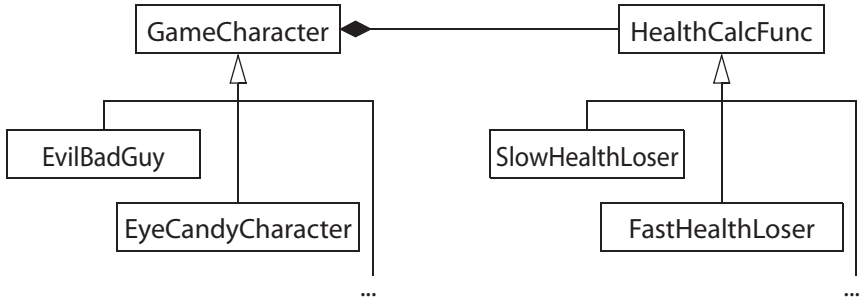
Pessoalmente, acho que `tr1::function` resolve tudo de uma maneira tão maravilhosa que me deixa muito empolgado. Se você não está empolgado, pode ser porque está encarando a definição de `ebg2` e tentando imaginar o que está acontecendo com a chamada a `tr1::bind` (tr1::vincular). Permita-me explicar isso.

Queremos dizer que, para calcular a taxa de vida de `ebg2`, deve ser usada a função membro `health` (vida) na classe `GameLevel` (fase do jogo). Agora, `GameLevel::health` é uma função declarada para receber um parâmetro (uma referência a um `GameCharacter`), mas ela, na verdade, precisa de dois parâmetros, porque também obtém um parâmetro `GameLevel` implícito – aquele para o qual `this` aponta. As funções de cálculo de vida para `GameCharacter`s, entretanto, recebem um único parâmetro: o `GameCharacter` cuja vida será calculada. Se fôssemos usar `GameLevel::health` para o cálculo de vida de `ebg2`, precisaríamos "adaptá-la" de alguma forma para que, em vez de receber dois parâmetros (um `GameCharacter` e um `GameLevel`), ela recebesse apenas um (um `GameCharacter`). Nesse exemplo, devemos sempre usar `currentLevel` (fase atual) como objeto `GameLevel` para o cálculo de vida de `ebg2`, então "vincular" `currentLevel` como o objeto `GameLevel` a ser usado cada vez que `GameLevel::health` for chamada para calcular a vida de `ebg2`. É isso que a chamada a `tr1::bind` faz: especifica que a função de cálculo de saúde de `ebg2` deve sempre usar `currentLevel` como o objeto `GameLevel`.

Estou ignorando um monte de detalhes quanto à chamada a `tr1::bind`, pois esses detalhes não fariam tanta diferença, e nos distrairiam do ponto fundamental que quero defender: ao usar `tr1::function` em vez de um ponteiro para função, permitimos que os clientes usem *qualquer entidade chamadora compatível* calcular a saúde de um personagem. Isso não é legal?

O padrão Estratégia "clássico"

Se você é mais chegado em padrões de projeto que em C++, uma abordagem mais convencional para o padrão Estratégia seria transformar a função de cálculo de vida em uma função membro virtual de uma hierarquia de cálculo de saúde separada. O projeto da hierarquia resultante se pareceria com o seguinte:



Se você não conhece a notação de UML, isso diz apenas que `GameCharacter` é a raiz de uma hierarquia de herança na qual `EvilBadGuy` e `EyeCandyCharacter` (personagem atraente) são classes derivadas; `HealthCalcFunc` é a raiz de uma hierarquia de herança com as classes `SlowHealthLoser` (perdedor lento de vida) e `FastHealthLoser` (perdedor rápido de vida); e cada objeto do tipo `GameCharacter` contém um ponteiro para um objeto da hierarquia de `HealthCalcFunc`.

Aqui está o esqueleto de código correspondente:

```
class GameCharacter;                                    // declaração à frente

class HealthCalcFunc {
public:
  ...
  virtual int calc(const GameCharacter& gc) const
  { ... }
  ...

};

HealthCalcFunc defaultHealthCalc;

class GameCharacter {
public:
  explicit GameCharacter(HealthCalcFunc *phcf = &defaultHealthCalc)
  : pHealthCalc(phcf )
  { }

  int healthValue( ) const
  { return pHealthCalc->calc(*this); }

  ...
private:
  HealthCalcFunc *pHealthCalc;
};
```

O interessante dessa abordagem é que ela é facilmente reconhecível pelas pessoas familiarizadas com a implementação "padrão" do padrão Estratégia, além de oferecer a possibilidade de um algoritmo de cálculo de vida existente poder ser modificado com a inclusão de uma classe derivada à hierarquia de `HealthCalcFunc`.

Resumo

O conselho fundamental deste item é considerar alternativas às funções virtuais ao buscar um projeto para o problema que você está tentando resolver. Veja uma rápida recapitulação das alternativas que examinamos:

- Use o **idioma de interface não virtual** (NVI), uma forma do padrão de projeto Método Template que envolve funções membro não virtuais públicas em torno de funções virtuais menos acessíveis.
- Substitua as funções virtuais por **membros de dados que sejam ponteiros para funções**, uma manifestação resumida do padrão de projeto Estratégia.
- Substitua as funções virtuais por **membros de dados do tipo** `tr1::function`, permitindo o uso de qualquer entidade chamável com uma assinatura compatível com o que você precisa. Isso também é uma forma do padrão de projeto Estratégia.
- Substitua as funções virtuais em uma hierarquia por **funções virtuais em outra hierarquia**. Essa é a implementação convencional do padrão de projeto Estratégia.

Essa não é uma lista exaustiva de alternativas de projeto às funções virtuais, mas ela deve ser capaz de convencê-lo de que *existem* alternativas. Além disso, suas vantagens e desvantagens comparativas devem deixar claro que você *deve* considerá-las.

Para evitar que você se prenda aos caminhos do projeto orientado a objetos, olhe ao seu redor de tempos em tempos. Existem muitos outros caminhos; vale a pena passar um tempo explorando-os.

Lembretes

» As alternativas às funções virtuais incluem o idioma NVI e várias formas do padrão de projeto Estratégia. O idioma NVI é, ele próprio, um exemplo do padrão de projeto Método Template.

» Uma desvantagem de mover funcionalidades de uma função membro para uma função fora da classe é que a função não membro não tem acesso aos membros não públicos da classe.

» Os objetos `tr1::function` agem como ponteiros generalizados para funções. Esses objetos oferecem suporte para todas as entidades chamáveis compatíveis com uma determinada assinatura-alvo.

Item 36: Nunca redefina uma função não virtual herdada

Suponhamos que eu lhe dissesse que uma classe `D` é publicamente derivada de uma classe `B` e que existe uma função membro `mf` definida na classe `B`. Os parâmetros e o tipo de retorno de `mf` não são importantes, então vamos considerar apenas que ambos são `void`. Em outras palavras, estou dizendo o seguinte:

```
class B {
public:
   void mf( );
   ...
};

class D: public B { ... };
```

Mesmo sem saber nada sobre `B`, `D` ou `mf`, dado um objeto `x` do tipo `D`,

```
D x;                                    // x é um objeto do tipo D
```

você provavelmente ficaria bastante surpreendido se

```
B *pB = &x;                             // obtém ponteiro para x

pB->mf( );                              // chama mf por meio de um ponteiro
```

se comportasse diferentemente disto:

```
D *pD = &x;                             // obtém ponteiro para x

pD->mf( );                              // chama mf por meio de um ponteiro
```

Isso porque, em ambos os casos, você está invocando a função membro no objeto `x`. Sendo ela a mesma função e o mesmo objeto em ambos os casos, deveria se comportar da mesma maneira, certo?

Certo, deveria. Mas ela pode não se comportar. Em particular, ela não se comporta se `mf` é não virtual e `D` define sua própria versão de `mf`:

```
class D: public B {
public:
   void mf( );                          // oculta B::mf; veja o Item 33

   ...

};

pB->mf( );                              // chama B::mf

pD->mf( );                              // chama D::mf
```

A razão para esse comportamento ambíguo é que as funções *não virtuais* como `B::mf` e `D::mf` são estaticamente vinculadas (veja o Item 37). Ou seja, uma vez que `pB` é declarado como de tipo ponteiro para `B`, as funções não virtuais invocadas por meio de `pB` serão *sempre* aquelas definidas para a

classe B, mesmo que pB aponte para um objeto de uma classe derivada de B, e é isso o que acontece nesse exemplo.

As funções *virtuais*, por outro lado, são dinamicamente vinculadas (mais uma vez, veja o Item 37), então, elas não sofrem desse problema. Se mf fosse uma função virtual, uma chamada a mf por meio de pB ou de pD resultaria em uma invocação de D::mf, porque o que é *realmente* apontado por pB e pD é um objeto do tipo D.

Se você estiver escrevendo a classe D e redefinir uma função não virtual mf que herda da classe B, os objetos da classe D provavelmente exibirão comportamento inconsistente. Em particular, qualquer objeto D pode agir tanto como B quanto como D quando mf for chamada, e o fator determinante não terá nada a ver com o objeto propriamente dito, mas com o tipo declarado do ponteiro que o aponta. As referências exibem o mesmo comportamento confuso dos ponteiros.

Mas esse é apenas um argumento pragmático. O que você realmente quer, eu sei, é um tipo de justificativa teórica para não redefinir as funções não virtuais herdadas. Tenho o prazer em fornecê-la.

O item 32 explica que a herança pública significa "é um(a)", e o Item 34 descreve por que declarar uma função não virtual em uma classe estabelece uma invariante sob a especialização para essa classe. Se você aplicar essas observações às classes B e D e à função membro B::mf, então

- Tudo o que se aplica a objetos B também se aplica a objetos D, porque todo objeto D é um objeto B;
- As classes derivadas de B devem herdar tanto a interface *quanto* a implementação de mf, porque mf é não virtual em B.

Agora, se D redefinir mf, existe uma contradição em seu projeto. Se D *realmente* precisa implementar mf de modo diferente de B, e se todo objeto B – independentemente do grau de especialização – *realmente* precisa usar a implementação de B para mf, então não é verdade que todo D é um B. Nesse caso, D não deveria herdar publicamente de B. Por outro lado, se D *realmente* precisa herdar publicamente de B, e se D realmente precisa implementar mf de modo diferente de B, então não é verdade que mf reflete uma invariante em relação à especialização de B. Nesse caso, mf deveria ser virtual. Por fim, se cada D *realmente* é um B, e se mf realmente corresponde a uma invariante em relação à especialização de B, então D não precisa redefinir mf, e ele não deve tentar isso.

Independentemente de qual argumento se aplica, é preciso abrir mão de algo, e sob nenhuma circunstância isso deve ser a proibição de redefinir uma função não virtual herdada.

Se este item lhe dá uma sensação de *déjà vu*, é porque você provavelmente já leu o Item 7, que explica por que os destrutores em classes-base

polimórficas devem ser virtuais. Se você violar essa recomendação (por exemplo, declarando um destrutor não virtual em uma classe-base polimórfica), estará violando também esta recomendação, porque as classes derivadas invariavelmente redefiniriam uma função não virtual herdada: o destrutor da classe-base. Isso seria verdade mesmo para as classes derivadas que não declaram construtor algum, porque, como o Item 5 explica, o destrutor é uma das funções membro que os compiladores geram se você mesmo não fornecer um. Em essência, o item 7 não é nada mais do que um caso especial deste item, apesar de ser importante o suficiente para merecer ser um item por si só.

Lembrete

» Nunca redefina uma função não virtual herdada.

Item 37: Nunca redefina um valor padrão de parâmetro herdado de uma função

Vamos simplificar essa discussão desde o início. Existem apenas dois tipos de funções que você pode herdar: virtuais e não virtuais. Entretanto, é sempre um erro redefinir uma função não virtual herdada (veja o Item 36); então, podemos, seguramente, limitar nossa discussão aqui à situação na qual você herda uma função *virtual* com um valor padrão de parâmetro.

Sendo esse o caso, a justificativa deste item torna-se bastante direta: as funções virtuais são vinculadas dinamicamente, mas os valores de parâmetros padrão são vinculados estaticamente.

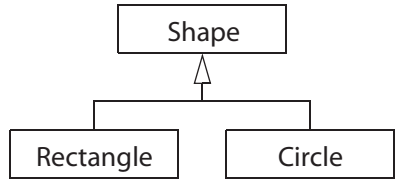
O que é isso? Você está dizendo que a diferença entre vinculação estática e dinâmica deu um nó sua mente já sobrecarregada? (Só para constar, a vinculação estática também é conhecida como *vinculação precoce*, e a vinculação dinâmica também é conhecida como *vinculação tardia*.) Vamos revisar, então.

O *tipo estático* de um objeto é o tipo que você declarou que ele tinha no texto do programa. Considere a seguinte hierarquia de classes:

```
// uma classe para formas geométricas
class Shape {
public:
   enum ShapeColor { Red, Green, Blue };

   // todas as formas devem oferecer uma função para que elas se desenhem
   virtual void draw(ShapeColor color = Red) const = 0;
   ...
};
```

```
class Rectangle: public Shape {
public:
   // observe o valor padrão de parâmetro diferente – ruim!
   virtual void draw(ShapeColor color = Green) const;
   ...
};

class Circle: public Shape {
public:
   virtual void draw(ShapeColor color) const;
   ...
};
```

Graficamente, essa hierarquia se parece com o seguinte:



Agora, considere os seguintes ponteiros:

```
Shape *ps;                          // tipo estático = Shape*
Shape *pc = new Circle;             // tipo estático = Shape*
Shape *pr = new Rectangle;          // tipo estático = Shape*
```

Nesse exemplo, `ps`, `pc` e `pr` são declarados como do tipo ponteiro para `Shape` (forma), de modo que todos possuam esse tipo como seu tipo estático. Observe que não faz absolutamente nenhuma diferença para o que eles estão *realmente* apontando – seu tipo estático é `Shape*` independentemente disso.

O *tipo dinâmico* de um objeto é determinado pelo tipo do objeto ao qual atualmente se refere. Ou seja, seu tipo dinâmico indica como ele se comportará. No exemplo acima, o tipo dinâmico de `pc` é `Circle*` (ponteiro para um círculo), e o tipo dinâmico de `pr` é `Rectangle*` (ponteiro para um retângulo). O ponteiro `ps` não tem, na verdade, um tipo dinâmico, porque não se refere a nenhum objeto (ainda).

Os tipos dinâmicos, como o nome sugere, podem ser modificados à medida que o programa é executado, em geral, por meio de atribuições:

```
ps = pc;                            // o tipo dinâmico de ps é
                                    // agora Circle*

ps = pr;                            // o tipo dinâmico de ps é
                                    // agora Rectangle*
```

As funções virtuais são *dinamicamente vinculadas*, ou seja, a função específica chamada é determinada pelo tipo dinâmico do objeto por meio do qual ela é invocada:

```
pc->draw(Shape::Red);                    // chama Circle::draw(Shape::Red)
pr->draw(Shape::Red);                    // chama Rectangle::draw(Shape::Red)
```

Isso tudo não é novidade, eu sei; você certamente entende de funções virtuais. A virada ocorre quando você pensa nas funções virtuais com valores de parâmetros padrão, porque, como eu disse antes, as funções virtuais são vinculadas dinamicamente, mas os parâmetros padrão são vinculados estaticamente. Isso significa que você pode acabar invocando uma função definida em uma *classe derivada*, mas usando um valor padrão de parâmetro de uma *classe-base*:

```
pr->draw( );                             // chama Rectangle::draw(Shape::Red)!
```

Nesse caso, o tipo dinâmico de `pr` é `Rectangle*`, então a função virtual de `Rectangle` é chamada, justamente o que você esperaria. Em `Rectangle::draw`, o valor padrão de parâmetro é `Green` (verde). Entretanto, como o tipo estático de `pr` é `Shape*`, o valor padrão de parâmetro para essa chamada à função é obtido da classe `Shape`, não da classe `Rectangle`! O resultado é uma chamada que consiste em uma estranha e quase sempre imprevista combinação de declarações para `draw`, tanto para `Shape` quanto para a classe `Rectangle`.

O fato de `ps`, `pc` e `pr` serem ponteiros não tem consequência nessa questão. Se eles fossem referências, o problema persistiria. As únicas coisas importantes são que `draw` é uma função virtual, e que um de seus valores de parâmetros padrão é redefinido em uma classe derivada.

Por que C++ insiste em agir dessa maneira perversa? A resposta tem a ver com eficiência em tempo de execução. Se os valores padrão de parâmetros fossem vinculados dinamicamente, os compiladores teriam que achar um jeito de determinar o(s) valor(es) padrão apropriado(s) para os parâmetros de funções virtuais em tempo de execução, o que seria mais lento e mais complicado que o mecanismo atual para determiná-los durante a compilação. A decisão foi equivocada quanto à velocidade e à simplicidade de implementação, e o resultado é que agora você tem um comportamento de execução eficiente, mas que, se você não conseguir adotar a recomendação deste item, é confuso.

Tudo está indo muito bem, mas veja o que acontece se você tentar seguir esta regra e também oferecer valores padrão de parâmetros aos usuários das classes-base e das derivadas:

```
class Shape {
public:
   enum ShapeColor { Red, Green, Blue };

   virtual void draw(ShapeColor color = Red) const = 0;
   ...
};
```

```
class Rectangle: public Shape {
public:
   virtual void draw(ShapeColor color = Red) const;
   ...
};
```

Opa, duplicação de código. Pior ainda, duplicação de código com dependências: se o valor padrão de parâmetro é modificado em `Shape`, todas as classes derivadas que o repetem também devem ser modificadas. De outra maneira, elas terminariam redefinindo um valor de padrão de parâmetro herdado. O que fazer?

Quando você está com dificuldade para fazer uma função virtual se comportar da maneira que gostaria, é sábio considerar projetos alternativos, e o Item 35 é repleto de alternativas às funções virtuais. Uma das alternativas é o *idioma de interface não virtual* (idioma NVI): ter uma função pública não virtual em uma classe-base que chame uma função virtual privada que as classes derivadas possam redefinir. Aqui, temos a função não virtual especificando o parâmetro padrão, enquanto a função virtual faz o trabalho propriamente dito:

```
class Shape {
public:
   enum ShapeColor { Red, Green, Blue };

   void draw(ShapeColor color = Red) const                // agora uma função não virtual
   {
      doDraw(color);                                      // chama uma virtual
   }

   ...
private:
   virtual void doDraw(ShapeColor color) const = 0;       // o trabalho propriamente dito é
};                                                         // feito nesta função

class Rectangle: public Shape {
public:
   ...

private:
   virtual void doDraw(ShapeColor color) const;           // observe a falta de
   ...                                                    // um valor padrão de parâmetro
};
```

Como as funções não virtuais nunca devem ser sobrescritas por classes derivadas (veja o Item 36), esse projeto deixa claro que o valor padrão para o parâmetro `color` (cor) de `draw` (desenhar) deve sempre ser `Red` (vermelho).

Lembrete

» Nunca redefina um valor padrão de parâmetros herdados, porque os valores padrão herdados são estaticamente vinculados, enquanto as funções virtuais – as únicas funções que você deve sobrescrever – são dinamicamente vinculadas.

Item 38: Modele "tem um(a)" ou "é implementado(a) em termos de" com composição

A *composição* é a relação entre tipos que surge quando os objetos de um tipo contêm objetos de outro tipo. Por exemplo:

```
class Address { ... };                          // onde alguém vive

class PhoneNumber { ... };

class Person {
public:
   ...
private:
   std::string name;                            // objeto composto
   Address address;                             // idem anterior
   PhoneNumber voiceNumber;                     // idem anterior
   PhoneNumber faxNumber;                       // idem anterior
};
```

Nesse exemplo, os objetos `Person` (pessoa) são compostos de objetos `string` (cadeia de caracteres), `Address` (endereço) e `PhoneNumber` (número de telefone). Entre os programadores, o termo *composição* tem diversos sinônimos; ela também é conhecida como *uso de camadas*, *uso de contêineres*, *agregação* e *embarque de objetos (embedding)*.

O Item 32 explica que a herança pública significa "é um(a)". A composição possui um significado também. Na verdade, ela possui dois significados. A composição significa "tem um(a)" ou "é implementado(a) em termos de". Isso porque você está lidando com dois domínios diferentes em seu sistema de software. Alguns objetos em seus programas correspondem a coisas no mundo que você está modelando, como pessoas, veículos, frames de vídeo, etc. Esses objetos fazem parte de um *domínio de aplicação*. Outros objetos são puramente artefatos de implementação, como buffers, objetos de exclusão mútua, árvores de busca, etc. Esses tipos de objetos correspondem ao *domínio de implementação* de seu sistema de software. Quando a composição ocorre entre objetos no domínio de aplicação, ela expressa uma relação "tem um(a)". Quando ela ocorre no domínio de implementação, expressa uma relação "é implementado(a) em termos de".

A classe `Person` citada demonstra a relação "tem um(a)". Um objeto `Person` possui um nome, um endereço e números de telefone (voz e fax). Você não diria que uma pessoa é um nome ou um endereço; você diria que ela tem um nome e tem um endereço. A maioria das pessoas não tem muita dificuldade com essa distinção, então a confusão entre os papéis de "é um(a)" e "tem um(a)" é relativamente rara.

Mais problemática é a diferença entre "é um(a)" e "é implementado(a) em termos de". Por exemplo, suponhamos que você precise de um template para as classes que representam pequenos conjuntos de objetos, ou seja, coleções sem duplicatas. Como a reutilização é algo maravilhoso, seu primeiro instinto é empregar o template `set` (conjunto) da biblioteca padrão.

Por que escrever um novo template quando você pode usar um que já foi escrito?

Infelizmente, as implementações de `set`, em geral, incorrem em um custo extra de três ponteiros para cada elemento. Isso porque os conjuntos são normalmente implementados como árvores de busca balanceadas, algo que lhes permite garantir buscas, inserções e remoções em tempo logarítmico. Quando a velocidade é mais importante do que o espaço, esse é um projeto racional, mas acontece que, em sua aplicação, o espaço é mais importante do que a velocidade. Assim, o `set` da biblioteca padrão oferece a substituição errada para você (ocupa mais espaço por um ganho de velocidade). Parece que você precisará escrever seu próprio template no fim das contas.

Mesmo assim, a reutilização *é* algo maravilhoso. Sendo o ás das estruturas de dados que é, você sabe que uma, dentre as muitas escolhas para implementar conjuntos, são as listas encadeadas. Você também sabe que a biblioteca padrão de C++ tem um template `list`, então decide reutilizá-lo.

Em particular, você decide que seu novo template `Set` (conjunto) herda de `list` (lista). Ou seja, `Set<T>` herdará de `list<T>`. Afinal, em sua implementação, um objeto `Set` será, na verdade, um objeto `list`. Você então declara seu template `Set` como:

```
template<typename T>                          // a maneira errada de usar list para Set
class Set: public std::list<T> { ... };
```

Tudo pode parecer bem nesse ponto, mas, na verdade, há algo muito errado. Como explica o Item 32, se `D` é um `B`, tudo que é verdadeiro para `B` também é verdadeiro para `D`. Entretanto, um objeto `list` pode conter duplicatas, então, se o valor 3051 é inserido em uma lista definida como `list<int>` duas vezes, essa lista conterá duas cópias de 3051. Em contraste, um `Set` não pode conter duplicatas, então, se o valor 3051 é inserido em `Set<int>` duas vezes, o conjunto contém apenas uma cópia do valor. Assim, não é verdade que um `Set` é um `list`, porque algumas das coisas que são verdadeiras para objetos `list` não são verdadeiras para objetos `Set`.

Como o relacionamento entre essas duas classes não é do tipo "é um(a)", a herança pública é a maneira errada de modelar esse relacionamento. A maneira correta é entender que um objeto `Set` pode ser *implementado em termos de* um objeto `list`:

```
template<class T>                             // a maneira correta de usar list para Set
class Set {
public:
  bool member(const T& item) const;

  void insert(const T& item);
  void remove(const T& item);

  std::size_t size( ) const;

private:
  std::list<T> rep;                           // representação para os dados de Set
};
```

As funções membro de `Set` podem depender fortemente das funcionalidades já oferecidas por `list` e por outras partes da biblioteca padrão; então, a implementação é direta, desde que você esteja familiarizado com o básico da programação com a STL:

```
template<typename T>
bool Set<T>::member(const T& item) const
{
  return std::find(rep.begin( ), rep.end( ), item) != rep.end( );
}

template<typename T>
void Set<T>::insert(const T& item)
{
  if (!member(item)) rep.push_back(item);
}

template<typename T>
void Set<T>::remove(const T& item)
{
  typename std::list<T>::iterator it =          // veja o Item 42 para mais informações sobre
    std::find(rep.begin( ), rep.end( ), item);  // "typename" aqui

  if (it != rep.end( )) rep.erase(it);
}

template<typename T>
std::size_t Set<T>::size( ) const
{
  return rep.size( );
}
```

Essas funções são simples o suficiente para serem candidatas lógicas à internalização, embora eu saiba que você vai querer revisar a discussão no Item 30 antes de tomar qualquer decisão concreta sobre internalizações.

Pode-se argumentar que a interface de `Set` estaria mais em concordância com a recomendação do Item 18 (projetar interfaces que sejam fáceis de usar corretamente e difíceis de usar incorretamente), caso fossem seguidas as convenções para os contêineres STL. No entanto, seguir essas convenções aqui exigiria a inclusão de um monte de coisas que tornariam o relacionamento entre `Set` e `list` obscuro. Como o relacionamento é o ponto principal deste item, trocaremos a compatibilidade com a STL por clareza didática. Além do quê, detalhes sobre a interface de `Set` não devem ofuscar o que é incontestavelmente correto a respeito de `Set`: o relacionamento entre ele e `list`. O relacionamento não é do tipo "é um(a)" (apesar de, a princípio, parecer ser); é do tipo "é implementado(a) em termos de".

Lembretes

- A composição tem significados completamente diferentes da herança pública.
- No domínio de aplicação, a composição significa "tem um(a)". No domínio de implementação, significa "é implementado(a) em termos de".

Item 39: Use a herança privada com bom-senso

O Item 32 mostra que C++ trata a herança pública como um relacionamento "é um(a)". Ele faz isso mostrando que os compiladores, quando é dada uma hierarquia na qual uma classe `Student` (alunos) herda publicamente da classe `Person` (pessoa), convertem implicitamente alunos em pessoas quando isso for necessário para que uma chamada à função seja bem-sucedida. Vale a pena repetir uma parte daquele exemplo usando herança privada em vez de herança pública:

```
class Person { ... };

class Student: private Person { ... };      // a herança agora é privada

void eat(const Person& p);                  // qualquer um pode comer

void study(const Student& s);               // só os estudantes estudam

Person p;                                   // p é uma pessoa (Person)
Student s;                                  // s é um aluno (Student)

eat(p);                                     // ok, p é uma pessoa (Person)

eat(s);                                     // erro! um aluno (Student) não é uma pessoa (Person)
```

Claramente, a herança privada não significa "é um(a)". O que significa então?

"Opa!", você diz. "Antes de irmos para o significado, vamos analisar o comportamento. Como a herança privada se comporta?" Bem, a primeira regra que governa a herança privada, você acabou de ver em ação: ao contrário da herança pública, os compiladores geralmente *não* convertem um objeto da classe derivada (como `Student`) em um objeto da classe-base (como `Person`) se o relacionamento de herança entre as classes for privado. É por isso que a chamada a `eat` (comer) falha para o objeto `s`. A segunda regra é que os membros herdados de uma classe-base privada se tornam membros privados da classe derivada, mesmo se fossem protegidos ou públicos na classe-base.

Isso tudo nos trás de volta ao significado. A herança privada significa "é implementado(a) em termos de". Se você faz uma classe `D` herdar privadamente de uma classe `B`, faz isso porque está interessado em tirar vantagem de alguns dos recursos disponíveis na classe `B`, e não porque existe um relacionamento conceitual entre objetos dos tipos `B` e `D`. Como tal, a herança privada é puramente uma técnica de implementação. (É por isso que tudo o que você herda de uma classe-base privada se torna privado em sua classe; tudo não passa de detalhes de implementação.) Usando os termos introduzidos no Item 34, a herança privada significa que *apenas* a implementação deve ser herdada; a interface deve ser ignorada. Se `D` herda privadamente de `B`, significa que os objetos `D` são implementados em termos de objetos `B`, nada mais. A herança privada não significa nada durante o *projeto* de software, apenas durante a *implementação* de software.

O fato de a herança privada significar "é implementada em termos de" é um pouco perturbador, porque o Item 38 afirma que a composição pode significar a mesma coisa. Como, então, você escolhe entre eles? A resposta é simples: use composição sempre que puder, e use a herança privada sempre que precisar. Quando você precisa? Principalmente quando os membros protegidos e/ou as funções virtuais entram no contexto, apesar de existir também um caso limite no qual as questões de espaço podem apontar em direção à herança privada. Vamos nos preocupar com o caso limite mais tarde. Afinal, é um caso limite.

Suponhamos que estivéssemos trabalhando em uma aplicação envolvendo `Widget`s e decidimos que precisaríamos entender melhor como os `Widget`s estão sendo usados. Por exemplo, não apenas queremos saber coisas como a frequência com que as funções membros de `Widget` são chamadas, mas também como as taxas de chamadas mudam com o tempo. Os programas com fases distintas de execução podem ter diferentes perfis comportamentais durante fases diferentes. Por exemplo, as funções usadas durante a fase de análise sintática de um compilador são muito diferentes das funções usadas durante a otimização e a geração de código.

Decidimos modificar a classe `Widget` para rastrear quantas vezes cada função membro é chamada. Em tempo de execução, examinaremos periodicamente essa informação, possivelmente junto com os valores de cada `Widget` e quaisquer outros dados que nos parecerem úteis. Para fazer isso funcionar, precisamos configurar algum tipo de cronômetro, para que saibamos quando é hora de coletar as estatísticas de uso.

Preferindo reutilizar código existente em vez de escrever código novo, reviramos nossa caixa de ferramentas e ficamos felizes de encontrar a seguinte classe:

```
class Timer {
public:
   explicit Timer(int tickFrequency);

   virtual void onTick( ) const;                    // chamada automaticamente para
                                                    // cada intervalo de tempo (tick)

   ...
};
```

Era justamente o que estávamos procurando. Um objeto `Timer` (cronômetro) pode ser configurado para ser acionado sempre que precisarmos, e, em cada acionamento, ele chamará uma função virtual. Podemos redefinir essa função virtual de forma que ela examine o estado atual do mundo de `Widget`. Perfeito!

Para `Widget` redefinir uma função virtual em `Timer`, `Widget` deve herdar de `Timer`. Mas a herança pública é inapropriada nesse caso. Não é verdade que um `Widget` é um `Timer`. Os clientes de `Widget` não devem ser capazes de chamar `onTick` (quando ocorrer um acionamento) em um `Widget`, porque isso não faz parte da interface conceitual de `Widget`. Permitir essa chamada

à função faria com que ficasse mais fácil para os clientes de `Widget` usarem sua interface incorretamente, uma clara violação do conselho do Item 18 de deixar as interfaces fáceis de serem usadas corretamente e difíceis de serem usadas incorretamente. A herança pública não é uma opção válida aqui.

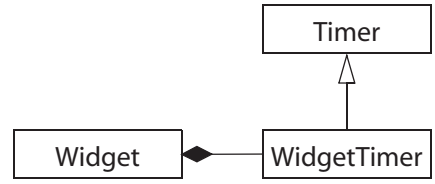
Logo, herdamos privadamente:

```
class Widget: private Timer {
private:
   virtual void onTick( ) const;                    // verifica os dados de uso de Widget, etc.
   ...
};
```

Por meio da herança privada, a função pública `onTick` de `Timer` torna-se privada em `Widget`, e mantemos isso assim quando a redeclaramos. Mais uma vez, colocar `onTick` na interface pública induziria incorretamente os clientes a pensar que podem chamá-la, e isso violaria o Item 18.

Esse é um bom projeto, mas vale a pena observar que a herança privada não é estritamente necessária. Se estivéssemos determinados a usar composição no lugar dela, poderíamos. Seria só declarar uma classe aninhada privada dentro de `Widget` que herdaria publicamente de `Timer`, redefiniria `onTick` lá e colocaria um objeto desse tipo dentro de `Widget`. Aqui está um rascunho da abordagem:

```
class Widget {
private:
   class WidgetTimer: public Timer {
   public:
      virtual void onTick( ) const;
      ...
   };
   WidgetTimer timer;
   ...
};
```



Esse projeto é mais complicado do que aquele que usa apenas herança privada, porque envolve tanto herança (pública) quanto composição, bem como a introdução de uma nova classe (`WidgetTimer` – um cronômetro para `Widgets`). Para ser honesto, eu o mostrei aqui principalmente para lembrá-lo de que existe mais de uma maneira de abordar um problema de projeto, e vale a pena nos treinarmos para considerar múltiplas abordagens (veja também o Item 35). Independentemente disso, posso pensar em duas razões pelas quais você pode preferir a herança pública e a composição em vez da herança privada.

Primeiro, você pode querer projetar `Widget` de forma a permitir classes derivadas, mas também pode querer impedir que as classes derivadas redefinam `onTick`. Se `Widget` herdar de `Timer`, isso não será possível, nem mesmo se a herança for privada. (Lembre-se do Item 35, de que as classes derivadas podem redefinir funções virtuais mesmo que não lhes seja permitido chamar essas funções.) Mas se `WidgetTimer` for privada em `Widget` e herdar de

`Timer`, as classes derivadas de `Widget` não terão acesso a `WidgetTimer`, logo, não poderão herdar dela ou redefinir suas funções virtuais. Se você já programou em Java ou em C# e sente falta da capacidade de impedir que as classes derivadas redefinam as funções virtuais (ou seja, os métodos finais – `final` – de Java e os métodos selados – `sealed` – de C#), agora tem uma ideia de como chegar mais próximo desse comportamento em C++.

Segundo, você pode querer minimizar as dependências de compilação de `Widget`. Se `Widget` herda de `Timer`, a definição de `Timer` deve estar disponível quando `Widget` for compilada, então o arquivo que define `Widget` provavelmente precisa incluir `Timer.h`. Por outro lado, se `WidgetTimer` for movida para fora de `Widget`, e `Widget` contiver apenas um ponteiro para `WidgetTimer`, `Widget` pode fazer uma declaração simples para a classe `WidgetTimer`; ela não precisa incluir nada relacionado a `Timer`. Para sistemas grandes, esses desacoplamentos podem ser importantes. (Para obter mais detalhes sobre como minimizar as dependências de compilação, consulte o Item 31.)

Destaquei anteriormente que a herança privada é útil principalmente quando uma classe que gostaria de ser derivada quer acesso às partes protegidas de uma classe que gostaria de ser base, ou quando ela gostaria de redefinir uma ou mais de suas funções virtuais, mas o relacionamento conceitual entre as classes é "é implementado(a) em termos de" em vez de "é um(a)". Entretanto, eu também disse que existia um caso limite envolvendo otimização de espaço que poderia levá-lo a preferir a herança privada à composição.

O caso limite é, de fato, bastante limite: ele se aplica apenas quando você está lidando com uma classe que não possui nenhum dado nela. Essas classes não possuem membros de dados não estáticos, nenhuma função virtual (porque a existência dessas funções adiciona um *vptr* a cada objeto – veja o Item 7) e nenhuma classe-base virtual (porque essas classes-base também incorrem em um custo de tamanho – veja o Item 40). Conceitualmente, os objetos dessas classes vazias não devem usar espaço, pois não existem dados por objeto a serem armazenados. Entretanto, existem razões técnicas pelas quais C++ decreta que esses objetos devem ter tamanho diferente de zero, então, se você fizer o seguinte,

```
class Empty { };                    // não tem dados, então os objetos não devem
                                    // usar memória

class HoldsAnInt {                  // devem precisar de apenas um espaço para um inteiro
private:
  int x;
  Empty e;                          // não deve exigir memória
};
```

descobrirá que `sizeof(HoldsAnInt) > sizeof(int);` um membro de dados `Empty` (vazio) requer memória. Na maioria dos compiladores, `sizeof(Empty)` é 1, porque o decreto de C++ contra objetos livres de tamanho zero em geral é satisfeito pela inserção silenciosa de um `char` nos objetos "vazios". Entretanto, os requisitos de alinhamento (veja o Item 50)

podem fazer os compiladores adicionarem deslocamento a classes como `HoldsAnInt` (mantém um inteiro); assim, é provável que os objetos de `HoldsAnInt` ganhem apenas o tamanho de um `char`; provavelmente, eles crescerão o suficiente para armazenar um segundo inteiro. (Em todos os compiladores que testei, foi o que ocorreu.)

Mas talvez você tenha notado que fui cuidadoso ao dizer que objetos "livres" não devem ter tamanho zero. Essa restrição não se aplica a partes da classe-base de objetos da classe derivada, porque não são livres. Se você *herdar* de `Empty` em vez de conter um objeto deste tipo,

```
class HoldsAnInt: private Empty {
private:
   int x;
};
```

é quase certo que descobrirá que `sizeof(HoldsAnInt) == sizeof(int)`. Isso é conhecido como a *otimização de base vazia* (EBO – *empty base optimization*), e é algo implementado por todos os compiladores que testei. Se você é um desenvolvedor de bibliotecas cujos clientes se preocupam com espaço, vale a pena conhecer a EBO. Também vale a pena saber que a EBO geralmente é viável apenas sob herança simples. As regras que governam o layout de objetos em C++ geralmente dizem que a EBO não pode ser aplicada às classes derivadas que possuem mais de uma base.

Na prática, as classes "vazias" não são realmente vazias. Apesar de nunca terem membros de dados não estáticos, frequentemente contêm definições de tipos (`typedefs`), enumerações, membros de dados estáticos ou funções virtuais. A STL possui muitas classes tecnicamente vazias que contêm membros úteis (normalmente definições de tipos), incluindo as classes-base `unary_function` e `binary_function`, das quais as classes para objetos funções definidos pelo usuário em geral herdam. Graças à implementação ampla da EBO, essa herança raramente aumenta o tamanho das classes que estão herdando.

Mesmo assim, vamos voltar ao básico. A maioria das classes não é vazia, então a EBO é raramente uma justificativa legítima para usar herança privada. Além disso, a maioria das heranças corresponde a um relacionamento "é um(a)", e esse é um trabalho para a herança pública, e não privada. Tanto a composição quanto a herança privada significam "é implementado(a) em termos de", mas a composição é mais fácil de entender, então você deve usá-la sempre que puder.

A herança privada provavelmente é uma estratégia de projeto legítima quando você estiver lidando com duas classes não relacionadas por um relacionamento "é um(a)", em que uma delas precisa acessar os membros protegidos da outra, ou precisa redefinir uma ou mais de suas funções virtuais. Mesmo nesse caso, vimos que um misto de herança pública e composição pode, frequentemente, levar ao comportamento que você quer, apesar de haver um aumento na complexidade do projeto. Usar a herança privada

com bom-senso significa empregá-la quando, tendo considerado todas as alternativas, ela for a melhor maneira de expressar o relacionamento entre duas classes em seu sistema de software.

Lembretes

» A herança privada significa "é implementado(a) em termos de". Em geral, é inferior à composição, mas faz sentido quando uma classe derivada precisa acessar membros protegidos da classe-base ou precisa redefinir as funções virtuais herdadas.

» Diferentemente da composição, a herança privada pode permitir a otimização de base vazia. Isso pode ser importante para os desenvolvedores que buscam minimizar o tamanho de seus objetos.

Item 40: Use a herança múltipla com bom-senso

Quando o assunto é herança múltipla (HM), a comunidade de C++, em geral, divide-se em dois grupos básicos. Um grupo acredita que, se a herança simples (HS) for boa, a herança múltipla só pode ser melhor. O outro argumenta que a herança simples é boa, mas a herança múltipla não vale o incômodo. Neste item, nosso objetivo principal é entender ambas as perspectivas relacionadas à questão do uso de HM.

Uma das primeiras coisas a reconhecer é que, quando a HM entra no mundo do projeto de software, torna-se possível herdar o mesmo nome (por exemplo, de uma função, de uma definição de tipo, etc) de mais de uma classe-base. Isso leva a novas oportunidades para a ocorrência de ambiguidades. Por exemplo:

```
class BorrowableItem {                  // algo que uma biblioteca deixa você pegar emprestado
public:
   void checkOut( );                    // retira o item da biblioteca
   ...
};

class ElectronicGadget {
private:
   bool checkOut( ) const;              // realiza autoteste, retorna se
   ...                                  // o teste for bem-sucedido
};

class MP3Player:                        // observe a HM aqui
   public BorrowableItem,               // (algumas bibliotecas emprestam tocadores de MP3)
   public ElectronicGadget
{ ... };                                // a definição da classe não é importante

MP3Player mp;

mp.checkOut( );                         // ambíguo! Qual checkOut?
```

Observe que, nesse exemplo, a chamada a `checkOut` (retirar) é ambígua, mesmo que apenas uma das duas funções esteja disponível. (`checkOut` é

pública em `BorrowableItem` – item que pode ser emprestado, mas privada em `ElectronicGadget` – dispositivo eletrônico.) Isso está de acordo com as regras de C++ para resolver as chamadas a funções sobrecarregadas; antes de ver se uma função é acessível, C++ primeiro identifica a função que combina melhor com a chamada. Ele verifica a acessibilidade apenas após encontrar a função que combina melhor. Nesse caso, ambos os `checkOut`s são boas combinações, então não existe uma combinação melhor. A acessibilidade de `ElectronicGadget::checkOut`, dessa forma, nunca é examinada.

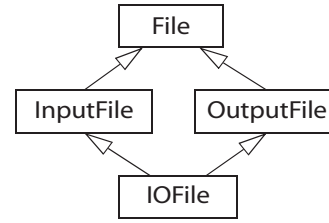
Para resolver a ambiguidade, você deve especificar de qual classe-base deve ser chamada a função:

```
mp.BorrowableItem::checkOut( );                    // ah, esse checkOut...
```

Você pode tentar chamar explicitamente `ElectronicGadget::checkOut` também, é claro, mas então o erro de ambiguidade será substituído por um erro do tipo "você está tentando chamar uma função membro privada".

A herança múltipla só significa herdar de mais de uma classe-base, porém não é raro a HM ser encontrada em hierarquias que têm classes-base de nível mais alto também. Isso pode levar ao que, às vezes, é chamado de "diamante mortal da HM".

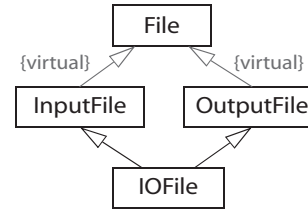
```
class File { ... };

class InputFile: public File { ... };

class OutputFile: public File { ... };

class IOFile: public InputFile,
              public OutputFile
{ ... };
```



Em qualquer momento que você tiver uma hierarquia de herança com mais de um caminho entre uma classe-base e uma classe derivada (como entre `File` – arquivo – e `IOFile` – arquivo de entrada e saída – acima, que contêm caminhos tanto por meio de `InputFile` – arquivo de entrada – quanto de `OutputFile` – arquivo de saída), você deve enfrentar a questão de querer ou não que os membros de dados na classe-base sejam replicados para cada um dos caminhos. Por exemplo, suponhamos que a classe `File` possua um membro de dados `fileName` (nome do arquivo). Quantas cópias desse campo `IOFile` deveria ter? Por um lado, ela herda uma cópia de cada uma de suas classes-base, sugerindo que `IOFile` deve ter dois membros de dados `fileName`. Por outro lado, a lógica diz que um objeto `IOFile` possui apenas um nome de arquivo; assim, o campo `fileName` que ela herda por meio de suas duas classes-base não deve ser replicado.

C++ não se posiciona nesse debate. Ela suporta feliz ambas as opções, embora seu padrão seja realizar a replicação. Se isso não for o que quer, você precisa fazer a classe com os dados (nesse caso, `File`) ser uma classe-base virtual. Para fazer isso, você precisa que todas as classes que imediatamente herdam dela usem *herança virtual*:

```
class File { ... };

class InputFile: virtual public File { ... };

class OutputFile: virtual public File { ... };

class IOFile: public InputFile,
              public OutputFile
{ ... };
```



A biblioteca padrão de C++ contém uma hierarquia múltipla como essa, com a diferença de que as classes são templates de classes, e os nomes são `basic_ios`, `basic_istream`, `basic_ostream` e `basic_iostream` em vez de `File`, `InputFile`, `OutputFile` e `IOFile`.

Do ponto de vista do comportamento correto, a herança pública deve ser sempre virtual. Se esse fosse o único ponto de vista, a regra seria simples: sempre que você usar herança pública, use herança pública *virtual*. No entanto, a correção não é a única perspectiva. Evitar a replicação de campos herdados requer algumas tarefas de bastidores da parte dos compiladores, e o resultado é que os objetos criados a partir de classes que usam herança virtual são, geralmente, maiores do que seriam sem herança virtual. O acesso aos membros de dados nas classes-base virtuais é também mais lento do que àquele nas classes-base não virtuais. Os detalhes variam de compilador para compilador, mas a força básica é clara: a herança virtual é custosa.

Existem outros custos também. As regras que governam a inicialização de classes-base virtuais são mais complicadas e menos intuitivas do que as das classes-base não virtuais. A responsabilidade para inicializar uma classe-base virtual é da *classe mais derivada* na hierarquia. As implicações desta regra incluem (1) as classes derivadas das classes-base virtuais que requerem que a inicialização esteja ciente de suas classes-base virtuais, independentemente de quão distantes as classes-base são, e, (2) quando uma nova classe derivada é adicionada à hierarquia, ela deve assumir as responsabilidades de inicialização para suas classes-base virtuais (diretas e indiretas).

Minha recomendação sobre classes-base virtuais (herança virtual) é simples. Primeiro, não use classes-base virtuais a menos que você precise fazê-lo. Por padrão, use herança não virtual. Segundo, se você precisar usar classes-base virtuais, tente evitar colocar dados nelas. Dessa maneira, você não precisa se preocupar com as estranhezas nas regras de inicialização (e, como ocorre, nas de atribuição) para essas classes. Vale a pena observar que Interfaces em Java e .NET, as quais são de muitas maneiras comparáveis às classes-base virtuais em C++, não podem conter dados.

Vamos agora nos deter à seguinte classe de Interface em C++ (veja o Item 31) para modelar pessoas:

```
class IPerson {
public:
```

```
    virtual ~IPerson( );

    virtual std::string name( ) const = 0;
    virtual std::string birthDate( ) const = 0;
};
```

Os clientes de `IPerson` (interface para `Person`) devem programar em termos de ponteiros e referências para `IPerson`, porque as classes abstratas não podem ser instanciadas. Para criar objetos que podem ser manipulados como objetos `IPerson`, os clientes de `IPerson` usam funções fábrica (mais uma vez, veja o Item 31) para instanciar as classes concretas derivadas de `IPerson`:

```
// função fábrica para criar um objeto Person a partir de um ID único da base de dados;
// veja o Item 18 para saber por que o tipo de retorno não é um ponteiro bruto
std::tr1::shared_ptr<IPerson> makePerson(DatabaseID personIdentifier);

// função para obter um ID da base de dados do usuário
DatabaseID askUserForDatabaseID( );


DatabaseID id(askUserForDatabaseID( ));

std::tr1::shared_ptr<IPerson> pp(makePerson(id));          // cria um objeto
                                                           // suportando a
                                                           // interface IPerson

...                                                        // manipula *pp via
                                                           // funções membro
                                                           // de IPerson
```

Mas como `makePerson` (criar uma pessoa) cria os objetos para os quais ela retorna ponteiros? Claramente, deve existir alguma classe concreta derivada de `IPerson` que `makePerson` possa instanciar.

Suponhamos que essa classe seja chamada de `CPerson` (pessoa concreta). Como uma classe concreta, `CPerson` deve fornecer implementações para as funções virtuais puras que ela herda de `IPerson`. Ela poderia escrever essas funções do zero, mas seria melhor tirar proveito dos componentes existentes que fazem a maioria ou todo o necessário. Por exemplo, imaginemos uma classe antiga de acesso a uma base de dados específica chamada `PersonInfo` (informações sobre uma pessoa) que ofereça a essência das necessidades de `CPerson`:

```
class PersonInfo {
public:
    explicit PersonInfo(DatabaseID pid);
    virtual ~PersonInfo( );

    virtual const char * theName( ) const;
    virtual const char * theBirthDate( ) const;
    ...

private:
    virtual const char * valueDelimOpen( ) const;          // veja
    virtual const char * valueDelimClose( ) const;         // abaixo
    ...
};
```

Você pode dizer que essa é uma classe antiga, porque as funções membro retornam `const char*` em vez dos objetos `string`. Mesmo assim, se o sapato servir, porque não usá-lo? Os nomes das funções membro dessa classe sugerem que o resultado provavelmente será bastante confortável.

Você descobrirá que `PersonInfo` foi projetada para facilitar a impressão de campos da base de dados em vários formatos, com o início e o fim do valor de cada campo delimitados por cadeias especiais. Por padrão, os delimitadores de início e fim para valores de campo são os colchetes, então o valor do campo "*Ring-tailed Lemur*" seria formatado dessa forma:

```
[Ring-tailed Lemur]
```

Em reconhecimento ao fato de que os colchetes não são universalmente desejados pelos clientes de `PersonInfo`, as funções virtuais `valueDelimOpen` (abertura de delimitador de valor) e `valueDelimClose` (fechamento de delimitador de valor) permitem que as classes derivadas especifiquem suas próprias cadeias de abertura e de fechamento de delimitadores de cadeias. As implementações das funções membro de `PersonInfo` chamam essas funções virtuais para adicionar os delimitadores apropriados para os valores que elas retornam. Usando `PersonInfo::theName` (o nome em `PersonInfo`) como exemplo, o código se pareceria com o seguinte:

```
const char * PersonInfo::valueDelimOpen( ) const
{
   return "[";                                   // delimitador padrão de abertura
}

const char * PersonInfo::valueDelimClose( ) const
{
   return "]";                                   // delimitador padrão de fechamento
}

const char * PersonInfo::theName( ) const
{
   // reserva espaço para o valor de retorno; já que ele é
   // estático, é automaticamente inicializado como zero
   static char value[Max_Formatted_Field_Value_Length];

   // escreve o delimitador de abertura
   std::strcpy(value, valueDelimOpen( ));

   adiciona à cadeia de entrada o nome do campo desse objeto (sendo cuidadoso
   para evitar transbordamentos de espaço)

   // escreve delimitador de fechamento
   std::strcat(value, valueDelimClose( ));

   return value;
}
```

Alguém poderia questionar o projeto antiquado de `PersonInfo::theName` (especialmente o uso de um buffer estático de tamanho fixo, algo que é um problema tanto para transbordamentos quanto para o uso de linhas de execução (`threads`) – veja também o Item 21), mas ponha de lado essas questões e foque-se no seguinte: `theName` chama `valueDelimOpen` para gerar o delimitador de abertura da cadeia que ela retornará; en-

tão, ela gera o valor do nome propriamente dito e depois chama `valueDelimClose`.

Como `valueDelimOpen` e `valueDelimClose` são funções virtuais, o resultado retornado por `theName` depende não apenas de `PersonInfo`, mas também das classes derivadas de `PersonInfo`.

Como implementador de `CPerson`, essa é uma boa notícia, porque, ao estudar os detalhes na documentação de `IPerson`, você descobre que `name` (nome) e `birthDate` (data de nascimento) precisam retornar valores não adornados, ou seja, não são permitidos delimitadores. Assim, se uma pessoa é chamada `Homer`, uma chamada à função `name` para essa pessoa deve retornar "Homer" e não "[Homer]".

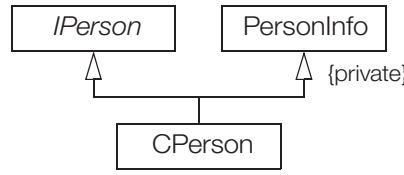
O relacionamento entre `CPerson` e `PersonInfo` é que `PersonInfo`, coincidentemente, tem algumas funções que facilitariam a implementação de `CPerson`. É isso. Logo, seu relacionamento é do tipo "é implementado em termos de", e sabemos que esses relacionamentos podem ser representados de duas maneiras: por composição (veja o Item 38) e por herança privada (veja o Item 39). O Item 39 destaca que a composição é geralmente a abordagem preferencial, mas a herança é necessária se as funções virtuais tiverem de ser redefinidas. Nesse caso, `CPerson` precisa redefinir `valueDelimOpen` e `valueDelimClose`, então uma composição simples não será suficiente. A solução mais direta é fazer `CPerson` herdar privadamente de `PersonInfo`, apesar de o Item 39 explicar que, com um pouco mais de trabalho, `CPerson` também poderia usar uma combinação de composição e de herança para redefinir as funções virtuais de `PersonInfo`. Aqui, usaremos a herança privada.

Mas `CPerson` também deve implementar a interface `IPerson`,o que chama o uso de herança pública e leva a uma aplicação racional de herança múltipla: combinar herança pública de uma interface com a herança privada de uma implementação:

```
class IPerson {                                  // esta classe especifica a
public:                                          // interface a ser implementada
  virtual ~IPerson( );

  virtual std::string name( ) const = 0;
  virtual std::string birthDate( ) const = 0;
};
class DatabaseID { ... };                        // usada abaixo; os detalhes não são
                                                 // importantes

class PersonInfo {                               // esta classe possui funções
public:                                          // úteis na implementação
  explicit PersonInfo(DatabaseID pid);           // da interface IPerson
  virtual ~PersonInfo( );

  virtual const char * theName( ) const;
  virtual const char * theBirthDate( ) const;
  ...
private:
```

```
    virtual const char * valueDelimOpen( ) const;
    virtual const char * valueDelimClose( ) const;
    ...
};
class CPerson: public IPerson, private PersonInfo {                    // observe o uso de
public:
    explicit CPerson(DatabaseID pid): PersonInfo(pid) { }

    virtual std::string name( ) const                                  // implementações HM
    { return PersonInfo::theName( ); }                                 // das funções
                                                                       // membros de
    virtual std::string birthDate( ) const                             // IPerson requeridas
    { return PersonInfo::theBirthDate( ); }

private:                                                               // funções
    const char * valueDelimOpen( ) const { return ""; }               // delimitadoras
    const char * valueDelimClose( ) const { return ""; }              // virtuais
};                                                                     // herdadas
```

Em UML, o projeto se pareceria com o seguinte:



Este exemplo demonstra que a HM pode ser tanto útil quanto compreensível.

No fim das contas, a herança múltipla é apenas outra ferramenta na caixa de ferramentas orientadas a objeto. Comparada com a herança simples, em geral ela é mais complicada de ser usada e mais complicada de ser entendida; então, se você tem um projeto com herança simples (HS) que é mais ou menos equivalente a um projeto que usa herança múltipla, o projeto que usa HS é quase sempre preferível. Se o único projeto que você consegue propor envolve HM, você deve pensar um pouco mais – quase sempre existe *alguma* maneira de fazer a HS funcionar. Ao mesmo tempo, a HM é, algumas vezes, a maneira mais clara, mais racional e de manutenção mais fácil a ser usada para que o trabalho seja realizado. Quando esse for o caso, não tenha medo de usá-la. Apenas se certifique de usá-la com bom senso.

Lembretes

- A herança múltipla é mais complexa do que a herança simples. Ela pode levar a novas questões de ambiguidade e à necessidade de herança virtual.
- A herança virtual impõe custos em termos de tamanho, velocidade e complexidade de inicialização e de atribuição. Ela é mais prática quando as classes virtuais base não têm dados.
- A herança múltipla tem usos legítimos. Um cenário envolve a combinação de herança pública de uma classe de Interface com herança privada de uma classe que ajuda na implementação.

CAPÍTULO 7

TEMPLATES E PROGRAMAÇÃO GENÉRICA

A motivação inicial para os templates em C++ era direta: possibilitar a criação de contêineres seguros em relação a tipos, como `vector`, `list` e `map` (vetor, lista e mapa). No entanto, quanto mais as pessoas trabalhavam com templates, maior era a variedade de coisas que elas descobriam que podiam fazer com eles. Os contêineres eram bons, mas a programação genérica – a habilidade de escrever código independentemente dos tipos de objetos que estão sendo manipulados – era muito melhor. Algoritmos da STL como `for_each`, `find` e `merge` (para cada, buscar e mesclar) são exemplos dessa programação. Por fim, descobriu-se que o mecanismo de templates de C++ era, ele próprio, completo em relação ao Turing: podia ser usado para calcular qualquer valor computável. Isso levou à metaprogramação de templates – a criação de programas que são executados dentro de compiladores C++ e que param de rodar quando a compilação está completa. Hoje, os contêineres são apenas uma pequena parte dos templates de C++. Apesar da amplitude das aplicações de templates, toda a programação baseada em templates se apoia em um conjunto de ideias centrais. Essas ideias são o foco deste capítulo.

Este capítulo não fará de você um programador especialista em templates, mas o tornará um programador melhor. Ele também dará as informações que você precisa para expandir suas fronteiras de programação com templates o máximo que você quiser.

Item 41: Entenda as interfaces implícitas e o polimorfismo em tempo de compilação

O mundo da programação orientada a objetos gira em torno de interfaces *explícitas* e de polimorfismo em *tempo de execução*. Por exemplo, dada a seguinte classe (sem significado),

```
class Widget {
public:
   Widget( );
   virtual ~Widget( );
```

```
    virtual std::size_t size( ) const;
    virtual void normalize( );
    void swap(Widget& other);                                    // veja o Item 25

    ...
};
```

e esta função (igualmente sem significado),

```
void doProcessing(Widget& w)
{
    if (w.size( ) > 10 && w != someNastyWidget) {
        Widget temp(w);
        temp.normalize( );
        temp.swap(w);
    }
}
```

podemos dizer o seguinte sobre `w` em `doProcessing` (realizar processamento):

- Como `w` é declarado para ser do tipo `Widget`, `w` deve suportar a interface de `Widget`. Podemos buscar a interface no código-fonte (por exemplo, no arquivo `.h` de `Widget`) para ver exatamente como ele é; então, eu chamo isso de *interface explícita* – uma interface explicitamente visível no código-fonte.

- Como algumas das funções membro de `Widget` são virtuais, as chamadas de `w` a essas funções exibirão *polimorfismo em tempo de execução*: a função específica a ser chamada será determinada em tempo de execução baseada no tipo dinâmico de `w` (veja o Item 37).

O mundo dos templates e da programação genérica é fundamentalmente diferente. Nesse mundo, as interfaces explícitas e o polimorfismo em tempo de execução continuam a existir, mas são menos importantes. Em vez disso, as *interfaces implícitas* e o *polimorfismo em tempo de compilação* movem-se para o topo. Para ver como esse é o caso, veja o que acontece quando trocamos `doProcessing` de forma que, em vez de ser uma função, ela agora é um template de função:

```
template<typename T>
void doProcessing(T& w)
{
    if (w.size() > 10 && w != someNastyWidget) {
        T temp(w);
        temp.normalize();
        temp.swap(w);
    }
}
```

Agora, o que podemos dizer a respeito de `w` em `doProcessing`?

- A interface que `w` deve suportar é determinada pelas operações realizadas em `w` no template. Nesse exemplo, parece que o

tipo de `w` (`T`) deve suportar as funções membro `size` (tamanho), `normalize` (normalizar) e `swap` (trocar), construtor de cópia (para criar `temp`) e comparação para diferença (para a comparação com `someNastyWidget` – algum Widget maldoso). Logo veremos que isso não é bem exato, mas por ora é suficientemente verdadeiro. O importante é que o conjunto de expressões que deve ser válido para que o template seja compilado é a *interface implícita* que `T` deve suportar.

- As chamadas a funções envolvendo `w` como `operator>` e `operator!=` podem envolver a instanciação de templates para que sejam bem-sucedidas. Essa instanciação ocorre durante a compilação. Como a instanciação de templates de funções com diferentes parâmetros de template leva a diferentes funções sendo chamadas, isso é conhecido como *polimorfismo em tempo de compilação*.

Mesmo que você nunca tenha usado templates, deve conhecer a diferença entre polimorfismo em tempo de execução e em tempo de compilação, pois é similar à diferença entre o processo de determinar qual função de um conjunto de funções sobrecarregadas deve ser chamado (o que acontece em tempo de compilação) e a vinculação dinâmica de chamadas a funções virtuais (que ocorre em tempo de execução). Entretanto, a diferença entre as interfaces implícitas e as explícitas é nova (introduzida pelos templates), e deve ser examinada mais de perto.

Uma interface explícita consiste em assinaturas de funções, ou seja, nomes de funções, tipos de parâmetros, tipos de retorno, etc. A interface pública da classe `Widget`, por exemplo,

```
class Widget {
public:
  Widget( );
  virtual ~Widget( );

  virtual std::size_t size( ) const;
  virtual void normalize( );
  void swap(Widget& other);
};
```

consiste em um construtor, em um destrutor e nas funções `size`, `normalize` e `swap`, juntamente com os tipos de parâmetros, tipos de retorno e a constância dessas funções. (Ela também inclui o construtor de cópia e o operador de atribuição por cópia gerados pelo compilador – veja o Item 5). Ela também pode incluir definições de tipos (`typedef`s) e – se você foi tão corajoso a ponto de violar a recomendação do Item 22 de criar membros de dados privados – membros de dados, embora, neste caso, não exista nenhum declarado.

Uma interface implícita é um tanto diferente. Ela não se baseia em assinaturas de funções, consiste em *expressões* válidas. Olhe novamente a expressão condicional no início do template `doProcessing`:

```
template<typename T>
void doProcessing(T& w)
{
  if (w.size( ) > 10 && w != someNastyWidget) {
  ...
```

A interface implícita para `T` (o tipo de `w`) parece ter as seguintes restrições:

- Deve oferecer uma função membro chamada `size` que retorna um valor inteiro.
- Deve suportar uma função `operator!=` que compara dois objetos do tipo `T`. (Aqui, consideramos que `someNastyWidget` é do tipo `T`.)

Graças à possibilidade de sobrecarga de operadores, nenhuma dessas restrições precisa ser satisfeita. Sim, `T` deve suportar uma função membro `size`, embora valha a pena mencionar que uma função pode ser herdada de uma classe-base. Mas essa função membro não precisa retornar um tipo inteiro; ela não precisa nem mesmo retornar um tipo numérico. Para essa questão, ela não precisa se quer retornar um tipo para o qual `operator>` seja definido! Tudo o que ela precisa é retornar um objeto de um tipo `X` tal que exista um `operator>` que possa ser chamado com um objeto do tipo `X` e um `int` (porque 10 é do tipo `int`). O `operator>` não precisa receber um parâmetro do tipo `X`, porque poderia receber um parâmetro do tipo `Y`, e isso seria permitido desde que existisse uma conversão implícita de objetos do tipo `X` para objetos do tipo `Y`!

De um modo similar, não existe nenhum requisito que diga que `T` deve suportar `operator!=`, porque seria tão aceitável quanto para `operator!=` receber um objeto do tipo `X` e um objeto do tipo `Y`. Desde que `T` possa ser convertido em `X` e o tipo de `someNastyWidget` possa ser convertido em `Y`, a chamada a `operator!=` seria válida.

(Como comentário, essa análise não leva em conta a possibilidade de `operator&&` poder ser sobrecarregado, modificando o significado da expressão acima de uma conjunção para algo potencialmente bastante diferente.)

A cabeça da maioria das pessoas começa a doer quando elas começam a pensar, pela primeira vez, nas interfaces implícitas dessa maneira, mas não chega a ser um caso para uma aspirina. As interfaces implícitas são simplesmente feitas de um conjunto de expressões válidas. As expressões propriamente ditas podem parecer complicadas, mas as restrições que elas impõem geralmente são diretas. Por exemplo, dada a expressão condicional

```
if (w.size( ) > 10 && w != someNastyWidget) ...
```

é difícil dizer muita coisa a respeito das restrições nas funções `size`, `operator>`, `operator&&` ou `operator!=`, mas é fácil identificar a restrição na expressão como um todo. A parte condicional de uma sentença `if` deve ser uma expressão booleana, então, independentemente dos tipos exatos envolvidos, qualquer valor oriundo de

"`w.size( ) > 10 && w != someNastyWidget`" deve ser compatível com `bool`. Isso faz parte da interface implícita que `doProcessing` impõe em seu parâmetro de tipo `T`. O resto da interface requerida por `doProcessing` é que as chamadas ao construtor de cópia, a `normalize` e a `swap` devem ser válidas para objetos do tipo `T`.

As interfaces implícitas impostas nos parâmetros de um template são tão reais quanto as interfaces explícitas impostas nos objetos de uma classe, e ambas são verificadas durante a compilação. Assim como você não pode usar um objeto de uma maneira contraditória com a interface explícita que sua classe oferece (o código não compilará), você não pode tentar usar um objeto em um template, a menos que esse objeto suporte a interface implícita que o template requer (mais uma vez, o código não será compilado).

Lembretes

- Tanto as classes quanto os templates suportam interfaces e polimorfismo.
- Para as classes, as interfaces são explícitas e centradas em assinaturas de funções. O polimorfismo ocorre em tempo de execução por meio de funções virtuais.
- Para os parâmetros de templates, as interfaces são implícitas e baseadas em expressões válidas. O polimorfismo ocorre durante a compilação por meio da instanciação de templates e por meio da resolução de sobrecarga de funções.

Item 42: Entenda os dois significados de `typename`

Pergunta: qual é a diferença entre `class` (classe) e `typename` (nome de tipo) nas seguintes declarações de templates?

```
template<class T> class Widget;                    // usa "class"
template<typename T> class Widget;                 // usa "typename"
```

Resposta: nenhuma. Quando estiver declarando um parâmetro de tipo de template, `class` e `typename` significam exatamente a mesma coisa. Alguns programadores preferem usar `class` todo o tempo, porque é mais fácil de digitar. Outros (incluindo eu) preferem `typename`, porque sugere que o parâmetro não precisa ser um tipo classe (ou seja, pode ser um tipo predefinido). Alguns poucos desenvolvedores empregam `typename` quando for permitido qualquer tipo e reservam `class` para o momento em que apenas os tipos definidos pelo usuário são aceitáveis. Mas, do ponto de vista de C++, `class` e `typename` significam exatamente a mesma coisa quando você estiver declarando um parâmetro de template.

Entretanto, C++ nem sempre vê `class` e `typename` como equivalentes. Algumas vezes, você precisa usar `typename`. Para entender quando, precisamos falar sobre os dois tipos de nomes aos quais você pode fazer referência em um template.

Vamos supor que temos um template para uma função que recebe um contêiner compatível com a STL que mantém objetos que possam ser atribuídos para `int`s. Além disso, suponhamos que essa função simplesmente imprima o valor de seu segundo elemento. É uma função tola implementada de maneira tola, e, da maneira como a escrevi abaixo, ela não deve nem mesmo ser compilada, mas, por favor, ignore essas coisas, pois existe um método em minha loucura:

```
template<typename C>                          // imprime o 2º elemento no
void print2nd(const C& container)             // contêiner;
{                                             // isso não é C++ válido!
   if (container.size( ) >= 2) {
      C::const_iterator iter(container.begin( ));   // obtém um iterador para o 1º elemento
      ++iter;                                 // copia esse elemento para um int
      int value = *iter;                      // copia esse elemento para um int
      std::cout << value;                     // imprime o int
   }
}
```

Destaquei as duas variáveis locais nessa função, `iter` (iterador) e `value` (valor). O tipo de `iter` é `C::const_iterator`, um tipo que depende do parâmetro de template `C`. Os nomes em um template que dependem de um parâmetro de template são chamados de *nomes dependentes*. Quando um nome dependente é aninhado dentro de uma classe, eu o chamo de *nome dependente aninhado*. `C::const_iterator` é um nome dependente aninhado. Na verdade, é um *nome de tipo dependente aninhado*, ou seja, um nome dependente aninhado que se refere a um tipo.

A outra variável local em `print2nd` (imprime 2º), `value`, possui o tipo `int`. `int` é um nome que não depende de nenhum parâmetro de template. Esses nomes são conhecidos como *nomes não dependentes*. (Não tenho ideia de por que não são chamados de nomes independentes. Se, como eu, você achar o termo "não dependente" uma aberração, tem minha simpatia, mas "não dependente" é o termo para esses tipos de nomes; então, tal como eu, revire seus olhos e resigne-se a ele.)

Os nomes dependentes aninhados podem levar a dificuldades na análise sintática. Por exemplo, supor que deixamos `print2nd` ainda mais tola iniciando da seguinte forma:

```
template<typename C>
void print2nd(const C& container)
{
   C::const_iterator * x;
   ...
}
```

Parece que estamos declarando `x` como uma variável local que é um ponteiro para um `C::const_iterator`, mas isso acontece apenas porque "sabemos" que `C::const_iterator` é um tipo. Mas, e se `C::const_iterator` não fosse um tipo? E se `C` tivesse um membro de dados estático que coincidentemente fosse chamado de `const_iterator`, e se `x` coincidentemente ti-

vesse o nome de uma variável global? Nesse caso, o código acima não declararia uma variável local: seria uma multiplicação de `C::const_iterator` por `x`! Claro, parece loucura, mas é *possível*, e as pessoas que escrevem analisadores sintáticos C++ precisam se preocupar com todos os tipos de entrada, mesmo as mais malucas.

Até `C` ser conhecido, não é possível saber se `C::const_iterator` é um tipo ou não; quando o template `print2nd` é analisado sintaticamente, `C` ainda não é conhecido. C++ tem uma regra para resolver essa ambiguidade: se o analisador sintático encontrar um nome dependente aninhado em um template, ele entende que o nome *não* é um tipo, a menos que você diga o contrário. Por padrão, os nomes dependentes aninhados *não* são tipos. (Existe uma exceção a essa regra, que analisarei daqui a pouco.)

Com isso em mente, olhe mais uma vez para o início de `print2nd`:

```
template<typename C>
void print2nd(const C& container)
{
  if (container.size( ) >= 2) {
    C::const_iterator iter(container.begin( ));        // considera-se que este nome
    ...                                                // não seja um tipo
```

Agora deve estar claro porque esse não é um código C++ válido. A declaração de `iter` faz sentido apenas se `C::const_iterator` for um tipo, mas não dissemos a C++ que ele é, e C++ considera que não é. Para corrigir essa situação, precisamos dizer a C++ que `C::const_iterator` é um tipo. Fazemos isso ao colocarmos `typename` imediatamente na frente dele:

```
template<typename C>                                   // isso é C++ válido
void print2nd(const C& container)
{
  if (container.size( ) >= 2) {
    typename C::const_iterator iter(container.begin( ));

    ...

  }
}
```

A regra geral é simples: sempre que você se referir a um nome de tipo dependente aninhado, deve precedê-lo pela palavra `typename`. (Mais uma vez, descreverei uma exceção em breve.)

Deve-se usar a palavra `typename` para identificar apenas os nomes de tipos dependentes aninhados; outros nomes não devem tê-la. Por exemplo, veja um template de função que recebe tanto um contêiner quanto um iterador para esse contêiner:

```
template<typename C>                    // typename permitido (assim como "class")
void f(const C& container,              // typename não permitido
       typename C::iterator iter);      // typename exigido
```

`C` não é um nome de tipo dependente aninhado (não é aninhado dentro de nada dependente de um parâmetro de template), então ele não precisa ser precedido por `typename` quando estiver declarando `conteiner`. `C::iterator`, por outro lado, é um nome de tipo dependente aninhado, então precisa ser precedido por `typename`.

A exceção à regra "`typename` deve preceder nomes de tipos dependentes aninhados" é que `typename` não deve preceder os nomes de tipos dependentes aninhados em uma lista de classes-base ou como identificador de uma classe-base em uma lista de inicialização de membros. Por exemplo:

```
template<typename T>
class Derived: public Base<T>::Nested {          // lista de classes-base: typename não
public:                                          // permitido
   explicit Derived(int x)
   : Base<T>::Nested(x)                          // identificador da classe-base em mem.
   {                                             // lista de inic.: typename não permitido
      typename Base<T>::Nested temp;             // uso de tipo dependente aninhado
      ...                                        // o nome não está em uma lista de classes-base
   }                                             // nem como identificador de uma classe-base
   ...                                           // em uma lista de inic. de
                                                 // mem.: typename necessário
};
```

Essas inconsistências são desagradáveis, mas depois que ganhar um pouco de experiência, raramente as notará.

Vamos examinar um último exemplo de `typename`, porque ele é representativo de algo que você verá em código real. Suponhamos que você esteja escrevendo um template de função que receba um iterador, e queremos fazer uma cópia local, `temp`, do objeto para o qual o iterador aponta. Poderíamos fazer isso da seguinte forma:

```
template<typename IterT>
void workWithIterator(IterT iter)
{
   typename std::iterator_traits<IterT>::value_type temp(*iter);
   ...
}
```

Não deixe que `std::iterator_traits<IterT>::value_type` o assuste. É apenas um uso de uma classe padrão de traits (veja o Item 47), a maneira pela qual C++ diz que "o tipo de algo apontado por objetos do tipo `IterT`". A sentença declara uma variável local (`temp`) do mesmo tipo para o qual os objetos `IterT` apontam. Se `IterT` for do tipo `vector<int>::iterator`, `temp` é do tipo `int`. Se `IterT` for do tipo `list<string>::iterator`, `temp` é do tipo `string`. Como `std::iterator_traits<IterT>::value_type` é um nome de tipo dependente aninhado (`value_type` é aninhado dentro de `iterator_traits<IterT>` e `IterT` é um parâmetro de template), devemos precedê-lo com `typename`.

Se você pensa que ler `std::iterator_traits<IterT>::value_type` é desagradável, imagine o que é digitá-lo. Se, como a maioria dos programadores, você acha que só de pensar em digitá-lo mais de uma vez já é horripilante, então vai querer criar um `typedef`. Para os nomes de membros abstratos como `value_type` (mais uma vez, veja o Item 47 para mais informações sobre traits), uma convenção comum é que o nome da definição de tipo seja igual ao nome do membro de traits, então essa definição de tipo local é frequentemente definida como segue:

```
template<typename IterT>
void workWithIterator(IterT iter)
{
  typedef typename std::iterator_traits<IterT>::value_type value_type;
  value_type temp(*iter);
  ...
}
```

Muitos programadores acham que a justaposição "`typedef typename`" é inicialmente irritante, mas é uma consequência lógica das regras que se referem a nomes de tipos dependentes aninhados. Você vai se acostumar a elas rapidamente, afinal, tem uma forte motivação. Quantas vezes você vai querer digitar `typename std::iterator_traits<IterT>::value_type`?

Como nota de fechamento, devo mencionar que a garantia de cumprimento das regras relacionadas a `typename` variam de um compilador para outro. Alguns compiladores aceitam código no qual `typename` se faz necessário, mas está faltando; alguns aceitam código no qual `typename` está presente, mas não é permitido; e uns poucos (normalmente os mais antigos) rejeitam `typename` quando está presente e é necessário. Isso significa que a interação de `typename` com os nomes de tipos dependentes aninhados pode levar a algumas dores de cabeça consideráveis em relação à portabilidade de código.

Lembretes

- Quando estiver declarando parâmetros de template, `class` e `typename` são intercambiáveis.
- Use `typename` para identificar os nomes de tipos dependentes aninhados, exceto em listas de classes-base ou como identificador de classe-base em uma lista de inicialização de membros.

Item 43: Saiba como acessar nomes em classes-base com templates

Suponha que precisamos escrever um aplicativo que possa enviar mensagens a diversas empresas diferentes. As mensagens podem ser enviadas ou criptografadas ou em texto plano (sem criptografia). Se tivermos informação suficiente durante a compilação para determinar quais mensagens irão para quais empresas, podemos empregar uma solução baseada em templates:

```
class CompanyA {
public:
   ...
   void sendCleartext(const std::string& msg);
   void sendEncrypted(const std::string& msg);
   ...
};

class CompanyB {
public:
   ...
   void sendCleartext(const std::string& msg);
   void sendEncrypted(const std::string& msg);
   ...
};

...                                          // classes para outras companhias

class MsgInfo { ... };                       // classe para manter informações
                                             // usada para criar uma mensagem

template<typename Company>
class MsgSender {
public:
   ...                                       // construtor, destrutor, etc.

   void sendClear(const MsgInfo& info)
   {
      std::string msg;
      cria mensagem a partir da informação;

      Company c;
      c.sendCleartext(msg);
   }

   void sendSecret(const MsgInfo& info)      // similar a sendClear, exceto que
   { ... }                                   // chama c.sendEncrypted

};
```

Isso funcionará bem, mas suponhamos que, às vezes, quiséssemos manter em um log algumas informações cada vez que enviássemos uma mensagem. Uma classe derivada poderia facilmente adicionar essa capacidade, e parece ser uma maneira racional de fazer isso:

```
template<typename Company>
class LoggingMsgSender: public MsgSender<Company> {
public:
   ...                                       // construtores, destrutor, etc.
   void sendClearMsg(const MsgInfo& info)
   {
      escreve informações "antes de enviar" no log

      sendClear(info);                       // chama a função da classe-base;
                                             // esse código não será compilado!

      escreve informações "após enviar" no log
   }
   ...
};
```

Observe como a função de envio de mensagens na classe derivada tem um nome diferente (`sendClearMsg` – envia mensagem em texto plano) daquela em sua classe-base (lá, ela é chamada de `sendClear`). Esse é um bom projeto, pois evita o problema de ocultar nomes herdados (veja o Item 33), bem como os problemas inerentes que ocorrem ao redefinir uma função não virtual herdada (veja o Item 36). Mas o código acima não será compilado, pelo menos não com compiladores que estejam em conformidade com a gramática de C++. Esses compiladores reclamarão que `sendClear` não existe. Podemos ver que `sendClear` está na classe-base, mas os compiladores não a procurarão lá. Precisamos entender por quê.

O problema é que, quando os compiladores encontram a definição para o template de classe `LoggingMsgSender` (mantenedor de informações de log de envio de mensagens), eles não sabem de que classe herdam. Claro, é de `MsgSender<Company>` (enviador de mensagens), mas `Company` (empresa) é um parâmetro de template que não será conhecido até mais tarde (quando `LoggingMsgSender` for instanciado). Sem saber o que `Company` é, não é possível saber como a classe `MsgSender<Company>` se parece. Em particular, não é possível saber se ela tem uma função `sendClear`.

Para tornar o problema concreto, suponha que tivéssemos uma classe `CompanyZ` (empresa Z) que insistisse em usar comunicações criptografadas:

```
class CompanyZ {                                    // essa classe não oferece
public:                                             // uma função sendCleartext
   ...
   void sendEncrypted(const std::string& msg);
   ...
};
```

O template `MsgSender` geral é inapropriado para `CompanyZ` porque oferece uma função `sendClear` que não faz sentido para os objetos da classe `CompanyZ`. Para corrigir esse problema, podemos criar uma versão especializada de `MsgSender` para `CompanyZ`:

```
template<>                                          // uma especialização total de
class MsgSender<CompanyZ> {                         // MsgSender; igual ao template
public:                                             // geral, com a diferença de que
   ...                                              // sendCleartext é omitido
   void sendSecret(const MsgInfo& info)
   { ... }

};
```

Observe a sintaxe "`template <>`", no início dessa definição de classe. Isso significa que ele não é nem um template, nem uma classe propriamente dita. Em vez disso, é uma versão especializada do template `MsgSender` para ser usada quando o argumento do template for `CompanyZ`. Isso é conhecido como *especialização de* template *total*: o *template* `MsgSender` é especializado para o tipo `CompanyZ`, e a especialização é *total* – uma vez que o parâmetro de tipo tenha sido definido como `CompanyZ`, nenhum outro aspecto dos parâmetros do template pode variar.

Como `MsgSender` foi especializado para `CompanyZ`, considere mais uma vez a classe derivada `LoggingMsgSender`:

```
template<typename Company>
class LoggingMsgSender: public MsgSender<Company> {
public:
  ...

  void sendClearMsg(const MsgInfo& info)
  {
    escreve informações "antes de enviar" ao log

    sendClear(info);                          // se Company == CompanyZ,
                                              // esta função não existe!
    escreve informações "após enviar" ao log
  }

  ...
};
```

Como diz o comentário, esse código não faz sentido quando a classe-base for `MsgSender<CompanyZ>`, porque essa classe não oferece uma função `sendClear`. É por isso que C++ rejeita a chamada: a linguagem reconhece que os templates da classe-base podem ser especializados e que essas especializações podem não oferecer a mesma interface que o template geral. Como resultado, ele geralmente se recusa a examinar classes-base com templates em busca de nomes herdados. Em certo sentido, quando cruzamos de C++ Orientado a Objetos para o C++ com Templates (veja o Item 1), a herança para de funcionar.

Para reinicializá-la, precisamos, de alguma forma, desabilitar o comportamento "não procure em classes-base com templates" de C++. Existem três maneiras de fazer isso. Primeiro, você pode adicionar "`this->`" ao início das chamadas a funções da classe-base:

```
template<typename Company>
class LoggingMsgSender: public MsgSender<Company> {
public:
  ...

  void sendClearMsg(const MsgInfo& info)
  {
    escreve informações "antes de enviar" ao log

    this->sendClear(info);                    // ótimo, considera que
                                              // sendClear será herdado
    escreve informações "após enviar" ao log
  }

  ...
};
```

Segundo, você pode empregar uma declaração `using`, uma solução que deve soar familiar se você já leu o Item 33. Esse Item explica como as declarações `using` levam nomes de classes-base ocultos ao escopo de uma classe derivada. Podemos, então, escrever `sendClearMsg` como:

```
template<typename Company>
class LoggingMsgSender: public MsgSender<Company> {
public:
  using MsgSender<Company>::sendClear;          // diz para os compiladores considerarem
  ...                                           // que sendClear está na
                                                // classe-base
  void sendClearMsg(const MsgInfo& info)
  {
    ...
    sendClear(info);                            // bom, considera que
    ...                                         // sendClear será herdada
  }

  ...
};
```

(Embora uma declaração `using` funcione tanto aqui quanto no Item 33, os problemas que estão sendo solucionados são diferentes. Aqui, a situação não é que os nomes da classe-base são ocultados por nomes da classe derivada, e sim que os compiladores não buscam nos escopos das classes-base a não ser que digamos para que o façam.)

Uma terceira maneira para fazer seu código compilar é especificar explicitamente que a função que está sendo chamada está na classe-base:

```
template<typename Company>
class LoggingMsgSender: public MsgSender<Company> {
public:
  ...
  void sendClearMsg(const MsgInfo& info)
  {
    ...
    MsgSender<Company>::sendClear(info);        // ótimo, considera que
    ...                                         // sendClear será
  }                                             // herdado

  ...
};
```

Essa é geralmente a maneira menos desejável de resolver o problema, porque, se a função chamada é virtual, a qualificação explícita desliga o comportamento de vinculação virtual.

Do ponto de vista da visibilidade de nomes, cada uma dessas abordagens faz a mesma coisa: promete aos compiladores que quaisquer especializações subsequentes no template da classe-base suportarão a interface oferecida pelo template geral. Essa promessa é tudo o que os compiladores

precisam quando analisam sintaticamente um template de uma classe derivada como `LoggingMsgSender`, mas, se ela for infundada, a verdade surgirá durante as compilações subsequentes. Por exemplo, se o código-fonte posteriormente contiver

```
LoggingMsgSender<CompanyZ> zMsgSender;

MsgInfo msgData;
...                                    // coloca informações em msgData

zMsgSender.sendClearMsg(msgData);      // erro! não será compilado
```

a chamada a `sendClearMsg` não será compilada, porque, nesse ponto, os compiladores sabem que a classe-base é a especialização de template `MsgSender<CompanyZ>`, e eles sabem que essa classe não oferece a função `sendClear`, a qual `sendClearMsg` está tentando chamar.

Fundamentalmente, a questão é se os compiladores vão diagnosticar referências inválidas a membros das classes-base mais cedo (quando as definições de templates da classe derivada são analisadas sintaticamente) ou mais tarde (quando esses templates são instanciados com argumentos de template específicos). A política de C++ é preferir diagnósticos precoces, e é por isso que ele considera que não sabe nada sobre os conteúdos das classes-base quando elas são instanciadas a partir de templates.

Lembrete

» Em templates de classes derivadas, refira-se aos nomes nos templates da classe-base por meio de um prefixo "`this->`", por meio de declarações `using`, ou de uma qualificação explícita da classe-base.

Item 44: Fatore código independente de parâmetros a partir de templates

O uso de templates é uma maneira maravilhosa de economizar tempo e evitar replicação de código. Em vez de digitar 20 classes similares, cada uma com 15 funções membro, você digita um template de classe e deixa que os compiladores instanciem as 20 classes específicas e as 300 funções que você precisa. (As funções membro de templates de classe são implicitamente instanciadas apenas quando usadas, então você obterá todas as 300 funções membro apenas se cada uma delas for realmente usada.) Os templates de funções são igualmente interessantes. Em vez de escrever várias funções, você escreve um template de função e deixa que os compiladores façam o resto. A tecnologia não é magnífica?

Sim, bem... às vezes. Se você não for cuidadoso, o uso de templates pode levar a um *inchaço de código*: códigos binários com código replicado (ou quase totalmente replicado), dados replicados, ou ambos. O resultado pode ser códi-

go-fonte que parece estar com tudo em cima, mas cujo código-objeto é gordo e flácido. Esse código-objeto raramente é elegante, então você precisa saber como evitar essa fanfarrice binária.

Sua ferramenta principal tem o nome pomposo de *análise de similaridade e de variabilidade*, mas não há nada de pomposo sobre a ideia dela. Mesmo que você nunca tenha escrito um template em sua vida, você faz essa análise o tempo todo.

Quando você está escrevendo uma função e se dá conta que parte da implementação da função é, essencialmente, igual à implementação de outra função, você simplesmente replica o código? É claro que não. Você fatora o código comum a partir das duas funções, coloca-o em uma terceira função e faz as outras duas funções chamarem a nova. Ou seja, você analisa as duas funções para encontrar as partes que são comuns e as partes que variam, então move as partes comuns para uma nova função e mantém as partes variáveis nas funções originais. De modo similar, se você está escrevendo uma classe e se dá conta de que algumas partes da classe são iguais a partes de outra classe, você não replica as partes comuns. Em vez disso, move as partes comuns para uma nova classe e, então, usa herança ou composição (veja os Itens 32, 38 e 39) para dar às classes originais acesso aos recursos comuns. As partes diferentes das classes originais – as que variam – permanecem em seus locais originais.

Quando está escrevendo templates, você realiza a mesma análise e evita a replicação da mesma forma, mas aqui há uma mudança. Em código não template, a replicação é explícita: você pode *ver* que existe duplicação entre duas funções ou entre duas classes. Em código com templates, a replicação é implícita: existe apenas uma cópia do código-fonte do template, então você precisa treinar a si mesmo para perceber a replicação que pode ocorrer quando um template for instanciado múltiplas vezes.

Por exemplo, suponhamos que você queira escrever um template para matrizes quadradas de tamanho fixo que, dentre outras coisas, suporte a inversão de matrizes.

```
template<typename T,                    // template para matrizes n x n de
         std::size_t n>                 // objetos do tipo T; veja abaixo para obter mais info
class SquareMatrix {                    // no parâmetro size_t
public:
  ...
  void invert( );                       // inverte a matriz no local
};
```

Esse template recebe um parâmetro de tipo, `T`, mas também recebe um parâmetro do tipo `size_t` (tamanho de `T`) – um *parâmetro que não é um tipo*. Os parâmetros que não são tipos são menos comuns do que os parâmetros de tipos, mas são completamente legais e, como nesse exemplo, podem ser bem naturais.

Agora, considere o código a seguir:

```
SquareMatrix<double, 5> sm1;
...
sm1.invert( );                    // chama SquareMatrix<double, 5>::invert

SquareMatrix<double, 10> sm2;
...
sm2.invert( );                    // chama SquareMatrix<double, 10>::invert
```

Duas cópias de `invert` (inverter) serão instanciadas aqui. As funções não serão idênticas, porque uma funcionará em matrizes 5 x 5 e outra em matrizes 10 x 10, mas, fora as constantes 5 e 10, as duas funções serão iguais. Essa é uma maneira clássica de provocar um inchaço de código induzido por templates.

O que você deve fazer se vir duas funções que eram idênticas, caractere a caractere, exceto pelo uso de 5 em uma versão e 10 em outra? Seu instinto seria criar uma versão que recebe um valor como parâmetro e, então, chama a função parametrizada com 5 ou 10 em vez de replicar o código. Seu instinto está correto! Veja uma primeira tentativa de fazer isso para `SquareMatrix` (matriz quadrada):

```
template<typename T>                      // classe-base independente de tamanho para
class SquareMatrixBase {                  // matrizes quadradas
protected:
   ...
   void invert(std::size_t matrixSize);   // inverte uma matriz de um determinado tamanho
   ...
};

template<typename T, std::size_t n>
class SquareMatrix: private SquareMatrixBase<T> {
private:
   using SquareMatrixBase<T>::invert;     // torna a versão classe-base de invert
                                          // invisível nesta classe; veja os Itens 33
                                          // e 43
public:
   ...
   void invert( ) { invert(n); }          // faz chamadas internalizadas à classe-base
};                                        // versão de invert
```

Como você pode ver, a versão parametrizada de `invert` está em uma classe-base, `SquareMatrixBase` (matriz quadrada base). Como `SquareMatrix`, `SquareMatrixBase` é um template, mas, diferentemente de `SquareMatrix`, ela possui parâmetros de template apenas no tipo dos objetos da matriz, não no tamanho da matriz. Logo, todas as matrizes que mantêm um dado tipo de objeto compartilham uma mesma classe `SquareMatrixBase`. Elas compartilharão, então, uma só cópia da versão de `invert` dessa classe.

`SquareMatrixBase::invert` pretende ser apenas uma maneira pela qual as classes derivadas podem evitar replicação de código, então ela é pro-

tegida (`protected`) em vez de ser pública (`public`). O custo adicional de chamá-la deve ser zero, porque os `invert`s das classes derivadas chamam a versão da classe-base usando funções internalizadas (a internalização [`inline`] é implícita – veja o Item 30). Observe também que a herança entre `SquareMatrix` e `SquareMatrixBase` é privada. Isso reflete, precisamente, o fato de que a razão para a existência da classe-base é apenas facilitar a implementação das classes derivadas, e não expressar um relacionamento "é um(a)" entre `SquareMatrix` e `SquareMatrixBase`. (Para obter mais informações sobre a herança privada, veja o Item 39.)

Até agora tudo bem, mas existe uma questão delicada de que ainda não tratamos. Como `SquareMatrixBase::invert` sabe com quais dados deve operar? Ela sabe o tamanho da matriz a partir de seus parâmetros, mas como sabe onde os dados para uma matriz em particular estão? Aparentemente, apenas as classes derivadas sabem isso. Como elas comunicam isso à classe-base de forma que ela possa fazer a inversão?

Uma possibilidade seria adicionar outro parâmetro a `SquareMatrixBase::invert`, talvez um ponteiro para o início de uma parte da memória em que os dados da matriz estejam. Isso funcionaria, mas com certeza, `invert` não é a única função em `SquareMatrix` que pode ser escrita de maneira independente de tamanho e que foi movida em `SquareMatrixBase`. Se existirem diversas funções assim, tudo o que precisaremos é uma maneira de encontrar a memória que mantém os valores na matriz. Poderíamos adicionar um parâmetro extra a todas elas, mas estaríamos dizendo a `SquareMatrixBase` a mesma informação repetidamente. Isso parece errado.

Uma alternativa é fazer `SquareMatrixBase` armazenar um ponteiro para a memória para os valores da matriz. E, já que ela está armazenando esse ponteiro, poderia muito bem armazenar o tamanho da matriz. O projeto resultante se parece com o seguinte:

```
template<typename T>
class SquareMatrixBase {
protected:
  SquareMatrixBase(std::size_t n, T *pMem)          // armazena o tamanho da matriz e um
  : size(n), pData(pMem) { }                        // ponteiro para os valores da matriz

  void setDataPtr(T *ptr) { pData = ptr; }          // reatribui pData
  ...

private:
  std::size_t size;                                 // tamanho da matriz

  T *pData;                                         // ponteiro para valores da matriz
};
```

Isso deixa as classes derivadas decidirem como alocar a memória. Algumas implementações podem decidir armazenar os dados da matriz dentro do objeto `SquareMatrix`:

```
template<typename T, std::size_t n>
class SquareMatrix: private SquareMatrixBase<T> {
public:
  SquareMatrix( )                          // envia o tamanho da matriz e
  : SquareMatrixBase<T>(n, data) { }       // ponteiro de dados para a classe-base
  ...

private:
  T data[n*n];
};
```

Os objetos desse tipo não têm a necessidade de alocação de memória dinâmica, mas podem ser muito grandes. Uma alternativa seria colocar os dados para cada matriz no monte (heap):

```
template<typename T, std::size_t n>
class SquareMatrix: private SquareMatrixBase<T> {
public:
  SquareMatrix( )                          // configura o ptr de dados da classe-base para null
  :  SquareMatrixBase<T>(n, 0),            // aloca memória para os valores
     pData(new T[n*n])                     // da matriz, salva um ponteiro para a
  { this->setDataPtr(pData.get( )); }      // memória, e dá uma cópia dele
  ...                                      // para a classe-base

private:
  boost::scoped_array<T> pData;            // veja o Item 13 para informações sobre
};                                         // boost::scoped_array
```

Independentemente de onde os dados estão armazenados, o resultado principal de um ponto de vista inchado é que agora muitas – talvez todas – funções membro de `SquareMatrix` podem ser chamadas internalizadas simples para as versões da classe-base que são compartilhadas com todas as outras matrizes que mantêm o mesmo tipo de dados, independentemente de seu tamanho. Ao mesmo tempo, os objetos `SquareMatrix` de diferentes tamanhos são tipos distintos, então, mesmo que, por exemplo, `SquareMatrix<double, 5>` e `SquareMatrix<double, 10>` usem as mesmas funções membro de `SquareMatrixBase<double>`, não é possível passar um `SquareMatrix<double, 5>` para uma função que espera um `SquareMatrix<double, 10>`. Legal, não?

Legal, sim, mas não é de graça. As versões de `invert` com os tamanhos inseridos manualmente e de maneira fixa provavelmente geram um código melhor do que a versão compartilhada na qual o tamanho é passado como parâmetro de função ou é armazenado no objeto. Por exemplo, nas versões de tamanho específico, os tamanhos seriam constantes de tempo de compilação, logo, elegíveis para otimizações como propagação de constantes, incluindo o fato de serem inseridas nas instruções geradas como operandos imediatos. Isso não pode ser feito na versão independente de tamanho.

Por outro lado, ter apenas uma versão de `invert` para múltiplos tamanhos de matrizes diminui o tamanho do executável, e isso pode reduzir

o tamanho do conjunto de trabalho do programa e melhorar a referência de localidade na cache de instruções. Essas coisas podem aumentar a velocidade de execução de seu programa, compensando quaisquer perdas de otimização em versões de tamanho específico de `invert`. Qual efeito predominaria? A única maneira de saber é experimentando ambas as formas e observando o comportamento em sua plataforma particular e em conjuntos de dados representativos.

Outra consideração de eficiência está relacionada ao tamanho dos objetos. Se você não for cuidadoso, mover para cima (em direção a uma classe-base) versões independentes de tamanho de funções pode aumentar o tamanho geral de cada objeto. Por exemplo, no código que acabei de mostrar, cada objeto `SquareMatrix` possui um ponteiro para seus dados na classe `SquareMatrixBase`, mesmo que cada classe derivada já tenha uma maneira de chegar aos dados. Isso aumenta o tamanho de cada objeto `SquareMatrix` no tamanho de um ponteiro no mínimo. É possível modificar o projeto de forma que esses ponteiros sejam desnecessários, mas, mais uma vez, existe aí um dilema. Por exemplo, fazer a classe-base armazenar um ponteiro protegido (`protected`) para os dados da matriz leva à perda de encapsulamento descrita no Item 22. Isso também pode levar a complicações de gerenciamento de recursos: se a classe-base armazena um ponteiro para os dados da matriz, mas esses dados podem ter sido alocados dinamicamente ou armazenados fisicamente dentro do objeto da classe derivada (de acordo com o que vimos), como poderemos determinar se o ponteiro deve ser apagado? Essas questões têm respostas, mas, quanto mais sofisticadas forem as respostas, mais complicadas as coisas se tornam. Chega certo ponto em que um pouco de duplicação de código parece ser uma bênção.

Este item discutiu apenas o inchaço devido a parâmetros de template que não são tipos, mas os parâmetros de tipo também podem provocar inchaço. Por exemplo, em muitas plataformas, os tipos de dados `int` e `long` possuem a mesma representação binária, então as funções membro para, digamos, `vector<int>` e `vector<long>` tendem a ser idênticas – a definição exata de inchaço. Alguns ligadores unirão as implementações de funções idênticas, mas outros não farão isso, o que deve dizer que alguns templates instanciados tanto para `int` quanto para `long` poderão causar inchaço de código em alguns ambientes. De maneira similar, na maioria das plataformas, todos os tipos ponteiro possuem a mesma representação binária, então os templates que mantêm tipos ponteiro (como, por exemplo, `list<int*>`, `list<const int*>`, `list<SquareMatrix<long,3>*>`, etc) costumam ser capazes de usar uma única implementação subjacente para cada função membro. Em geral, isso significa implementar funções membro que funcionem com ponteiros fortemente tipados (ou seja, ponteiros `T*`) ao fazê-los chamar funções que funcionam com ponteiros sem tipos (ou seja, `void*`). Algumas implementações da biblioteca padrão de C++ fazem isso para templates como `vector`, `deque` e `list`. Se você está preocupado

com o inchaço de código em seus templates, provavelmente quer desenvolver templates que façam o mesmo.

Lembretes

» Os templates geram múltiplas classes e múltiplas funções, então qualquer código que não seja dependente de um parâmetro de template causa inchaço.

» O inchaço devido a parâmetros de template sem tipo normalmente pode ser eliminado ao substituirmos os parâmetros de template por parâmetros de funções ou por membros de dados de classes.

» O inchaço devido a parâmetros de tipo pode ser reduzido ao compartilharmos implementações para a instanciação de tipos com representações binárias idênticas.

Item 45: Use templates de funções membro para aceitar "todos os tipos compatíveis"

Os *ponteiros espertos* são objetos que agem de maneira bastante parecida com ponteiros, mas que adicionam funcionalidades que os ponteiros não fornecem. Por exemplo, o Item 13 explica como `auto_ptr` e `tr1::shared_ptr` podem ser usados para apagar automaticamente recursos baseados no monte no momento certo. Os iteradores em contêineres STL são quase sempre ponteiros espertos; certamente, você não pode esperar mover um ponteiro predefinido de um nó em uma lista encadeada para o outro usando "++", mas isso funciona para os iteradores de listas (`list::iterator`s).

Uma das coisas que os ponteiros fazem bem é oferecer suporte a conversões implícitas. Os ponteiros das classes derivadas implicitamente são convertidos em ponteiros da classe-base, os ponteiros para objetos não constantes são convertidos em ponteiros para objetos constantes, etc. Por exemplo, considere algumas conversões que podem ocorrer em uma hierarquia de três níveis:

```
class Top { ... };
class Middle: public Top { ... };
class Bottom: public Middle { ... };
Top *pt1 = new Middle;                  // converte Middle* ⇒ Top*
Top *pt2 = new Bottom;                  // converte Bottom* ⇒ Top*
const Top *pct2 = pt1;                  // converte Top* ⇒ const Top*
```

Emular essas conversões em classes de ponteiros espertos definidas pelo usuário pode ser complicado. Precisamos que o seguinte código seja compilado:

```
template<typename T>
class SmartPtr {
public:                                   // os ponteiros espertos em geral são
   explicit SmartPtr(T *realPtr);         // inicializados por ponteiros predefinidos
   ...
};
SmartPtr<Top> pt1 =                       // converte SmartPtr<Middle> ⇒
   SmartPtr<Middle>(new Middle);          //   SmartPtr<Top>

SmartPtr<Top> pt2 =                       // converte SmartPtr<Bottom> ⇒
   SmartPtr<Bottom>(new Bottom);          //   SmartPtr<Top>

SmartPtr<const Top> pct2 = pt1;           // converte SmartPtr<Top> ⇒
                                          //   SmartPtr<const Top>
```

Não existe um relacionamento inerente entre diferentes instanciações do mesmo template, então, os compiladores veem `SmartPtr<Middle>` e `SmartPtr<Top>` (ambos ponteiros espertos) como classes completamente diferentes, menos fortemente relacionadas que, digamos, `vector<float>` e `Widget`. Para obter as conversões entre as classes `SmartPtr` que queremos, precisamos programá-las explicitamente.

No código de exemplo de ponteiros espertos citado, cada sentença cria um novo objeto do tipo ponteiro esperto; assim, por enquanto, nos focaremos em como escrever construtores de ponteiros espertos que se comportem da maneira que queremos. Uma observação importante é que não é possível escrever todos os construtores de que precisamos. Na hierarquia acima, podemos construir um `SmartPtr<Top>` a partir de um `SmartPtr<Middle>` ou de um `SmartPtr<Bottom>`, mas, se a hierarquia for estendida no futuro, os objetos `SmartPtr<Top>` precisarão ser passíveis de serem construídos a partir de outros tipos de ponteiros espertos. Por exemplo, se posteriormente adicionássemos:

```
class BelowBottom: public Bottom { ... };
```

precisaríamos suportar a criação de objetos `SmartPtr<Top>` a partir de objetos `SmartPtr<BelowBottom>`, e, certamente, não vamos querer precisar modificar o template `SmartPtr` para fazer isso.

Em princípio, o número de construtores de que precisamos é ilimitado. Como um template pode ser instanciado para gerar um número ilimitado de funções, parece que não precisamos de uma *função* construtora para `SmartPtr`, e sim de um template de construção. Esses templates são exemplos de *templates de funções membro* (muitas vezes conhecidos apenas como *templates de membros*) – templates que geram funções membro de uma classe:

```
template<typename T>
class SmartPtr {
public:
   template<typename U>                   // template de membro
   SmartPtr(const SmartPtr<U>& other);    // para um "construtor de cópia
   ...                                    // generalizado"
};
```

Isso diz que, para cada tipo `T` e para cada tipo `U`, pode-se criar um `SmartPtr<T>` a partir de um `SmartPtr<U>`, porque `SmartPtr<T>` tem um construtor que recebe um parâmetro `SmartPtr<U>`. Construtores como esse – que criam um objeto a partir de outro objeto cujo tipo é uma instanciação diferente do mesmo template (ou seja, cria um `SmartPtr<T>` a partir de um `SmartPtr<U>`) – são, algumas vezes, conhecidos como *construtores de cópia generalizados*.

O construtor de cópia generalizado mostrado não é declarado como explícito (`explicit`). Isso é deliberado. Conversões de tipo entre tipos ponteiros predefinidos (por exemplo, a partir de ponteiros de classes derivadas para ponteiros de classes-base) são implícitas, e não exigem uma conversão explícita (cast), então, é racional que os ponteiros espertos emulem esse comportamento. Omitir `explicit` no construtor com templates faz justamente isso.

Como está declarado, o construtor de cópia generalizado para `SmartPtr` oferece mais do que queremos. Sim, queremos poder criar um `SmartPtr<Top>` a partir de um `SmartPtr<Bottom>`, mas não queremos poder criar um `SmartPtr<Bottom>` a partir de um `SmartPtr<Top>`, já que isso é contrário à herança pública (veja o Item 32). Também não queremos poder criar um `SmartPtr<int>` a partir de um `SmartPtr<double>`, porque não existe uma conversão implícita correspondente de `int*` para `double*`. Precisamos, de alguma forma, selecionar o conjunto de funções membro que esse template de membro gerará.

Considerando que `SmartPtr` segue o caminho de `auto_ptr` e de `tr1::shared_ptr` ao oferecer uma função membro `get` que retorna uma cópia do ponteiro predefinido mantido pelo objeto do tipo do ponteiro esperto (veja o Item 15), podemos usar a implementação do template de construtor para restringir as conversões apenas àquelas que queremos:

```
template<typename T>
class SmartPtr {
public:
  template<typename U>
  SmartPtr(const SmartPtr<U>& other)          // inicializa este ponteiro mantido
  : heldPtr(other.get( )) { ... }             // com o ponteiro mantido por outro

  T* get( ) const { return heldPtr; }
  ...

private:                                      // ponteiro predefinido mantido
  T *heldPtr;                                 // pelo SmartPtr
};
```

Usamos a lista de inicialização de membros para inicializar os membros de dados do tipo `T*` de `SmartPtr<T>` com o ponteiro do tipo `U*` mantido pelo `SmartPtr<U>`. Isso será compilado apenas se existir uma conversão implícita a partir de um ponteiro `U*` para um ponteiro `T*`, e isso é exatamente o que queremos. O efeito em cascata é que agora `SmartPtr<T>` possui um construtor de cópia generalizado que será compilado apenas se for passado um parâmetro de um tipo compatível.

A utilidade dos templates de funções membro não é limitada aos construtores. Outra regra comum para eles é em relação ao suporte à atribuição. Por exemplo, o `shared_ptr` de TR1 (mais uma vez, veja o Item 13) suporta a construção para todos os tipos de ponteiros predefinidos compatíveis, `tr1::shared_ptr`s, `auto_ptr`s e `tr1::weak_ptr`s (veja o Item 54, bem como atribuições de todos esses, exceto de `tr1::weak_ptr`s). Veja um excerto da especificação de TR1 para `tr1::shared_ptr`, incluindo sua preferência para usar `class` em vez de `typename` quando estiver declarando parâmetros de template. (Como o Item 42 explica, eles significam exatamente a mesma coisa nesse contexto).

```
template<class T> class shared_ptr {
public:
  template<class Y>                                    // constrói a partir de
    explicit shared_ptr(Y * p);                        // qualquer ponteiro
  template<class Y>                                    // predefinido compatível,
    shared_ptr(shared_ptr<Y> const& r);                // shared_ptr,
  template<class Y>                                    // weak_ptr, ou
    explicit shared_ptr(weak_ptr<Y> const& r);         // auto_ptr
  template<class Y>
    explicit shared_ptr(auto_ptr<Y>& r);
  template<class Y>                                    // atribui a partir de
    shared_ptr& operator=(shared_ptr<Y> const& r);     // qualquer
  template<class Y>                                    // shared_ptr ou
    shared_ptr& operator=(auto_ptr<Y>& r);             // auto_ptr compatíveis
  ...
};
```

Todos esses construtores são explícitos (`explicit`), exceto o construtor de cópia generalizado. Isso significa que as conversões implícitas de um tipo de `shared_ptr` para outro são permitidas, mas as conversões *implícitas* de um ponteiro predefinido ou de outro tipo de ponteiro esperto não são permitidas (as conversões *explícitas* – por meio de um cast – são permitidas). Também interessante é como os `auto_ptr`s passados para os construtores de `tr1::shared_ptr` e os operadores de atribuição não são declarados como constantes (através de `const`), em contraste a como os `tr1::shared_ptr`s e `tr1::weak_ptr`s são passados. Essa é uma consequência do fato de que `auto_ptr`s podem ser modificados independentemente quando são copiados (veja o Item 13).

Os templates de funções membro são ótimos, mas não alteram as regras básicas da linguagem. O Item 5 explica que duas das quatro funções membro que os compiladores podem gerar são o construtor de cópias e o operador de atribuição por cópia. `tr1::shared_ptr` declara um construtor de cópia generalizado, e é claro que, quando os tipos `T` e `Y` são iguais, o construtor de cópia generalizado pode ser instanciado para criar o construtor de cópia "normal". Então, será que os compiladores gerarão um construtor de cópia para `tr1::shared_ptr`, ou instanciarão o template de construtor de cópia generalizado quando um objeto `tr1::shared_ptr` for construído a partir de outro objeto `tr1::shared_ptr` do mesmo tipo?

Como já disse, os templates de membros não modificam as regras da linguagem, e as regras dizem que, se um construtor de cópia é necessário e você não declara um, será gerado um para você automaticamente. Declarar um construtor de cópia generalizado (um template de membro) em uma classe não impede que os compiladores gerem seu próprio construtor de cópia (sem templates); então, se você quer controlar todos os aspectos da construção de cópias, deve declarar tanto um construtor de cópia generalizado quanto um construtor de cópia "normal". O mesmo se aplica à atribuição. Veja um trecho extraído da definição de `tr1::shared_ptr` que exemplifica isso:

```
template<class T> class shared_ptr {
public:
  shared_ptr(shared_ptr const& r);                    // construtor de cópia

  template<class Y>                                   // construtor de cópia
    shared_ptr(shared_ptr<Y> const& r);               // generalizado

  shared_ptr& operator=(shared_ptr const& r);         // atribuição de cópia

  template<class Y>                                   // atribuição de cópia
    shared_ptr& operator=(shared_ptr<Y> const& r);    // generalizada
  ...
};
```

Lembretes

» Use templates de funções membro para gerar funções que aceitem todos os tipos compatíveis.

» Se você declara templates de membros para a construção de cópias generalizada ou a atribuição generalizada, ainda assim precisa declarar também o construtor de cópia e o operador de atribuição por cópia.

Item 46: Defina funções não membro dentro de templates quando desejar conversões de tipo

O Item 24 explica por que apenas as funções não membro são candidatas a conversões de tipo implícitas em todos os argumentos, e usa como exemplo a função `operator*` para uma classe de números racionais (`Rational`). Recomendo que você estude esse exemplo antes de continuar, pois este item amplia a discussão com uma modificação aparentemente inofensiva ao exemplo do Item 24: ele usa templates tanto em `Rational` quanto em `operator*`:

```
template<typename T>
class Rational {
public:
  Rational(const T& numerator = 0,          // veja o Item 20 para saber por que os
           const T& denominator = 1);       // parâmetros agora são passados por referência

  const T numerator( ) const;               // veja o Item 28 para saber por que os valores
  const T denominator( ) const;             // de retorno ainda são passados por valor,
  ...                                       // e o Item 3 para saber por que são constantes

};
```

```
template<typename T>
const Rational<T> operator*(const Rational<T>& lhs,
                            const Rational<T>& rhs)
{ ... }
```

Como no Item 24, queremos suportar aritmética de modo misto, então queremos que o código a seguir seja compilado. Esperamos que isso aconteça, pois estamos usando o mesmo código que funciona no Item 24. A única diferença é que `Rational` e `operator*` são templates agora:

```
Rational<int> oneHalf(1, 2);          // este exemplo é do Item 24,
                                      // com a diferença de que Rational é agora um template

Rational<int> result = oneHalf * 2;   // erro! não será compilado
```

O fato de esse código não ser compilado sugere que há algo diferente entre a classe `Rational` com templates e a versão sem templates, e de fato há. No Item 24, os compiladores sabem qual função estamos tentando chamar (`operator*` recebendo dois `Rational`s), mas aqui os compiladores *não* sabem que função queremos chamar. Em vez disso, eles estavam tentando *descobrir* que função instanciar (ou seja, criar) a partir do template denominado `operator*`. Eles sabem que, supostamente, devem instanciar alguma função chamada `operator*` que recebe dois parâmetros do tipo `Rational<T>`, mas, para poder fazer a instanciação, eles precisam descobrir o que é `T`. O problema é que eles não podem.

Ao tentar deduzir `T`, eles procuram nos tipos dos argumentos passados na chamada a `operator*`. Nesse caso, os tipos são `Rational<int>` (o tipo de `oneHalf` – uma metade) e `int` (o tipo de 2). Cada parâmetro é considerado separadamente.

A dedução usando `oneHalf` é fácil. O primeiro parâmetro de `operator*` é declarado como do tipo `Rational<T>`, e o primeiro argumento passado para `operator*` é do tipo `Rational<int>`, então `T` deve ser `int`. Infelizmente, a dedução para o outro parâmetro não é tão simples. O segundo parâmetro para `operator*` é declarado como do tipo `Rational<T>`, mas o segundo argumento passado para `operator* (2)` é do tipo `int`. Como os compiladores descobrem o que é `T` nesse caso? Você talvez esperasse que eles usassem o construtor não explícito de `Rational<int>` para converter 2 em um `Rational<int>`, permitindo que deduzissem que `T` é `int`, mas eles não fazem isso. Eles não fazem porque as funções de conversão de tipos implícitas *nunca* são consideradas durante a dedução de argumentos de templates. Nunca. Essas conversões são usadas durante as chamadas a funções, sim, mas, antes de você chamar uma função, precisa saber quais funções existem. Para saber isso, você precisa deduzir os tipos de parâmetros para os *templates* de função relevantes (de forma que você possa instanciar as funções apropriadas). Mas a conversão de tipos implícita por meio de chamadas a construtores não é considerada durante a dedução de argumentos dos templates. O Item 24 não envolve template algum, então a dedução de argumentos de templates não era algo a ser considerado. Agora que estamos na parte de templates de C++ (veja o Item 1), essa é a questão fundamental.

Podemos liberar os compiladores do desafio de deduzir os argumentos de template ao aproveitarmos o fato de uma declaração amiga (`friend`) em uma classe de template poder se referir a uma função específica. Isso significa que a classe `Rational<T>` pode declarar `operator*` para `Rational<T>` como função amiga.

Os templates de classes não dependem da dedução de argumentos de templates (esse processo se aplica apenas a templates de funções), então `T` sempre é conhecido no momento em que a classe `Rational<T>` é instanciada. Isso faz ser fácil para a classe `Rational<T>` declarar a função `operator*` apropriada como amiga:

```
template<typename T>
class Rational {
public:
...
friend                                                  // declara função
   const Rational operator*(const Rational& lhs,        // operator* (veja
                            const Rational& rhs);       // abaixo para mais detalhes)
};

template<typename T>                                    // declara funções
const Rational<T> operator*(const Rational<T>& lhs,     // operator*
                            const Rational<T>& rhs)
{ ... }
```

Agora, nossas chamadas de modo misto para `operator*` serão compiladas, porque, quando o objeto `oneHalf` é declarado como do tipo `Rational<int>`, a classe `Rational<int>` é instanciada e, como parte desse processo, a função amiga `operator*` que recebe parâmetros `Rational<int>` é automaticamente declarada. Como *função* declarada (não um template de função), os compiladores podem usar funções de conversão implícitas (como o construtor não explícito de `Rational`) quando a chamam; é assim que elas fazem a chamada de modo misto ser bem-sucedida.

Aliás, "bem-sucedida" é uma expressão engraçada nesse contexto, porque, apesar de o código ser compilado, ele não ligará. Veremos isso daqui a pouco, mas, primeiro, eu gostaria de destacar a sintaxe usada para declarar `operator*` dentro de `Rational`.

Dentro de um template de classe, o nome do template pode ser usado como atalho para o template e para seus parâmetros, então, dentro de `Rational<T>`, poderíamos escrever `Rational` em vez de `Rational<T>`. Isso economiza apenas alguns poucos caracteres nesse exemplo, mas, quando existem múltiplos parâmetros ou nomes de parâmetros mais longos, isso pode tanto economizar a digitação quanto resultar em código mais claro. Eu trouxe isso à tona porque `operator*` é declarada como recebendo e retornando `Rational`s em vez de `Rational<T>`s. Teria sido igualmente válido declarar `operator*` como segue:

```
template<typename T>
class Rational {
public:
   ...

friend
   const Rational<T> operator*(const Rational<T>& lhs,
                               const Rational<T>& rhs);
   ...
};
```

Entretanto, é mais fácil (e mais comum) usar a forma abreviada.

Agora, voltemos ao problema da ligação. O código de modo misto é compilado, porque os compiladores sabem que queremos chamar uma função específica (`operator*` recebendo um `Rational<int>` e um `Rational<int>`), mas essa função é apenas *declarada* dentro de `Rational`, não *definida* lá. Nossa intenção é que o template `operator*` fique fora da classe que fornece essa definição, mas as coisas não funcionam dessa maneira. Se declararmos, nós mesmos, uma função (que é o que estamos fazendo dentro do template `Rational`), somos também responsáveis por definir essa função. Nesse caso, nunca fornecemos uma definição, e é por isso que os ligadores não conseguem encontrar nenhuma.

A coisa mais simples que, possivelmente, poderia funcionar é mesclar o corpo de `operator*` à sua declaração:

```
template<typename T>
class Rational {
public:
   ...

friend const Rational operator*(const Rational& lhs, const Rational& rhs)
{
   return Rational( lhs.numerator( ) * rhs.numerator( ),          // mesma implementação
                    lhs.denominator( ) * rhs.denominator( ));     // como no
}                                                                 // Item 24
};
```

De fato, isso funciona como pretendido: as chamadas de modo misto a `operator*` agora são compiladas, ligadas e executadas. Viva!

Uma observação interessante sobre essa técnica é que o uso de amizade não tem nada a ver com a necessidade de acessar partes não públicas da classe. Para que as conversões em todos os argumentos sejam possíveis, precisamos de uma função não membro (o Item 24 ainda se aplica), e para ter a função apropriada instanciada automaticamente, precisamos declarar a função dentro da classe. A única maneira de declarar uma função não membro dentro de uma classe é torná-la amiga. Então, é isso o que fazemos. Pouco convencional? Sim. Eficaz? Sem dúvida.

Como o Item 30 explica, as funções definidas dentro de uma classe são declaradas implicitamente como `inline`, e isso inclui funções amigas

como `operator*`. Você pode minimizar o impacto dessas declarações internalizadas (`inline`) com `operator*` não fazendo nada exceto chamando uma função ajudante definida fora da classe. No exemplo deste item, não há muitos motivos para fazer isso, já que `operator*` já é implementada como uma função de uma linha, mas, para corpos de funções mais complexos, isso pode ser desejado. Vale a pena dar uma olhada na abordagem "faça com que o amigo chame o ajudante".

O fato de `Rational` ser um template significa que a função ajudante normalmente será também um template, então, o código no arquivo de cabeçalho que define `Rational` ficará da seguinte forma:

```
template<typename T> class Rational;                          // declara
                                                              // template
                                                              // Rational
template<typename T>                                          // declara
const Rational<T> doMultiply( const Rational<T>& lhs,         // template
                              const Rational<T>& rhs);        // ajudante

template<typename T>
class Rational {
public:
  ...

friend
  const Rational<T> operator*(const Rational<T>& lhs,
                              const Rational<T>& rhs)         // amiga
  { return doMultiply(lhs, rhs); }                            // chama ajudante
  ...
};
```

Muitos compiladores essencialmente obrigam que você coloque todas as definições de template em arquivos de cabeçalho, então você pode precisar definir `doMultiply` (realizar multiplicação) em seu cabeçalho também. (Como explica o Item 30, esses templates não precisam ser internalizados.) Isso poderia se parecer com o seguinte:

```
template<typename T>                                          // define
const Rational<T> doMultiply(const Rational<T>& lhs,          // template
                             const Rational<T>& rhs)          // ajudante em
{                                                             // arquivo de cabeçalho,
return Rational<T>(lhs.numerator( ) * rhs.numerator( ),       // se necessário
                   lhs.denominator( ) * rhs.denominator( ));
}
```

Como template, `doMultiply` não suportará a multiplicação de modo misto, é claro, mas ela não precisa fazer isso. Ela será chamada apenas por `operator*`, e `operator*` *suporta* operações de modo misto! Em resumo, a *função* `operator*` suporta quaisquer conversões de tipo que sejam necessárias para garantir que dois objetos `Rational` sejam multiplicados, então ela passa esses dois objetos para uma instanciação apropriada do template `doMultiply` para realizar a multiplicação real. Sinergia em ação, não?

Lembrete

» Quando estiver escrevendo um template de classe que ofereça funções relacionadas ao template que suporta conversões de tipo implícitas em todos os parâmetros, defina essas funções como amigas dentro do template de classe.

Item 47: Use classes de trait para informações sobre tipos

A STL é feita principalmente de templates para contêineres, iteradores e algoritmos, mas também possui alguns templates utilitários. Um deles é chamado de `advance` (avançar), que move um iterador especificado em uma distância especificada:

```
template<typename IterT, typename DistT>          // avança iter d
void advance(IterT& iter, DistT d);               // unidades; se d < 0,
                                                  // retrocede iter
```

Conceitualmente, `advance` só faz `iter += d`, mas `advance` não pode ser implementada dessa maneira, porque apenas os iteradores de acesso aleatório suportam a operação `+=`. Os tipos iteradores menos poderosos precisam implementar `advance` aplicando iterativamente `++` ou `--` `d` vezes.

Hum, você não se lembra de suas categorias de iteradores STL? Sem problemas, faremos uma minirrevisão. Existem cinco categorias de iteradores, que correspondem às operações que suportam. Os *iteradores de entrada* podem apenas avançar, se mover apenas um passo por vez, ler apenas aquilo para o que apontam e ler aquilo para o que apontam apenas uma vez. Eles são modelados com base no ponteiro de leitura para um arquivo de entrada; a biblioteca `istream_iterator` de C++ é representativa dessa categoria. Os *iteradores de saída* são análogos, mas para saída: podem apenas retroceder, se mover apenas um passo por vez, escrever apenas aquilo para o que apontam e apenas uma vez. Eles são modelados com base no ponteiro de escrita para um arquivo de saída; `ostream_iterator` é a epítome dessa categoria. Essas são as duas categorias de iteradores menos poderosas. Como os iteradores de entrada e de saída podem apenas avançar e ler ou escrever aquilo para o que apontam no máximo uma vez, são adequados só para algoritmos de uma única passada.

Uma categoria de iteradores mais poderosa é a dos iteradores avante (*forward iterators*). Esses iteradores podem fazer tudo o que os iteradores de entrada e de saída podem fazer, além de poderem ler ou escrever aquilo para o que apontam mais de uma vez. Isso torna esses iteradores viáveis para os algoritmos de múltiplas passadas. A STL não oferece lista alguma simplesmente encadeada, mas algumas bibliotecas oferecem uma (normalmente chamada de `slist`), e os iteradores nesses contêineres são iteradores avante. Os iteradores em contêineres com hash de TR1 (veja o Item 54) também podem estar na categoria avante.

Os *iteradores bidimensionais* adicionam aos iteradores avante a habilidade de retroceder. Os iteradores para `list` da STL estão nessa categoria, assim como os iteradores para `set`, `multiset`, `map` e `multimap`.

A categoria de iteradores mais poderosa é dos *iteradores de acesso aleatório*. Esse tipo de iteradores adiciona aos iteradores bidirecionais a habilidade de realizar "aritmética de iteradores", ou seja, avançar ou retroceder uma distância arbitrária em tempo constante. Essa aritmética é análoga à aritmética de ponteiros, o que não chega a ser surpreendente, dado que os iteradores de acesso aleatório são modelados como ponteiros predefinidos, e os ponteiros predefinidos podem agir como iteradores de acesso aleatório. Os iteradores para `vector`, `deque` e `string` são de acesso aleatório.

Para cada uma das cinco categorias de iteradores, C++ tem uma "estrutura de etiqueta" (tag struct) na biblioteca padrão que serve para identificá-las:

```
struct input_iterator_tag { };

struct output_iterator_tag { };

struct forward_iterator_tag: public input_iterator_tag { };

struct bidirectional_iterator_tag: public forward_iterator_tag { };

struct random_access_iterator_tag: public bidirectional_iterator_tag { };
```

Os relacionamentos de herança entre essas estruturas são relacionamentos "é um(a)" válidos (veja o Item 32): é verdade que todos os iteradores avante são também iteradores de entrada, etc. Veremos a utilidade dessa herança em breve.

Mas vamos voltar a `advance`. Dadas as diferentes capacidades dos iteradores, uma maneira de implementar `advance` seria usar a estratégia do menor denominador comum, ou seja, um laço que incrementa ou decrementa iterativamente o iterador. Entretanto, essa abordagem levaria um tempo linear. Os iteradores de acesso aleatório suportam aritmética de iteradores em tempo constante, e gostaríamos de aproveitar essa habilidade quando ela estiver presente.

O que realmente queremos fazer é implementar `advance` essencialmente como o seguinte:

```
template<typename IterT, typename DistT>
void advance(IterT& iter, DistT d)
{
    if (iter é um iterador de acesso aleatório) {
        iter += d;                                  // usa aritmética de iteradores
    }                                               // para iteradores de acesso aleatório
    else {
        if (d >= 0) { while (d--) ++iter; }         // usa chamadas iterativas
        else { while (d++) --iter; }                // ++ ou -- para outras
    }                                               // categorias de iteradores
}
```

Isso exige que seja possível determinar se `iter` é um iterador de acesso aleatório, o que, por sua vez, requer saber se seu tipo, `iterT`, é um tipo de iterador de acesso aleatório. Em outras palavras, precisamos obter alguma informação acerca de um tipo. É isso que os traits o deixam fazer: eles permitem que você obtenha informações acerca de um tipo durante a compilação.

Os traits não são uma palavra-chave ou uma construção predefinida em C++ – eles são uma técnica e uma convenção seguidas pelos programadores C++. Uma das demandas feitas para essa técnica é que ela precisa funcionar tão bem para tipos predefinidos quanto funciona para tipos definidos pelo usuário. Por exemplo, se `advance` é chamada com um ponteiro (como `const char*`) e um `int`, `advance` precisa funcionar, mas isso significa que a técnica de traits deve funcionar para tipos predefinidos, como ponteiros.

O fato de que traits devem funcionar com tipos predefinidos significa que coisas como aninhar informações dentro de tipos não funcionarão, porque não é possível aninhar informações dentro de ponteiros. A informação de traits para um tipo, então, deve ser externa ao tipo. A técnica padrão é colocá-la em um template e em uma ou mais especializações do template. Para iteradores, o template na biblioteca padrão é chamado de `iterator_traits`:

```
template<typename IterT>                    // template para informações sobre
struct iterator_traits;                     // tipos de iteradores
```

Como você pode ver, `iterator_traits` é uma estrutura. Por convenção, os traits são sempre implementados como estruturas. Outra convenção é que as estruturas usadas para implementar traits são conhecidas como – eu não estou inventando isso – *classes* de traits.

A maneira pela qual `iterator_traits` funciona é tal que, para cada tipo `IterT`, uma definição de tipo chamada `iterator_category` é declarada na estrutura `iterator_traits<IterT>`. Essa definição de tipo identifica a categoria do iterador de `IterT`.

A estrutura `iterator_traits` implementa isso em duas partes. Primeiro, ela impõe o requisito de que qualquer tipo de iterador definido pelo usuário deve conter uma definição de tipo aninhada chamada `iterator_category`, que identifica a estrutura de tag apropriada. Os iteradores de deque são de acesso aleatório, por exemplo, então, uma classe para iteradores de deque se pareceria com seguinte:

```
template < ... >                            // parâmetros de template omitidos
class deque {
public:
  class iterator {
  public:
    typedef random_access_iterator_tag iterator_category;
    ...
  };
  ...
};
```

Entretanto, os iteradores de `list` são bidirecionais, então, eles fazem as coisas desta forma:

```
template < ... >
class list {
public:
   class iterator {
   public:
      typedef bidirectional_iterator_tag iterator_category;
      ...
   };
   ...
};
```

A estrutura `iterator_traits` só repete a definição de tipo aninhada da classe do iterador:

```
// o iterator_category para o tipo IterT é qualquer coisa que IterT disser que é;
// veja o Item 42 para mais informações sobre o uso de "typedef typename"
template<typename IterT>
struct iterator_traits {
   typedef typename IterT::iterator_category iterator_category;
   ...
};
```

Isso funciona bem para tipos definidos pelo usuário, mas não funciona de jeito algum para iteradores que são ponteiros, pois não existe um ponteiro com uma definição de tipos aninhada. A segunda parte da implementação de `iterator_traits` trata de iteradores que são ponteiros.

Para suportar esses iteradores, `iterator_traits` oferece uma *especialização de template parcial* para tipos ponteiro. Os ponteiros agem como iteradores de acesso aleatório, então essa é a categoria que `iterator_traits` especifica para eles:

```
template<typename IterT>                    // especialização de template parcial
struct iterator_traits<T*>                  // para tipos ponteiro predefinidos
{
   typedef random_access_iterator_tag iterator_category;
   ...
};
```

Neste ponto, você já sabe como projetar e implementar uma classe de `trait`:

- Identifique alguma informação sobre tipos que você gostaria de disponibilizar (por exemplo, para iteradores, suas categorias de iteração).
- Escolha um nome para identificar essa informação (por exemplo, `iterator_category`).
- Forneça um template e um conjunto de especializações (por exemplo, `iterator_traits`) que contenha a informação para os tipos que você quer oferecer suporte.

Dado `iterator_traits` – na verdade `std:: iterator_traits`, uma vez que faz parte da biblioteca padrão de C++ –, podemos redefinir nosso pseudocódigo para `advance`:

```
template<typename IterT, typename DistT>
void advance(IterT& iter, DistT d)
{
  if (typeid(typename std::iterator_traits<IterT>::iterator_category) ==
      typeid(std::random_access_iterator_tag))
  ...
}
```

Apesar de isso parecer promissor, não é o que queremos. Por um lado, isso nos trará problemas de compilação, mas exploraremos isso no Item 48; agora, existe uma questão mais fundamental a ser considerada. O tipo de `IterT` é conhecido durante a compilação, então `iterator_traits<IterT>::iterator_category` pode também ser determinada durante a compilação. Mesmo assim, a sentença `if` é avaliada em tempo de execução. Por que fazer algo em tempo de execução que poderíamos fazer durante a compilação? Perde-se tempo (literalmente), e isso incha nosso executável.

O que realmente queremos é uma construção condicional (ou seja, uma sentença `if...else`) para tipos que é avaliada durante a compilação. Acontece que C++ já tem uma maneira de obter esse comportamento, chamada de sobrecarga (overloading). Quando você sobrecarrega uma função `f`, especifica diferentes tipos de parâmetros para diferentes sobrecargas. Quando chama `f`, os compiladores pegam a melhor sobrecarga com base nos argumentos que você está passando. Os compiladores, em sua essência, dizem: "Se esta sobrecarga é a melhor correspondência para o que está sendo passado, chame esse `f`; se esta outra sobrecarga é a melhor corre, chame-a; e se a terceira é a melhor, chame-a", etc. Viu? Uma construção condicional para tipos em tempo de compilação. Para fazer `advance` comportar-se da maneira que queremos, tudo o que precisamos fazer é criar duas versões de uma função sobrecarregada contendo o principal de `advance`, declarando cada uma delas para receber um tipo diferente de objeto `iterator_category`. Eu uso o nome `doAdvance` (realizar o avanço) para essas funções:

```
template<typename IterT, typename DistT>                    // use esta implementação para
void doAdvance(IterT& iter, DistT d,                        // iteradores
               std::random_access_iterator_tag)             // de acesso aleatório
{
  iter += d;
}

template<typename IterT, typename DistT>                    // use esta implementação para
void doAdvance(IterT& iter, DistT d,                        // iteradores
               std::bidirectional_iterator_tag)             // de acesso bidirecional
{
  if (d >= 0) { while (d--) ++iter; }
  else { while (d++) --iter; }
}

template<typename IterT, typename DistT>                    // use esta implementação para
```

```
void doAdvance(IterT& iter, DistT d,                    // iteradores de entrada
               std::input_iterator_tag)
{
  if (d < 0 ) {
    throw std::out_of_range("Negative distance");       // veja abaixo
  }
  while (d--) ++iter;
}
```

Como `forward_iterator_tag` herda de `input _iterator_tag`, a versão de `doAdvance` para `input_iterator_tag` também manipulará iteradores avante. Essa é a motivação para a herança entre as várias estruturas `iterator_tag`. (Na verdade, isto faz parte da motivação para *todas* as heranças públicas: ser capaz de escrever código para tipos da classe-base que também funcione para tipos da classe derivada.)

A especificação de `advance` permite tanto distâncias positivas quanto negativas para iteradores de acesso aleatório e bidirecionais, mas o comportamento é indefinido se você tentar mover um iterador avante ou um iterador de entrada usando uma distância negativa. As implementações que verifiquei simplesmente consideraram que `d` era não negativo, entrando em um laço de contagem *muito* longo, "descendo" até zero se uma distância negativa fosse passada como entrada. No código mostrado, indiquei uma exceção sendo lançada. Ambas as implementações são válidas. Essa é a sina do comportamento indefinido: você *não pode prever* que ele acontecerá.

Dadas as várias sobrecargas para `doAdvance`, tudo o que `advance` precisa fazer é chamá-las, passando um objeto extra para o tipo de categoria de iterador apropriado de modo que o compilador use a resolução de sobrecarga para chamar a implementação apropriada:

```
template<typename IterT, typename DistT>
void advance(IterT& iter, DistT d)
{
  doAdvance(                                            // chama a versão
    iter, d,                                            // de doAdvance
    typename                                            // que é
      std::iterator_traits<IterT>::iterator_category( ) // apropriada para
  );                                                    // a categoria de iteração
}                                                       // de iter
```

Podemos agora resumir como usar uma classe de trait:

- Crie um conjunto de funções ou de templates de funções "trabalhadoras" sobrecarregadas (como `doAdvance`) que se distinguem entre si por um parâmetro de trait. Implemente cada função de acordo com a informação de trait passada.
- Crie uma função ou template de função "mestre" (como `advance`) que chame as trabalhadoras, passando as informações fornecidas por uma classe de trait.

Os traits são amplamente usados na biblioteca padrão. Existe `iterator_traits`, é claro, o qual, além de `iterator_category`, oferece quatro informações extras sobre iteradores (a mais útil delas é `value_type` – o Item 42 mostra um exemplo de seu uso). Existe também `char_traits`, que mantém informações sobre tipos de caracteres, e `numeric_limits`, que fornece informações sobre tipos numéricos, como seus valores representativos mínimos e máximos, etc. (O nome `numeric_limits` é um pouco surpreendente, porque a convenção mais comum é que as classes de trait terminem com "traits", mas como esse é o nome dela, então é assim que a chamamos.)

TR1 (veja o Item 54) introduz várias classes de traits novas que dão informações sobre tipos, incluindo `is_fundamental<T>` (que diz se `T` é um tipo predefinido), `is_array<T>` (que diz se `T` é um tipo vetor) e `is_base_of<T1, T2>` (que diz se `T1` é o mesmo tipo ou se é uma classe-base de `T2`). No fim das contas, TR1 adiciona cerca de 50 classes de traits ao padrão de C++.

Lembretes

- As classes de traits disponibilizam informações acerca de tipos durante a compilação. Elas são implementadas usando templates e especializações de templates.
- Juntamente com a sobrecarga, as classes de traits possibilitam realizar testes `if...else` em tipos em tempo de compilação.

Item 48: Fique atento à metaprogramação por templates

A metaprogramação por templates (TMP – *Template Metaprogramming*) é o processo de escrever programas C++ baseados em templates que são executados durante a compilação. Pense sobre isso por um minuto: um metaprograma por template é um programa escrito em C++ que executa *dentro do compilador C++*. Quando um programa TMP termina sua execução, sua saída – partes de código C++ instanciadas a partir de templates – são então compiladas como de costume.

Se isso não soa completamente bizarro para você é porque você não está pensando o suficiente a respeito.

C++ não foi projetada para metaprogramação por templates, mas, desde que a TMP foi descoberta no início dos anos 90, ela provou ser tão útil que, provavelmente, extensões serão adicionadas tanto à linguagem quanto à sua biblioteca padrão para tornar a TMP mais fácil. Sim, TMP foi descoberta – e não inventada. Os recursos subjacentes à TMP foram introduzidos quando os templates foram adicionados a C++. Bastava que alguém notasse como eles poderiam ser usados de maneiras inteligentes e inesperadas.

A TMP possui duas grandes forças. Primeiro, ela facilita coisas que, de outra forma, seriam difíceis ou impossíveis. Segundo, como ela é executada durante a compilação de C++, os templates podem deslocar o trabalho do tempo de execução para o tempo de compilação. Uma consequência é que alguns tipos de erros que são normalmente detectados em tempo de execução podem ser encontrados em tempo de compilação. Outra é que os programas em C++ que usam TMP podem ser mais eficientes de várias maneiras: executáveis menores, tempos de execução menores, menos requisitos de memória. (Entretanto, uma consequência do deslocamento do trabalho do tempo de execução para o de compilação é que a compilação demora mais. Os programas que usam TMP podem levar *muito mais* tempo para serem compilados que seus correspondentes sem TMP.)

Considere o pseudocódigo para `advance` da STL introduzido na página 228. (Isso está no Item 47. Você pode querer ler esse item agora, porque, aqui, considero que você conhece o conteúdo do Item 47.) Como na página 228, destaquei a parte em pseudocódigo do código:

```
template<typename IterT, typename DistT>
void advance(IterT& iter, DistT d)
{
   if (iter é um iterador de acesso aleatório) {
      iter += d;                                  // usa aritmética de operadores
   }                                              // para iteradores de acesso aleatório
   else {
      if (d >= 0) { while (d--) ++iter; }         // usa chamadas iterativas para
      else { while (d++) --iter; }                // ++ ou – para outras
   }                                              // categorias de iteradores
}
```

Podemos usar `typeid` para tornar o pseudocódigo real, o que leva a uma abordagem "normal" em C++ para esse problema – uma que faz todo o seu trabalho em tempo de execução:

```
template<typename IterT, typename DistT>
void advance(IterT& iter, DistT d)
{
   if ( typeid(typename std::iterator_traits<IterT>::iterator_category) ==
        typeid(std::random_access_iterator_tag)) {

      iter += d;                                  // usa aritmética de operadores
   }                                              // para iteradores de acesso aleatório

   else {
      if (d >= 0) { while (d--) ++iter; }         // usa chamadas iterativas para
      else { while (d++) --iter; }                // ++ ou -- para outras
   }                                              // categorias de iteradores
}
```

O Item 47 observa que essa abordagem baseada em `typeid` é menos eficiente do que a que usa traits, porque com ela, (1) o teste de tipo ocorre em tempo de execução, em vez de ocorrer em tempo de compilação e (2) o código para fazer o teste de tipo em tempo de execução deve estar presente

no executável. Na verdade, esse exemplo mostra como a TMP pode ser mais eficiente do que um programa C++ "normal", porque a abordagem de traits *é* TMP. Lembre-se, os traits permitem computações `if...else` sobre tipos em tempo de compilação.

Destaquei anteriormente que algumas coisas são mais fáceis em TMP do que em C++ "normal", e que `advance` oferece um exemplo disso também. O Item 47 menciona que a implementação baseada em `typeid` de `advance` pode levar a problemas de compilação, e veja um exemplo em que isso ocorre:

```
std::list<int>::iterator iter;
...
advance(iter, 10);                    // move iter 10 elementos para frente;
                                      // não será compilado com a implementação acima
```

Pense na versão de `advance` que será gerada para a chamada acima. Após substituir os tipos de `iter` e de 10 para os parâmetros de template `IterT` e `DistT`, obtemos o seguinte:

```
void advance(std::list<int>::iterator& iter, int d)
{
  if (typeid(std::iterator_traits<std::list<int>::iterator>::iterator_category) ==
      typeid(std::random_access_iterator_tag)) {

    iter += d;                                    // erro! não será compilado.

  }
  else {
    if (d >= 0) { while (d--) ++iter; }
    else { while (d++) --iter; }
  }
}
```

O problema é a linha destacada, aquela que usa `+=`. Nesse caso, estamos tentando usar `+=` em um `list<int>::iterator`, mas `list<int>::iterator` é um iterador bidirecional (veja o Item 47), então não suporta `+=`. Apenas os iteradores de acesso aleatório oferecem suporte a `+=`. Agora, sabemos que nunca tentaremos executar a linha `+=`, porque o teste com `typeid` sempre falharia para `list<int>::iterators`; no entanto, os compiladores são compelidos a garantir que todo o código seja válido, mesmo que não seja executado, e "`iter+=d`" não é válido quando `iter` não é um iterador de acesso aleatório. Compare isso com a solução usando TMP baseada em traits, em que cada uma delas usa apenas operações aplicáveis aos tipos para o qual ela é escrita.

Mostrou-se que a TMP é completa para Turing, ou seja, ela é poderosa o suficiente para computar qualquer coisa computável. Usando TMP, você pode declarar variáveis, realizar laços, escrever e chamar funções, etc. Mas essas construções se parecem bastante diferentes em relação às suas correspondentes em C++ "normal". Por exemplo, o Item 47 mostra como os condicionais `if...else` em TMP são expressos com templates e especializações de

templates. Mas isso é TMP no nível de linguagem de montagem. As bibliotecas para TMP (por exemplo, a MPL de Boost – veja o Item 55) oferecem uma sintaxe de mais alto nível, apesar de não ser algo que você confundiria com C++ "normal".

Para ver outro relance de como as coisas funcionam em TMP, vamos dar uma olhada nos laços. A TMP não possui uma construção de laço real, então o efeito dos laços é realizado por meio de recursão. (Se você não está confortável com recursão, precisará aprender isso antes de se aventurar com a TMP. De um modo geral ela é uma linguagem funcional, e a recursão está para as linguagens funcionais assim como a TV está para a cultura pop americana: são inseparáveis.) Mesmo a recursão não é do tipo normal; entretanto, como os laços em TMP não envolvem chamadas recursivas a funções, eles envolvem *instanciações recursivas de templates*.

O programa "*hello world*" de TMP consiste em computar um fatorial durante a compilação. Não é um programa que empolga muito, mas, mais uma vez, nem o "*hello world*" o é em outras introduções de linguagens. A computação do fatorial usando TMP demonstra o uso de laços por meio de instanciações recursivas de templates. Ela também demonstra uma maneira pela qual as variáveis são criadas e usadas em TMP. Veja:

```
template<unsigned n>                          // caso geral: o valor de
struct Factorial {                            // Factorial<n> é n vezes o valor
                                              // de Factorial<n-1>
   enum { value = n * Factorial<n-1>::value };

};

template<>                                    // caso especial: o
struct Factorial<0> {                         // Factorial<0> é 1
   enum { value = 1 };
};
```

Como esse metaprograma por template (na verdade, somente a metafunção de template `Factorial` – fatorial), você obtém o valor de fatorial de `n` referindo-se a `Factorial<n>::value`.

A parte de laço no código ocorre no ponto em que a instanciação de template `Factorial<n>` referencia a instanciação de template `Factorial<n-1>`. Como toda boa recursão, existe um caso especial que faz a recursão terminar. Aqui é a especialização de template `Factorial<0>`.

Cada instanciação do template `Factorial` é uma estrutura, e cada estrutura usa o hack de enumeração (veja o Item 2) para declarar uma variável de TMP chamada `value`. A variável `value` é o que mantém o valor atual da computação do fatorial. Se TMP tivesse uma construção de laço real, `value` seria atualizada toda vez no laço. Como a TMP usa a instanciação recursiva de templates no lugar de laços, cada instanciação obtém sua própria cópia de `value`, e cada cópia possui o valor apropriado para seu lugar no "laço".

Você poderia usar `Factorial` como segue:

```
int main( )
{
   std::cout << Factorial<5>::value;                    // imprime 120
   std::cout << Factorial<10>::value;                   // imprime 3628800
}
```

Se você acha que isso é a coisa mais legal do mundo, tem as habilidades de um metaprogramador de templates. Se os templates, as especializações, as instanciações recursivas, os hacks de enumeração e a necessidade de digitar coisas como `Fatorial<n-1>::value` fazem com que você se arrepie, você é um programador C++ bastante normal.

É claro, `Factorial` demonstra a utilidade da TMP tão bem quanto o "*hello world*" demonstra a utilidade de qualquer linguagem de programação convencional. Para entender por que vale a pena conhecê-la, é importante ter um melhor entendimento do que ela pode realizar. Veja três exemplos.

- **Garantir a correção de unidade de dimensão.** Em aplicações científicas e de engenharia, é essencial que unidades dimensionais (como massa, distância, tempo, etc) possam ser combinadas corretamente. Atribuir uma variável que represente massa com uma variável representando velocidade, por exemplo, é um erro, mas dividir uma variável de distância por uma variável de tempo e atribuir o resultado a uma variável de velocidade é aceitável. Usando TMP, é possível garantir (durante a compilação) que todas as combinações de unidades de dimensão em um programa sejam corretas, independentemente da complexidade dos cálculos. (Esse é um exemplo de como TMP pode ser usada para detecção de erros antecipadamente.) Um aspecto interessante desse uso de TMP é que os expoentes dimensionais fracionários podem ser suportados. Isso requer que essas frações sejam reduzidas *durante a compilação*, de forma que os compiladores possam confirmar, por exemplo, que a unidade $\texttt{time}^{1/2}$ é a mesma que $\texttt{time}^{4/8}$.

- **Otimizar operações de matrizes.** O Item 21 explica que algumas funções, incluindo `operator*`, devem retornar novos objetos, e o Item 44 introduz a classe `SquareMatrix`, então, considere o seguinte código:

  ```
  typedef SquareMatrix<double, 10000> BigMatrix;

  BigMatrix m1, m2, m3, m4, m5;                        // cria as matrizes e
  ...                                                  // dá valores a elas
  BigMatrix result = m1 * m2 * m3 * m4 * m5;           // computa seu produto
  ```

  Calcular `result` na maneira "normal" pede a criação de quatro matrizes temporárias, uma para o resultado de cada chamada a `operator*`. Além disso, as multiplicações independentes geram uma sequência de quatro laços sobre os elementos da matriz. Usando uma tecnologia avançada de templates relacionada à TMP chamada de *templates de expressões*, é possível eliminar as matrizes temporárias e mesclar os laços, tudo isso sem modificar a sintaxe do código cliente acima. O aplicativo de software resultante usa menos memória e roda incrivelmente mais rápido.

- **Gerar implementações personalizadas de padrões de projeto.** Padrões de projeto como o padrão Estratégia (Strategy – veja o Item 5), Observador (Observer), Visitante (Visitor), etc, podem ser implementados de diversas maneiras. Usando uma tecnologia baseada em TMP chamada de *projeto baseado em políticas*, é possível criar templates representando escolhas de projeto independentes ("políticas") que podem ser combinadas de maneiras arbitrárias para levar a implementações de padrões com comportamento personalizado. Por exemplo, essa técnica tem sido usada para permitir que alguns templates que implementam políticas comportamentais de ponteiros espertos gerem (durante a compilação) qualquer um dentre *centenas* de diferentes tipos de ponteiros espertos. Generalizada além do domínio de artefatos de programação como padrões de projeto e ponteiros espertos, essa tecnologia é a base para o que é conhecido como *programação gerativa (generative programming)*.

A TMP não é para todo mundo. A sintaxe não é intuitiva e o suporte ferramental é fraco. (Depuradores para metaprogramas por templates? Há!) Sendo uma linguagem "acidental" que foi descoberta de maneira relativamente recente, as convenções de programação em TMP são ainda, de certa forma, experimentais. Independentemente disso, as melhorias de eficiência que ocorrem ao deslocarmos o trabalho do tempo de execução para o tempo de compilação podem ser impressionantes, e a habilidade de expressar comportamento que é difícil ou impossível de ser implementado em tempo de execução também é atraente.

O suporte à TMP está aumentando. É provável que a próxima versão de C++ forneça suporte explícito para ela, e a TR1 já faz isso (veja o Item 54). Estão começando a publicação de livros sobre o assunto, e as informações sobre a TMP na Web estão cada vez mais ricas. A TMP provavelmente nunca será amplamente popular, mas, para alguns programadores – especialmente para os desenvolvedores de bibliotecas –, ela certamente será um item essencial.

Lembretes

» A metaprogramação por templates pode deslocar trabalho do tempo de execução para o tempo de compilação, habilitando uma detecção de erros antecipada e um melhor desempenho em tempo de execução.

» A TMP pode ser usada para gerar código personalizado baseada em combinações de escolhas de políticas, e ela também pode ser usada para evitar a geração de código inapropriado para certos tipos.

CAPÍTULO 8

PERSONALIZANDO NEW E DELETE

Em tempos em que os ambientes de computação estão ampliando o suporte predefinido para coleta de lixo (por exemplo, em Java e no .NET), a abordagem manual de C++ para o gerenciamento de memória pode parecer um tanto fora de moda. Mesmo assim, muitos desenvolvedores que estão trabalhando com aplicações de sistema exigentes escolhem C++ *porque* ele permite gerenciar a memória manualmente. Esses desenvolvedores estudam as características de uso de memória de seus aplicativos e então adaptam suas rotinas de alocação e de liberação para oferecer o melhor desempenho possível (tanto em termos de espaço quanto de tempo) para os sistemas que eles constroem.

Fazer isso requer um entendimento de como as rotinas de gerenciamento de memória em C++ se comportam, e este é o foco deste capítulo. Os dois principais jogadores nesse jogo são as rotinas de alocação e de liberação (`operator new` e `operator delete`), com um papel coadjuvante desempenhado pelo tratador de `new` – a função chamada quando `operator new` não puder satisfazer uma requisição de memória.

O gerenciamento de memória em um ambiente com múltiplas linhas de execução impõe desafios que não estão presentes em um sistema com uma só linha de execução, dado que o monte e o tratador de `new` são recursos globais modificados e, portanto, sujeitos a condições de corrida que podem acontecer no acesso a esses recursos em sistemas com múltiplas linhas de execução. Muitos itens neste capítulo mencionam o uso de dados estáticos modificáveis, algo que sempre coloca os programadores cientes dos problemas relacionados às linhas de execução em alerta. Sem uma sincronização apropriada, o uso de algoritmos de liberação de trancamento (*lock-free*) ou o projeto cuidadoso para impedir acesso concorrente, as chamadas às rotinas de memória podem facilmente levar a estruturas de dados gerenciadas corrompidas de monte. Em vez de lembrá-lo repetidamente desse perigo, o mencionarei aqui e considerarei que você manterá isso em mente no resto do capítulo.

Algo para você também ter em mente é que `operator new` e `operator delete` se aplicam apenas a alocações para objetos únicos. A memória para matrizes é alocada por `operator new[]` e liberada por `operator delete[]`. (Em ambos os casos, observe a parte "`[]`" dos no-

mes das funções. A menos que indicado de outra forma, tudo o que eu escrever sobre `operator new` e `operator delete` também se aplica a `operator new[]` e `operator delete[]`.

Por fim, observe que a memória do monte para contêineres da STL é gerenciada pelos objetos alocadores dos contêineres, não por `new` e `delete` diretamente. Sendo esse o caso, este capítulo não tem nada a dizer sobre alocadores da STL.

Item 49: Entenda o comportamento do tratador de `new`

Quando `operator new` não pode satisfazer uma requisição de alocação de memória, ele lança uma exceção. Há muito tempo, ele retornava um ponteiro nulo, e alguns compiladores mais antigos ainda fazem isso. Você ainda pode conseguir o comportamento antigo (ou quase), mas postergarei essa discussão até o final deste item.

Antes de `operator new` lançar uma exceção em resposta a uma requisição por memória que não pôde ser satisfeita, ele chama uma função de tratamento de erros especificável pelo cliente chamada de *tratador de new*. (Isso não é bem verdade. O que `operator new` realmente faz é um pouco mais complicado. No Item 51, são fornecidos detalhes.) Para especificar a função de tratamento de falta de memória, os clientes chamam `set_new_handler`, uma função da biblioteca padrão declarada em `<new>`:

```
namespace std {

  typedef void (*new_handler)( );
  new_handler set_new_handler(new_handler p) throw( );

}
```

Como você pode ver, `new_handler` é uma definição de tipo para um ponteiro que aponta para uma função que recebe e não retorna nada, e `set_new_handler` é uma função que recebe e retorna um `new_handler`. (O "`throw()`" no final da declaração de `set_ new_handler` é uma especificação de exceção. Ela, essencialmente, diz que essa função não lançará exceções, apesar de a verdade ser um pouco mais interessante. Para mais detalhes, consulte o Item 29.)

O parâmetro de `set_new_handler` é um ponteiro para a função que `operator new` deve chamar se ela não puder alocar a memória solicitada. O valor de retorno de `set_ new_handler` é um ponteiro para a função projetada para esse propósito antes de `set_new_handler` ser chamada.

Você deve usar `set_new_handler` como segue:

```
// função para chamar se operator new não puder alocar memória suficiente
void outOfMem( )
{
  std::cerr << "Unable to satisfy request for memory\n";
  std::abort( );
}
```

```
int main( )
{
   std::set_new_handler(outOfMem);
   int *pBigDataArray = new int[100000000L];
   ...
}
```

Se `operator new` não for capaz de alocar espaço para 100 milhões de inteiros, `outOfMem` será chamada e o programa abortará após mostrar uma mensagem de erro. (A propósito, considere o que acontece se a memória tiver de ser alocada dinamicamente durante o curso da escrita da mensagem de erro a `cerr`...).

Quando `operator new` não conseguir satisfazer uma requisição de memória, ele chama a função tratadora de `new` repetidamente até que *possa* encontrar memória suficiente. O código que dá vida a essas chamadas repetidas é mostrado no Item 51, mas essa descrição de alto nível é suficiente para se concluir que uma função tratadora de `new` bem projetada deve fazer uma das opções abaixo:

- **Disponibilizar mais memória.** Isso pode fazer com que a próxima tentativa de alocação de memória dentro de `operator new` seja bem-sucedida. Uma maneira de implementar essa estratégia é alocar um grande bloco de memória no início do programa e liberá-lo para o uso no programa na primeira vez em que o tratador de `new` for invocado.

- **Instalar um tratador de `new` diferente.** Se o tratador de `new` atual não puder disponibilizar mais nenhuma memória, talvez ele saiba de um tratador de `new` diferente que possa. Se esse for o caso, o tratador de `new` atual pode instalar o outro tratador de `new` em seu lugar (chamando `set_new_handler`). Na próxima vez em que `operator new` chamar a função tratadora de `new`, ele receberá o tratador mais recentemente instalado. (Uma variação desse esquema é um tratador de `new` modificar o seu *próprio* comportamento, então, na vez seguinte em que é invocado, ele faz algo diferente. Uma maneira de conseguir isso é fazer o tratador de `new` modificar dados estáticos, específicos do espaço de nomes, ou globais que afetem o comportamento do tratador de `new`.)

- **Desinstalar o tratador de `new`.** Ou seja, passar o ponteiro nulo para `set_new_handler`. Sem um tratador de `new` instalado, `operator new` lançará uma exceção quando a alocação de memória não for bem-sucedida.

- **Lançar uma exceção.** Essa opção lança uma exceção do tipo `bad_alloc` ou de algum tipo derivado de `bad_alloc`. Essas exceções não serão capturadas por `operator new`, então serão propagadas para o local que originou a requisição de memória.

- **Não retornar.** Nesse caso, em geral chama-se `abort` ou `exit`.

Essas escolhas dão uma flexibilidade considerável na implementação de funções tratadoras de `new`.

Às vezes, você quer tratar as falhas de alocação de memória de maneiras diferentes, dependendo da classe do objeto que está sendo alocado:

```
class X {
public:
   static void outOfMemory( );
   ...
};

class Y {
public:
   static void outOfMemory( );
   ...
};

X* p1 = new X;                    // se a alocação não for bem-sucedida
                                  // chama X::outOfMemory

Y* p2 = new Y;                    // se a alocação não for bem-sucedida
                                  // chama Y::outOfMemory
```

C++ não possui suporte para tratadores de `new` específicos por classe, mas ele não precisa de nenhum. Você mesmo pode implementar esse comportamento, basta que cada classe forneça suas próprias versões de `set_new_handler` e de `operator new`. O `set_new_handler` de uma classe permite que os clientes especifiquem o tratador de `new` para a classe (exatamente como o `set_new_handler` padrão permite que os clientes especifiquem o tratador de `new` global). O `operator new` da classe garante que o tratador de `new` específico da classe seja usado no lugar do tratador de `new` global quando for alocada memória para objetos da classe.

Suponhamos que você queira tratar as falhas de alocação de memória para a classe `Widget`. Você terá que manter o acompanhamento da função a ser chamada quando `operator new` não puder alocar memória suficiente para um objeto `Widget`, então vai declarar um membro estático do tipo `new_handler` para apontar para a função tratadora de `new` para a classe. `Widget` se parecerá com isto:

```
class Widget {
public:
   static std::new_handler set_new_handler(std::new_handler p) throw( );
   static void * operator new(std::size_t size) throw(std::bad_alloc);

private:
   static std::new_handler currentHandler;
};
```

Os membros estáticos de classe devem ser definidos fora da definição de classe (a menos que sejam constantes e inteiros – veja o Item 2), então:

```
std::new_handler Widget::currentHandler = 0;      // inicializa para nulo no
                                                  // arquivo de implem. da classe
```

A função `set_new_handler` em `Widget` salvará qualquer ponteiro que seja passado a ela e retornará qualquer ponteiro que tenha sido salvo antes da chamada. Isto é o que a versão padrão de `set_new_handler` faz:

```
std::new_handler Widget::set_new_handler(std::new_handler p) throw()
{
  std::new_handler oldHandler = currentHandler;
  currentHandler = p;
  return oldHandler;
}
```

Por fim, o `operator new` de `Widget` fará o seguinte:

1. Chamará o `set_new_handler` padrão com a função de tratamento de erros de `Widget`. Isso instala o tratador de `new` do `Widget` como o tratador de `new` global.

2. Chamará o `operator new` global para realizar a alocação de memória propriamente dita. Se a alocação falhar, o `operator new` global invocará o tratador do `new` de `Widget`, porque essa função foi recém-instalada como o tratador de `new` padrão. Se, por fim, o `operator new` global não conseguir alocar a memória, ele lançará uma exceção `bad_alloc`. Nesse caso, o `operator new` de `Widget` deve restaurar o tratador de `new` original, e então propagar a exceção. Para garantir que o tratador de `new` original seja sempre reconfigurado, `Widget` tratará o tratador de `new` global como um recurso e seguirá a recomendação do Item 13 de usar objetos de gerenciamento de recursos para impedir o vazamento de recursos.

3. Se o `operator new` conseguir alocar memória suficiente para um objeto `Widget`, o `operator new` do `Widget` retornará um ponteiro para a memória alocada. O destrutor para o objeto que gerencia o tratador de `new` padrão restaurará automaticamente o tratador de `new` padrão para o que ele era antes da chamada ao `operator new` de `Widget`.

Aqui temos como você faria tudo isso em C++. Começaremos com a classe de tratamento de recursos, que consiste em nada mais do que as operações RAII fundamentais de adquisição de um recurso durante a construção e liberação dele durante a destruição (veja o Item 13):

```
class NewHandlerHolder {
public:
  explicit NewHandlerHolder(std::new_handler nh)          // adquire o tratador de
  : handler(nh) { }                                       // new atual

  ~NewHandlerHolder( )                                    // libera-o
  { std::set_new_handler(handler); }

private:
  std::new_handler handler;                               // lembra dele

  NewHandlerHolder(const NewHandlerHolder&);              // impede cópias
  NewHandlerHolder&                                       // (veja o Item 14)
    operator=(const NewHandlerHolder&);
};
```

Isso simplifica bastante a implementação do `operator new` do `Widget`:

```
void * Widget::operator new(std::size_t size) throw(std::bad_alloc)
{
   NewHandlerHolder                                  // instala o tratador de new
      h(std::set_new_handler(currentHandler));       // do Widget

   return ::operator new(size);                      // aloca memória
                                                     // ou lança

}                                                    // restaura do tratador de new
                                                     // global
```

Os clientes de `Widget` usam suas novas capacidades de tratamento de `new` como segue:

```
void outOfMem( );                                    // declaração de função a ser chamada
                                                     // se a alocação de memória
                                                     // para objetos Widget falhar

Widget::set_new_handler(outOfMem);                   // define outOfMem como a
                                                     // função de trat. de new para Widget

Widget *pw1 = new Widget;                            // se a alocação de memória
                                                     // falhar, chama outOfMem

std::string *ps = new std::string;                   // se a alocação de memória falhar,
                                                     // chama a função tratadora de new
                                                     // global (se existir uma)

Widget::set_new_handler(0);                          // configura a função tratadora de new
                                                     // específica de Widget para
                                                     // nada (ou seja, null)

Widget *pw2 = new Widget;                            // se a alocação de memória falhar, lança uma
                                                     // exceção imediatamente. (Não existe
                                                     // uma função de trat. de new para
                                                     // a classe Widget).
```

O código para implementar esse esquema é o mesmo, independentemente da classe; então, um objetivo racional seria reusá-lo em outros locais. Uma maneira fácil de possibilitar isso é criar uma classe-base no "estilo mixin", ou seja, uma classe-base projetada para permitir que as classes derivadas herdem apenas uma capacidade específica – nesse caso, a habilidade de configurar um tratador de `new` específico de classe. Então, transforme a classe-base em um template, assim você pode obter uma cópia diferente dos dados da classe para cada classe derivada.

A parte da classe-base desse projeto deixa que as classes derivadas herdem as funções `set_new_handler` e `operator new` de que todas precisam, enquanto a parte template do projeto garante que cada classe derivada obtenha um membro de dados `currentHandler` (tratador atual) diferente. Isso pode parecer um pouco complicado, mas o código parece, de fato, bastante parecido. Na verdade, a única diferença real é que ele está agora disponível para qualquer classe que precisar dele:

```
template<typename T>                 // classe-base no "estilo mixin" para
class NewHandlerSupport {            // suporte
public:                              // a set_new_handler específico de classe
```

```
    static std::new_handler set_new_handler(std::new_handler p) throw( );
    static void * operator new(std::size_t size) throw(std::bad_alloc);

    ...                                          // para outras versões de operator new
                                                 // veja o Item 52
private:
    static std::new_handler currentHandler;
};

template<typename T>
std::new_handler
NewHandlerSupport<T>::set_new_handler(std::new_handler p) throw( )
{
    std::new_handler oldHandler = currentHandler;
    currentHandler = p;
    return oldHandler;
}

template<typename T>
void* NewHandlerSupport<T>::operator new(std::size_t size)
    throw(std::bad_alloc)
{
    NewHandlerHolder h(std::set_new_handler(currentHandler));
    return ::operator new(size);
}

// isso inicializa cada currentHandler para null
template<typename T>
std::new_handler NewHandlerSupport<T>::currentHandler = 0;
```

Com esse template de classe, fica fácil adicionar suporte de `set_new_handler` para `Widget`. Nesse caso, `Widget` apenas herda de `NewHandlerSupport<Widget>` (suporte para o tratador de `new`). (Pode parecer estranho, mas darei mais detalhes abaixo sobre o que está acontecendo exatamente.)

```
class Widget: public NewHandlerSupport<Widget> {
...                                              // como antes, mas sem declarações para
};                                               // set_new_handler ou operator new
```

Isso é tudo o que `Widget` precisa fazer para oferecer um `set_new_handler` específico de uma classe. Mas talvez você ainda esteja irritado por `Widget` herdar de `NewHandlerSupport<Widget>`. Se for esse o caso, sua irritação pode se intensificar quando você perceber que o template `NewHandlerSupport` nunca usa seu parâmetro de tipo `T`. Ele não precisa usá-lo. Tudo o que é necessário é uma cópia diferente de `NewHandlerSupport` – em particular, seu membro de dados estático `currentHandler` – para cada classe que herdar de `NewHandlerSupport`. O parâmetro de template `T` só distingue uma classe derivada de outra. O mecanismo de template por si só gera automaticamente uma cópia de `currentHandler` para cada `T` com o qual `NewHandlerSupport` é instanciada.

Quando `Widget` herdar de uma classe-base com templates que recebe `Widget` como parâmetro de tipo, não se sinta mal se a noção faz você se sentir um pouco desorientado. Ele inicialmente tem esse efeito em todo

mundo. Entretanto, acaba sendo uma técnica útil, ele tem um nome, embora um nome que reflete o fato de não parecer natural para ninguém na primeira vez que é visto. Essa técnica é chamada de *padrão de template curiosamente recorrente* (CRTP – *curiously recurring template pattern*). Falo sério.

Em certo ponto, publiquei um artigo sugerindo que um nome melhor seria "Faça por mim" porque, quando `Widget` herda de `NewHandlerSupport<Widget>`, está, na verdade, dizendo "Eu sou `Widget` e quero herdar da classe `NewHandlerSupport` para `Widget`". Ninguém usa meu nome proposto (nem mesmo eu), mas pensar sobre o CRTP como uma maneira de dizer "faça por mim" pode ajudá-lo a entender o que a herança com template está fazendo.

Templates como `NewHandlerSupport` fazem com que fique fácil adicionar um tratador de `new` específico para qualquer classe que queira um. A herança no estilo de mixins, entretanto, invariavelmente leva ao tópico de herança múltipla, e, antes de seguir por esse caminho, você deve ler o Item 40.

Até 1993, C++ requeria que `operator new` retornasse nulo quando era incapaz de alocar a memória solicitada. A função `operator new` é agora especificada para que lance uma exceção `bad_alloc`, mas um monte de código C++ foi escrito antes de os compiladores começarem a oferecer suporte à especificação revisada. O comitê de padronização de C++ não queria abandonar a base de código de teste para nulo, então forneceu formas alternativas de `operator new` que davam o comportamento tradicional "falha leva a nulo". Essas formas são chamadas de "sem lançamento" (nothrow), em parte porque empregam objetos `nothrow` (definidos no cabeçalho `<new>`) no local em que `new` é usado:

```
class Widget { ... };

Widget *pw1 = new Widget;                       // lança bad_alloc se
                                                // a alocação falhar

if (pw1 == 0) ...                               // este teste deve falhar


Widget *pw2 = new (std::nothrow) Widget;        // retorna 0 se a alocação
                                                // do Widget falhar

if (pw2 == 0) ...                               // este teste pode ser bem-sucedido
```

O `new` sem lançamento oferece uma garantia menos convincente sobre exceções do que inicialmente aparenta. Na expressão "`new (std::nothrow) Widget`", acontecem duas coisas. Primeiro, a versão sem lançamento de `operator new` é chamada para alocar memória suficiente para um objeto `Widget`. Se essa alocação falhar, `operator new` retorna o ponteiro nulo, como anunciado. Se for bem-sucedida, entretanto, o construtor de `Widget` é chamado e, nesse ponto, todas as apostas estão encerradas. O construtor de `Widget` pode fazer o que quiser. Ele mesmo pode alocar mais memória com `new` e, se o fizer, não fica restrito ao uso do `new` sem lançamento. Embora a chamada a `operator new` em

"`new (std::nothrow) Widget`" não lance, o construtor de `Widget` pode lançar. Se fizer isso, a exceção será propagada normalmente. Conclusão? Usar o `new` sem lançamento garante apenas que `operator new` não lance uma exceção, e não que uma expressão como "`new (std::nothrow) Widget`" nunca levará a uma exceção. Em todo o caso, você nunca precisará de um `new` sem lançamento.

Independentemente de você usar o `new` "normal" (ou seja, que lança exceções) ou seu primo sem lançamento um tanto inferior, é importante que entenda o comportamento do tratador de `new`, porque ele é usado em ambas as formas.

Lembretes

» `set_new_handler` permite especificar uma função a ser chamada quando as requisições de alocação de memória não podem ser satisfeitas.

» O `new` sem lançamento é de utilidade limitada, porque se aplica apenas à alocação de memória; as chamadas subsequentes a construtores podem ainda assim lançar exceções.

Item 50: Entenda quando faz sentido substituir `new` e `delete`

Vamos voltar aos fundamentos por um momento. Por que alguém iria querer substituir as versões de `operator new` ou `operator delete` fornecidas pelo compilador em primeiro lugar? Essas são as três razões mais comuns:

- **Detecção de erros de uso.** Não conseguir liberar memória por meio de `delete` alocada por `new` leva a vazamentos de memória. Usar mais de um `delete` em memória criada por `new` leva a comportamento indefinido. Se `operator new` mantém uma lista de endereços alocados e `operator delete` remove endereços da lista, é fácil detectar esses erros de uso. De maneira similar, uma variedade de erros de programação pode levar à escrita de dados em espaços de memória além de um bloco alocado (overrun) ou antes de um bloco alocado (underrun). Os operadores `new` personalizados podem alocar blocos a mais, de forma que exista espaço para colocar padrões de bytes conhecidos ("assinaturas") antes e depois da memória disponibilizada para os clientes. As funções `operator delete` podem verificar para ver se as assinaturas ainda estão intactas. Se elas não estiverem, é porque ocorreu algum acesso antes ou depois do bloco em algum momento da vida do bloco alocado. A função `operator delete` pode criar logs desse fato, juntamente com o valor do ponteiro com problemas.

- **Melhoria da eficiência.** As versões de `operator new` e `operator delete` que vêm junto com os compiladores são projetadas para uso de propósito geral. Elas precisam ser aceitáveis para programas de longa execução (como servidores da Web), mas também

precisam ser aceitáveis para os programas que executam por menos de um segundo. Precisam manipular séries de requisições para grandes blocos de memória, para pequenos blocos e para mistos dos dois. Também precisam acomodar padrões de alocação que variam da alocação dinâmica de alguns blocos que existem por toda a duração do programa à alocação e à liberação constante de um grande número de objetos de vida curta. Precisam ainda se preocupar com fragmentação do monte, um processo que, se não verificado, no fim das contas leva à incapacidade de satisfazer requisições para grandes blocos de memória, mesmo quando uma ampla quantidade de memória livre está distribuída entre muitos blocos pequenos.

Dadas as demandas feitas para os gerentes de memória, não é surpresa que os operadores `new` e `delete` que vêm com os compiladores adotem uma estratégia neutra. Elas funcionam razoavelmente bem para todo mundo e otimamente para ninguém. Se você tem um bom entendimento dos padrões de uso de memória dinâmica de seu programa, pode descobrir que versões personalizadas de `operator new` e `operator delete` são melhores em termos de desempenho do que as versões padrão. Por "melhores em termos de desempenho" quero dizer que elas são executadas mais rapidamente – algumas vezes, várias ordens de grandeza mais rapidamente – e que requerem menos memória – até 50% menos. Para algumas aplicações (não todas, obviamente), substituir os operadores `new` e `delete` padrão por versões personalizadas é uma maneira muito fácil de melhorar significativamente o desempenho.

- **Coleta de estatísticas de uso.** Antes de escolher o caminho da escrita de `new`s e `delete`s personalizados, é prudente obter informações sobre como seu aplicativo usa sua memória dinâmica. Qual é a distribuição dos tamanhos de blocos alocados? Qual é a distribuição de seus tempos de vida? Tendem a ser alocados e liberados em ordem FIFO ("first in, first out" – primeiro a entrar, primeiro a sair), em ordem LIFO ("last in, first out" – último a entrar, primeiro a sair) ou algo mais próximo a uma ordem aleatória? Os padrões de uso mudam com o tempo? Ou seja, seu aplicativo possui diferentes padrões de alocação/liberação em diferentes estágios da execução? Qual é a quantidade máxima de memória alocada dinamicamente em uso ao longo do tempo (ou seja, o pico máximo)? As versões personalizadas de `operator new` e `operator delete` facilitam a coleta desse tipo de informação.

Na teoria, escrever um `operator new` personalizado é bastante fácil. Por exemplo, veja uma rápida primeira passada em um `operator new` global que facilita a detecção de acessos fora dos blocos de memória disponíveis. Existe um monte de coisinhas erradas com ele, mas nos preocuparemos com isso mais tarde.

```
static const int signature = 0xDEADBEEF;
```

```
typedef unsigned char Byte;
                                        // este código possui diversas falhas – veja abaixo
void* operator new(std::size_t size) throw(std::bad_alloc)
{
  using namespace std;

  size_t realSize = size + 2 * sizeof(int); // aumenta o tamanho da requisição de forma que duas
                                        // assinaturas também caibam dentro
  void *pMem = malloc(realSize);        // chama malloc para obter a memória
  if (!pMem) throw bad_alloc( );        // real
                                        // escreve a assinatura na primeira e na
                                        // última parte da memória
  *(static_cast<int*>(pMem)) = signature;
  *(reinterpret_cast<int*>(static_cast<Byte*>(pMem)+realSize-sizeof(int))) =
    signature;
                                        // retorna um ponteiro para a memória
                                        // logo após a primeira assinatura
  return static_cast<Byte*>(pMem) + sizeof(int);
}
```

A maioria das limitações desse `operator new` tem a ver com a não adesão às convenções de C++ para funções desse nome. Por exemplo, o Item 51 explica que todos os `operator new`s devem conter um laço chamando uma função de tratamento de `new`, mas esse não faz isso. Mas como o Item 51 dedica-se a essas convenções, eu as ignorarei aqui. Quero me focar em uma questão mais sutil agora: *alinhamento*.

Muitas arquiteturas de computador requerem que os dados de determinados tipos sejam colocados em memória determinados em tipos de endereços. Por exemplo, uma arquitetura pode requerer que ocorram ponteiros em endereços que são múltiplos de quatro (ou seja, que sejam *alinhados em quatro bytes*) ou que `double`s devam ocorrer em endereços que são múltiplos de oito (ou seja, que sejam *alinhados em oito bytes*). Não seguir essas restrições poderia levar a exceções de hardware em tempo de execução. Outras arquiteturas permitem uma maior liberdade, apesar de oferecerem melhor desempenho se as preferências de alinhamento forem satisfeitas. Por exemplo, `double`s podem ser alinhados em qualquer limite de byte na arquitetura Intel x86, mas o acesso a eles é muito mais rápido se forem alinhados em oito bytes.

O alinhamento é relevante aqui, porque C++ requer que todos os `operator new`s retornem ponteiros que sejam alinhados adequadamente com qualquer tipo de dados. A função `malloc` trabalha com os mesmos requisitos; assim, fazer `operator new` retornar um ponteiro que ele obtém de `malloc` é seguro. Entretanto, no `operator new` acima, não estamos retornando um ponteiro que recebemos de `malloc`, estamos retornando um ponteiro que obtemos de `malloc` *deslocado pelo tamanho de um inteiro*. Não existe garantia de que isso seja seguro! Se o cliente chamou `operator new` para ter memória suficiente para um `double` (ou se estivesse escrevendo `operator new[]`, um vetor de `double`s) e estivéssemos executando em uma máquina em que os inteiros tivessem tamanhos de quatro bytes, mas os `double`s exigissem o alinhamento em oito bytes, provavel-

mente retornaríamos um ponteiro com um alinhamento inadequado. Isso pode fazer o programa quebrar. Ou pode apenas fazê-lo ser executado mais lentamente. De qualquer forma, provavelmente não é o que tínhamos em mente.

Detalhes como o alinhamento são o tipo de coisa que distinguem os gerenciadores de memória de qualidade profissional daqueles feitos de qualquer jeito por programadores distraídos pela necessidade de realizar outras tarefas. Escrever um gerenciador de memória personalizado que quase funciona é bastante fácil. Escrever um que funcione *bem* é bem mais difícil. Como regra geral, sugiro que você não tente fazê-lo a menos que seja obrigado a isso.

Em muitos casos, você não é. Alguns compiladores possuem opções que habilitam funcionalidades de depuração e de logging em suas funções de gerenciamento de memória. Uma rápida olhada na documentação de seu compilador pode eliminar sua necessidade de considerar a escrita de `new` e de `delete`. Em muitas plataformas, produtos comerciais podem substituir as funções de gerenciamento de memória que vêm com os compiladores. Para avaliar você mesmo sua funcionalidade melhorada e (possivelmente) um desempenho melhorado, tudo o que precisa fazer é religá-los (bem, e também comprá-los).

Outra opção são os gerenciadores de memória de código aberto. Eles estão disponíveis para muitas plataformas, então você pode baixá-los e tentar usar algum deles. Um desses alocadores de código aberto é a biblioteca Pool de Boost (veja o Item 55). A biblioteca Pool oferece alocadores sintonizados para uma das situações mais comuns nas quais o gerenciamento de memória personalizado é útil: a alocação de um grande número de pequenos objetos. Muitos livros de C++, incluindo edições anteriores deste aqui, mostram o código para um alocador de objetos pequeno de alto desempenho, mas, normalmente, omitem esses detalhes irritantes em relação à portabilidade e às considerações de alinhamento, segurança de linhas de execução, etc. Bibliotecas reais tendem a ter código muito mais robusto. Mesmo que você decida escrever seus próprios `new`s e `delete`s, ver as versões de código aberto provavelmente lhe trará ideias sobre detalhes que normalmente são ignorados e que separam o "quase funcionando" do "realmente funcionando". (Como o alinhamento é um desses detalhes, vale observar que a TR1 [veja o Item 54] inclui suporte para descobrir requisitos de alinhamento específicos de tipo.)

O tópico deste Item é saber quando faz sentido substituir as versões padrão de `new` e `delete`, seja globalmente, seja de classe em classe. Agora chegamos ao ponto de resumir isso em mais detalhes do que tínhamos feito anteriormente.

- **Detecção de erros de uso** (como acima).
- **Coleta de estatísticas sobre o uso de memória alocada dinamicamente** (também como acima).

- **Aumento da velocidade da alocação e da liberação.** Os alocadores de propósito geral são frequentemente (embora nem sempre) bem mais lentos do que as versões personalizadas, especialmente se as versões personalizadas forem projetadas para objetos de um tipo em particular. Os alocadores específicos de classe são um exemplo de aplicação de alocadores de tamanho fixo como aqueles oferecidos pela biblioteca Pool de Boost. Se sua aplicação possui uma só linha de execução, mas as rotinas padrão de gerenciamento de memória de seu compilador são seguras em relação a linhas de execução, você pode ter melhorias de velocidade consideráveis ao escrever alocadores inseguros em relação a linhas de execução. É claro, antes de chegar rapidamente à conclusão de que vale a pena tornar as funções `operator new` e `operator delete` mais rápidas, certifique-se de medir detalhadamente o desempenho de seu programa para confirmar que essas funções são realmente um gargalo.

- **Redução da sobrecarga de espaço do gerenciamento padrão de memória.** Os gerenciadores de memória de propósito geral são frequentemente (embora nem sempre) mais lentos do que as versões personalizadas, além de usarem mais memória. Isso porque é comum incorrem em sobrecarga para cada bloco alocado. Os alocadores sintonizados para objetos pequenos (como aqueles na biblioteca Pool de Boost) essencialmente eliminam tal sobrecarga.

- **Compensação de um alinhamento subótimo no alocador padrão.** Como mencionei anteriormente, é mais rápido acessar `double`s na arquitetura x86 quando eles são alinhados em oito bytes. Infelizmente, as funções `operator new` disponibilizadas por alguns compiladores não garantem o alinhamento em oito bytes para alocações dinâmicas de `double`s. Nesses casos, substituir o `operator new` padrão por um que garanta o alinhamento em oito bytes pode levar a grandes melhorias no desempenho de programas.

- **Agrupamento de objetos relacionados.** Se você sabe quais são as estruturas de dados em particular geralmente usadas em conjunto e quer minimizar a frequência de faltas de páginas quando estiver trabalhando com esses dados, pode fazer sentido criar um monte separado para as estruturas de dados de forma que fiquem juntas e agrupadas no mínimo possível de páginas. As versões de posicionamento de `new` e `delete` (veja o Item 52) podem possibilitar que se atinja tal agrupamento.

- **Obtenção de comportamento não convencional.** Às vezes, você quer que os operadores `new` e `delete` façam algo que as versões fornecidas pelo compilador não oferecem. Por exemplo, você pode querer alocar e liberar blocos em memória compartilhada, mas tem apenas uma API em C por meio da qual essa memória é gerenciada. Escrever versões personalizadas de `new` e de `delete` (provavelmente versões de posicionamento – mais uma vez, veja o Item 52) permitiria que você vestisse a API C em uma roupagem C++. Como outro exemplo, você pode querer escrever um operador `delete` personalizado que sobrescreva a memória liberada com zeros para aumentar a segurança dos dados de um aplicativo.

Lembrete

» Existem muitas razões válidas para escrever versões personalizadas de `new` e de `delete`, incluindo melhorias de desempenho, erros de uso de depuração do monte e coleta de informações de uso do monte.

Item 51: Adote a convenção quando estiver escrevendo `new` e `delete`

O Item 50 explica quando você pode escrever suas próprias versões de `operator new` e de `operator delete`, mas não explica as convenções que deve seguir quando fizer isso. As regras não são difíceis de seguir, mas algumas não são intuitivas, então é importante saber o que elas são e do que tratam.

Iniciaremos com `operator new`. Implementar um `operator new` que esteja em conformidade com as regras exige que se tenha o valor de retorno correto, chamando a função de tratamento de `new` quando não houver memória suficiente (veja o Item 49) e que se esteja preparado para lidar com requisições para nenhuma memória. Você também vai querer evitar a ocultação inadvertida da forma "normal" de `new`, embora essa seja mais uma questão de interface de classe do que um requisito de implementação; isso é tratado no Item 52.

A parte do valor de retorno de `operator new` é fácil. Se você puder fornecer a memória requisitada, deverá retornar um ponteiro para ela. Se não puder, siga a regra descrita no Item 49 e lance uma exceção do tipo `bad_alloc`.

Não é assim tão simples, entretanto, porque `operator new`, na verdade, tenta alocar memória mais de uma vez, chamando a função de tratamento de `new` após cada falha. A premissa aqui é que a função de tratamento de `new` pode fazer algo para liberar alguma memória. Apenas quando o ponteiro para a função de tratamento de `new` for nulo é que `operator new` lança uma exceção.

Curiosamente, C++ requer que `operator new` retorne um ponteiro legítimo, mesmo quando for necessário zero byte. (Requerer esse comportamento que soa estranho simplifica as coisas em todos os outros lugares da linguagem.) Sendo esse o caso, o pseudocódigo para um `operator new` que não seja uma função membro se pareceria com o seguinte:

```
void * operator new(std::size_t size) throw(std::bad_alloc)
{                                                  // seu operator new pode
   using namespace std;                            // receber parâmetros adicionais

   if (size == 0) {                                // trata requisições de 0 byte
      size = 1;                                    // tratando-as como
   }                                               // requisições de 1 byte
```

```
    while (true) {
      tenta alocar um número de bytes igual ao valor do parâmetro size
      if (a alocação foi bem-sucedida)
        return (um ponteiro para a memória);

      // a alocação não foi bem-sucedida; descubra
      //qual é a função de tratamento de new atual (veja abaixo)
      new_handler globalHandler = set_new_handler(0);
      set_new_handler(globalHandler);

      if (globalHandler) (*globalHandler)( );
      else throw std::bad_alloc( );
    }
}
```

O truque de tratar requisições para zero byte como se fossem para um byte parece traiçoeiro, mas é simples, permitido e funciona. E, na verdade, com que frequência você esperaria receber uma requisição para zero byte?

Você também pode estar olhando atravessado para o local no pseudocódigo em que o ponteiro para a função de tratamento de `new` é configurado como nulo e imediatamente depois reconfigurado para o que era originalmente. Infelizmente, não é possível chegar ao ponteiro para a função de tratamento de `new` diretamente, então você precisa chamar `set_new_handler` para descobrir o que há nele. Bruto, é verdade, mas também eficaz, pelo menos para o código com uma linha única de execução. Em um ambiente com múltiplas linhas de execução, você provavelmente precisará de algum tipo de cadeado para manipular, de maneira segura, as estruturas de dados (globais) por trás da função de tratamento de `new`.

O Item 49 destaca que `operator new` contém um laço infinito, e o código acima mostra esse laço explicitamente; "`while(true)`" é tão infinito quanto consegue ser. A única maneira para sair do laço é a memória ser alocada com sucesso, ou a função de tratamento de `new` fazer uma das coisas no Item 49: disponibilizar mais memória, instalar um tratador de `new` diferente, desinstalar o tratador de `new`, lançar uma exceção `bad_alloc` ou uma exceção derivada dela, ou não retornar. Agora deve estar claro por que o tratador de `new` deve fazer uma dessas coisas. Se ele não fizer, o laço dentro de `operator new` nunca terminará.

Muitas pessoas não se dão conta de que as funções membro `operator new` são herdadas pelas classes derivadas. Isso pode levar a algumas complicações interessantes. No pseudocódigo para `operator new` acima, observe que a função tenta alocar `size` bytes (a menos que `size` seja zero). Isso faz sentido perfeitamente, porque esse é o argumento que foi passado para a função. Entretanto, como explica o Item 50, uma das razões mais comuns para escrever um gerenciador de memória personalizado é a otimização da alocação de objetos para uma classe *específica*, não para uma classe ou para qualquer de suas classes derivadas. Ou seja, dado o `operator new` para uma classe `X`, o comportamento dessa função em geral é otimizado para objetos de tamanho `sizeof(X)` – nada maior ou

menor. Devido à herança, entretanto, é possível que o `operator new` em uma classe-base seja chamado para alocar memória para um objeto de uma classe derivada:

```
class Base {
public:
   static void * operator new(std::size_t size) throw(std::bad_alloc);
   ...
};

class Derived: public Base                          // Derivada não declara
{ ... };                                            // operator new

Derived *p = new Derived;                           // chama Base::operator new!
```

Se o `operator new` específico da classe-`base` não for projetado para lidar com isso – e existe uma chance de ele não ser –, a melhor maneira de lidar com essa situação é redirecionar as chamadas que exigem a quantidade "errada" de memória para o `operator new` padrão, assim:

```
void * Base::operator new(std::size_t size) throw(std::bad_alloc)
{
   if (size != sizeof(Base))                        // se o tamanho estiver "errado"
      return ::operator new(size);                  // faça com que o operator new
                                                    // padrão trate a requisição

   ...                                              // caso contrário, trate
                                                    // a requisição aqui
}
```

"Espere!", escuto você gritar. "Você se esqueceu de verificar o *caso patológico, mas ainda assim possível*, de quando `size` é zero". Na verdade, eu não esqueci, e, por favor, pare de usar itálico quando estiver gritando. O teste ainda está lá; ele apenas foi incorporado no teste de `size` em relação a `sizeof(Base)`. C++ funciona de algumas formas misteriosas, e uma dessas maneiras é decretar que todos os objetos livres possuem tamanho diferente de zero (veja o Item 39). Por definição, `sizeof(Base)` nunca pode ser zero; então, se `size` for zero, a requisição será redirecionada para `::operator new` e será responsabilidade dessa função tratar a requisição de uma maneira razoável.

Se você quer controlar a alocação de memória para vetores em cada classe, precisa implementar o primo de `operator new` específico para vetores, `operator new[]`. (Essa função é normalmente chamada de "array new", porque é difícil descobrir como pronunciar "`operator new[]`".) Se decidir escrever `operator new[]`, lembre-se de que tudo o que está fazendo é alocar uma porção de memória bruta – você não pode fazer nada para os objetos ainda não existentes no vetor. Na verdade, não pode descobrir nem mesmo quantos objetos estarão no vetor. Primeiro, você não sabe o tamanho de cada objeto. Afinal, um `operator new[]` da classe-base pode, por meio de herança, ser chamado para alocar memória para uma matriz de objetos da classe derivada, e objetos da classe derivada são normalmente maiores do que os objetos da classe-base.

Assim, dentro de `Base::operator new[]`, você não pode considerar que o tamanho de cada objeto que vai dentro do vetor é `sizeof(Base)`, e isso significa que você não pode considerar que o número de objetos no vetor é (*bytes requeridos*)/`sizeof(Base)`. Segundo, o parâmetro `size_t` passado para `operator new[]` pode ser para mais memória que será preenchida com objetos, porque, como o Item 16 explica, os vetores alocados dinamicamente podem incluir espaço extra para armazenar os números dos elementos do vetor.

Essas são as várias convenções que você precisa seguir quando estiver escrevendo `operator new`. Para `operator delete`, as coisas são mais simples. Praticamente tudo o que você precisa se lembrar é de que C++ garante que seja sempre seguro apagar o ponteiro nulo, então você precisa honrar essa garantia. Veja um pseudocódigo para um `operator delete` não membro:

```
void operator delete(void *rawMemory) throw( )
{
   if (rawMemory == 0) return;                    // não faça nada se o ponteiro
                                                  // nulo estiver sendo apagado

   libera a memória apontada por rawMemory;
}
```

A versão membro dessa função também é simples, com a diferença de que você precisa lembrar de verificar o tamanho do que está sendo apagado. Considerando que seu `operator new` específico de classe encaminhe as requisições com o tamanho "errado" para `::operator new`, você precisa encaminhar as requisições de exclusão "com o tamanho errado" para `::operator delete`:

```
class Base {                                      // mesmo que antes, mas agora
public:                                           // operator delete é declarado

   static void * operator new(std::size_t size) throw(std::bad_alloc);
   static void operator delete(void *rawMemory, std::size_t size) throw( );
   ...
};

void Base::operator delete(void *rawMemory, std::size_t size) throw( )
{
   if (rawMemory == 0) return;                    // verifica o ponteiro nulo

   if (size != sizeof(Base)) {                    // se o tamanho é "errado" faça
      ::operator delete(rawMemory);               // com que operator delete padrão
      return;                                     // trate a requisição
   }

   libera a memória apontada por rawMemory;

   return;
}
```

É interessante observar que o valor de `size_t` que C++ passa para `operator delete` pode ser incorreto se o objeto que está sendo apagado for derivado de uma classe-base que não tem um destrutor virtual. Essa é uma razão suficiente para conferir se suas classes-base têm destrutores

virtuais, mas o Item 7 descreve uma segunda razão, bem melhor. Por enquanto, simplesmente observe que, se você omitir os destrutores virtuais nas classes-base, as funções `operator delete` podem não funcionar corretamente.

Lembretes

» `operator new` deve conter um laço infinito que tente alocar memória, chamar o tratador de `new` se não puder satisfazer uma requisição de memória e tratar requisições para zero byte. Versões específicas de classe devem tratar requisições para blocos maiores do que o esperado.

» `operator delete` não deve fazer nada se for passado a ele um ponteiro nulo. Versões específicas de classe devem tratar requisições para blocos maiores do que o esperado.

Item 52: Escreva `delete` de posicionamento se escrever `new` de posicionamento

Os operadores `new` e `delete` de posicionamento (placement `new` e `delete`) não são as feras mais encontradas na fauna de C++, então não se preocupe se não conhecê-los. Em vez disso, lembre-se dos Itens 16 e 17, que dizem que, quando você escreve uma expressão `new` como

```
Widget *pw = new Widget;
```

duas funções são chamadas: uma para `operator new`, de forma a alocar memória, e outra para o construtor padrão de `Widget`.

Suponhamos que a primeira chamada seja bem-sucedida, mas a segunda resulte no lançamento de uma exceção. Se for esse o caso, a alocação de memória realizada no passo 1 deve ser desfeita. Caso contrário, teremos um vazamento de memória. O código cliente não pode liberar a memória porque, se o construtor de `Widget` lançar uma exceção, `pw` nunca receberá uma atribuição. Os clientes não conseguirão obter um ponteiro para a memória que deve ser liberada. A responsabilidade de desfazer o passo 1 deve, então, ser do sistema de tempo de execução de C++.

O sistema de tempo de execução chama o `operator delete` que corresponde à versão de `operator new` que chamou no passo 1, mas ele só pode fazer isso se souber qual `operator delete` é o adequado para ser chamado – podem existir vários –. Isso não é problema se você estiver lidando com as versões de `new` e `delete` que possuem as assinaturas normais, porque o `operator new` normal

```
void* operator new(std::size_t) throw(std::bad_alloc);
```

corresponde ao `operator delete` normal:

```
void operator delete(void *rawMemory) throw( );          // assinatura normal
                                                         // em escopo global

void operator delete(void *rawMemory,                    // assinatura normal
                     std::size_t size) throw( );         // comum no escopo
                                                         // da classe
```

Quando você usa apenas as formas normais de `new` e de `delete`, o sistema de tempo de execução não tem dificuldade de encontrar o `delete` que sabe como desfazer o que `new` fez. A questão sobre qual `delete` vai com esse `new` surge quando você começa a declarar as formas não normais de `operator new` – formas que recebem parâmetros adicionais.

Por exemplo, suponhamos que você escreva um `operator new` específico de classe que exija a especificação de uma `ostream` para a qual as informações de alocação devam ser gravadas em log, e você também escreveu um `operator delete` específico de classe normal:

```
class Widget {
public:
  ...
  static void* operator new(std::size_t size,            // forma de new
                            std::ostream& logStream)     // não normal
    throw(std::bad_alloc);

  static void operator delete(void *pMemory              // forma normal
                              size_t size) throw( );     // de delete específico
                                                         // de classe
  ...
};
```

Esse projeto é problemático, mas, antes de vermos por que, precisamos fazer uma breve pausa para ver terminologia.

Quando uma função `operator new` recebe parâmetros extras (além do argumento obrigatório `size_t`), essa função é conhecida como a versão de *posicionamento* de `new`. O `operator new` acima é então uma versão de posicionamento. Um `new` de posicionamento particularmente útil é aquele que recebe um ponteiro especificando onde um objeto deve ser construído. Esse `operator new` se parece com o seguinte:

```
void* operator new(std::size_t, void *pMemory) throw( );   // "new de
                                                           // posicionamento"
```

Essa versão de `new` faz parte da biblioteca padrão de C++, e você tem de acessá-la sempre que faz `#include<new>`. Dentre outras coisas, esse `new` é usado dentro de vetores (`vectors`) para criar objetos na capacidade não utilizada do vetor. Ele é também o `new` de posicionamento *original*. Na verdade, é assim que essa função é conhecida: como `new` de *posicionamento*. Desse modo, o termo "`new` de posicionamento" é sobrecarregado. Na maioria das vezes, quando as pessoas falam sobre o `new` de posicionamento, eles estão falando sobre essa função específica, `operator new`, que recebe um argumento adicional do tipo `void*`. Menos comum é elas falarem sobre qualquer versão de `operator new`

que recebe argumentos extras. O contexto geralmente esclarece qualquer ambiguidade, mas é importante entender que o termo geral "`new` de posicionamento" significa qualquer versão de `new` que recebe argumentos extras, porque a frase "`delete` de posicionamento" (que encontraremos em instantes) é derivada desse termo geral.

Mas vamos voltar à declaração da classe `Widget`, aquela cujo projeto eu disse que era problemático. A dificuldade é que essa classe fará com que surjam vazamentos de memória sutis. Considere o seguinte código cliente, que cria logs de informações de alocação para `cerr` quando um `Widget` estiver sendo criado:

```
Widget *pw = new (std::cerr) Widget;          // chama operator new, passando cerr como
                                              // a ostream; isso vaza memória
                                              // se o construtor de Widget lançar uma exceção
```

Mais uma vez, se a alocação de memória for bem-sucedida e o construtor de `Widget` lançar uma exceção, o sistema de tempo de execução será responsável por desfazer a alocação que `operator new` realizou. Entretanto, o sistema de tempo de execução não consegue, na realidade, entender como a versão chamada de `operator new` funciona, então ele não consegue desfazer a alocação. Em vez disso, o sistema de tempo de execução procura uma versão de `operator delete` que receba *o mesmo número e os mesmos tipos de argumentos extras* de `operator new`, e, se encontrar, é ela que será chamada. Nesse caso, `operator new` recebe um argumento extra do tipo `ostream&`, então o `operator delete` correspondente teria a seguinte assinatura:

```
void operator delete(void *, std::ostream&) throw( );
```

Por analogia às versões de posicionamento de `new`, as versões de `operator delete` que recebem parâmetros extras são conhecidas como `delete`s de posicionamento. Nesse caso, `Widget` não declara nenhuma versão de posicionamento de `operator delete`, então o sistema de tempo de execução não sabe como desfazer o que a chamada ao `new` de posicionamento faz. Como resultado, ele não faz nada. Nesse exemplo, *nenhum* `operator delete` *será chamado* se o construtor de `Widget` lançar uma exceção!

A regra é simples: se um `operator new` com parâmetros extras não coincidir com um `operator delete` com os mesmos parâmetros extras, nenhum `operator delete` será chamado se uma alocação de memória por `new` precisar ser desfeita. Para eliminar o vazamento de memória no código acima, `Widget` precisa declarar um `delete` de posicionamento que corresponda ao `new` de posicionamento que faz logging:

```
class Widget {
public:
  ...
  static void* operator new(std::size_t size, std::ostream& logStream)
    throw(std::bad_alloc);
```

```
    static void operator delete(void *pMemory) throw( );

    static void operator delete(void *pMemory, std::ostream& logStream)
      throw( );

    ...
};
```

Com essa mudança, se for lançada uma exceção a partir do construtor de `Widget` na sentença

```
Widget *pw = new (std::cerr) Widget;          // como antes, mas desta vez sem vazamento
```

o `delete` de posicionamento correspondente é automaticamente invocado, e isso permite que `Widget` não deixe nenhuma memória vazar.

Entretanto, considere o que acontece se não for lançada nenhuma exceção (o que normalmente será o caso) e chegarmos a um `delete` no código cliente:

```
delete pw;                                    // invoca o operator delete
                                              // normal
```

Como o comentário indica, isso chama o `operator delete` normal, não a versão de posicionamento. O `delete` de posicionamento é chamado *apenas* se uma exceção for lançada de uma chamada a construtor que combina com uma chamada a um `new` de posicionamento. Aplicar `delete` a um ponteiro (como `pw` acima) nunca leva uma chamada a uma versão de posicionamento de `delete`. *Nunca*.

Isso significa que, para impedir todos os vazamentos de memória associados com as versões de posicionamento de `new`, você deve fornecer tanto o `operator delete` normal (por exemplo, quando não for lançada nenhuma exceção durante a construção) quanto uma versão de posicionamento que receba os mesmos argumentos extras que `operator new` (por exemplo, quando uma exceção ocorrer). Faça isso, e você nunca mais perderá o sono por causa de vazamentos de memória sutis. Bem, ao menos não *esses* vazamentos de memória sutis.

Casualmente, como os nomes de funções membro ocultam funções com o mesmo nome em escopos mais externos (veja o Item 33), você precisa ter cuidado para que `new`s específicos de classe não ocultem outros `new`s (incluindo as versões normais) que seus clientes esperam. Por exemplo, se você tem uma classe-base que declara apenas uma versão de posicionamento de `operator new`, os clientes descobrirão que a forma normal de `new` estará indisponível para eles:

```
class Base {
public:
  ...
  static void* operator new(std::size_t size,                    // esse new oculta
                            std::ostream& logStream)             // as formas globais
    throw(std::bad_alloc);                                       // normais
  ...
};
```

```
Base *pb = new Base;                          // erro! a forma normal de
                                              // operator new está oculta

Base *pb = new (std::cerr) Base;              // ok, chama o new de posicionamento
                                              // de Base
```

De maneira semelhante, `operator new`s, em classes derivadas, ocultam tanto as versões globais de `operator new` quanto as herdadas:

```
class Derived: public Base {                  // herda de Base acima
public:
  ...
  static void* operator new(std::size_t size) // redeclara a forma
    throw(std::bad_alloc);                    // normal de new
  ...
};

Derived *pd = new (std::clog) Derived;        // erro! o new de posicionamento
                                              // de Base está oculto

Derived *pd = new Derived;                    // ok, chama o operator new de
                                              // Derived
```

O Item 33 discute esse tipo de ocultamento de nomes em detalhes, mas, para escrever funções de alocação de memória, o que você precisa lembrar é que, por padrão, C++ oferece as seguintes formas de `operator new` em escopo global:

```
void* operator new(std::size_t) throw(std::bad_alloc);          // new normal

void* operator new(std::size_t, void*) throw( );                // new de posicionamento

void* operator new(std::size_t,                                 // new sem lançamento –
                   const std::nothrow_t&) throw( );             // veja o Item 49
```

Se você declarar `operator new`s em uma classe, ocultará todas essas formas padrão. A menos que queira impedir que as classes cliente usem essas formas, certifique-se de torná-las disponíveis além de quaisquer formas personalizadas de `operator new` que criar. Para cada `operator new` que você disponibilizar, certifique-se de também oferecer o `operator delete` correspondente. Se quiser que essas funções se comportem normalmente, apenas faça com que suas versões específicas de classe chamem as versões globais.

Uma maneira fácil de fazer isso é criando uma classe-base contendo todas as formas normais de `new` e de `delete`:

```
class StandardNewDeleteForms {
public:
  // new/delete normal
  static void* operator new(std::size_t size) throw(std::bad_alloc)
  { return ::operator new(size); }

  static void operator delete(void *pMemory) throw( )
  { ::operator delete(pMemory); }
```

```
    // new/delete de posicionamento
    static void* operator new(std::size_t size, void *ptr) throw( )
    { return ::operator new(size, ptr); }

    static void operator delete(void *pMemory, void *ptr) throw( )
    { ::operator delete(pMemory, ptr); }

    // new/delete sem lançamento
    static void* operator new(std::size_t size, const std::nothrow_t& nt) throw( )
    { return ::operator new(size, nt); }

    static void operator delete(void *pMemory, const std::nothrow_t&) throw( )
    { ::operator delete(pMemory); }
};
```

Os clientes que querem melhorar as formas padrão com formas personalizadas podem simplesmente usar herança e declarações `using` (veja o Item 33) para obter as formas padrão:

```
class Widget: public StandardNewDeleteForms {          // herda as formas padrão
public:
  using StandardNewDeleteForms::operator new;          // torna essas
  using StandardNewDeleteForms::operator delete;       // formas visíveis

  static void* operator new(std::size_t size,          // adiciona um new de
                            std::ostream& logStream)   // posicionamento personalizado
    throw(std::bad_alloc);

  static void operator delete(void *pMemory,           // adiciona o delete
                              ostream& logStream)      // de posicionamento
                                                       // correspondente
    throw( );
  ...
};
```

Lembretes

» Quando você escrever uma versão de posicionamento de `operator new`, lembre-se de escrever a versão de posicionamento correspondente de `operator delete`. Se não fizer isso, seu programa pode ter vazamentos de memória sutis e intermitentes.

» Quando você declarar versões de posicionamento de `new` e de `delete`, cuide para não ocultar sem querer as versões normais dessas funções.

CAPÍTULO 9

MISCELÂNEA

Bem-vindo ao capítulo que engloba todas as outras coisas não classificadas nos outros capítulos, denominado "Miscelânea". Existem apenas três itens aqui, mas não deixe esse número diminuto ou sua localização pouco glamourosa enganá-lo. Eles são importantes.

O primeiro item enfatiza que os avisos do compilador não devem ser desconsiderados, a menos que você não queira que seu sistema de software se comporte adequadamente. O segundo oferece uma visão geral do conteúdo da biblioteca padrão de C++, incluindo as novas e significativas funcionalidades introduzidas por TR1. Por fim, o último item fornece uma visão geral de Boost, provavelmente o mais importante site relacionado a C++ de propósito geral. Tentar escrever softwares em C++ eficaz sem as informações desses itens é, na melhor das hipóteses, uma batalha difícil.

Item 53: Preste atenção aos avisos do compilador

Muitos programadores costumam ignorar os avisos do compilador. Afinal, se o problema fosse sério, ele seria um erro, certo? Esse pensamento pode ser relativamente inofensivo em outras linguagens, mas em C++ é bom apostar que os escritores de compiladores têm um melhor entendimento que você sobre o que está acontecendo. Por exemplo, veja um erro que todo mundo faz uma hora ou outra:

```
class B {
public:
   virtual void f( ) const;
};

class D: public B {
public:
   virtual void f( );
};
```

A ideia é que `D::f` redefine a função virtual `B::f`, mas existe um equívoco: em `B`, `f` é uma função membro constante (`const`), mas, em `D`, ela não é declarada como constante. Um compilador que conheço diz o seguinte sobre isso:

```
Aviso: D::f( ) oculta virtual B::f( )
```

Muitos programadores inexperientes respondem a essa mensagem dizendo a si mesmos: "*É claro* que `D::f` oculta `B::f` – é isso que ele *deve* fazer!". Errado. Esse compilador está tentando dizer que `f` declarado em `B` não foi redeclarado em `D`; em vez disso, ele está sendo ocultado completamente (veja o Item 33 para ter uma descrição de por que isso acontece). Ignorar esse aviso do compilador levará, quase certamente, a um comportamento errôneo de programa, seguido de um monte de depuração para descobrir algo que o compilador já havia detectado em um primeiro momento.

Depois de ganhar experiência com as mensagens de aviso de um compilador em particular, você aprenderá a entender o que significam as diferentes mensagens (o que é, em geral, bastante diferente daquilo que elas *parecem* significar, infelizmente). Uma vez que tenha adquirido essa experiência, você pode escolher ignorar uma faixa completa de avisos, embora geralmente seja considerada uma boa prática escrever código compilado sem avisos, mesmo no nível mais alto de avisos. Independentemente disso, antes de ignorar um aviso, é importante que você tenha certeza de que entendeu exatamente o que ele está tentando dizer.

Enquanto estivermos no tópico de avisos, lembre-se de que os avisos são dependentes da implementação, então, não é uma boa ideia ser preguiçoso em sua programação e confiar totalmente que os compiladores vão identificar seus erros para você. O código de ocultamento de função acima, por exemplo, passa por um compilador diferente (mas amplamente usado) sem reclamação.

Lembretes

- Leve os avisos do compilador a sério e busque compilar sem avisos no nível máximo de avisos suportado pelos seus compiladores.
- Não fique dependente dos avisos de compilação, pois compiladores diferentes avisam sobre coisas diferentes. Migrar para um novo compilador pode eliminar algumas das mensagens de erro de que você costumava depender.

Item 54: Familiarize-se com a biblioteca padrão, incluindo TR1

O padrão para C++ (o documento que define a linguagem e sua biblioteca) foi ratificado em 1998. Em 2003, foi publicada uma pequena atualização de "correção de bugs". O comitê de padronização, entretanto, continua o seu trabalho e até a finalização deste texto, uma "Versão 2.0" continuava sendo aguardada. A incerteza em relação à data explica por que as pessoas normalmente se referem à próxima versão de C++ como "C++0x" – a versão 200x de C++. C++0x provavelmente incluirá alguns novos e interessantes recursos de linguagens, mas a maioria das novas

funcionalidades de C++ virá na forma de adições à biblioteca padrão. Já sabemos quais são algumas das novas funcionalidades da biblioteca, porque ela tem sido especificada em um documento conhecido como TR1 ("*Technical Report 1*" do Grupo de Trabalho da Biblioteca C++). O comitê de padronização se reserva o direito de modificar as funcionalidades do TR1 antes de ele ser oficialmente introduzido no C++0x, mas é improvável que haja mudanças significativas. Para todas as intenções e propósitos, o TR1 define o início de uma nova versão de C++ (que poderíamos chamar de padrão C++ 1.1). Não é possível ser um programador C++ eficaz sem conhecer a funcionalidade de TR1, pois ela é uma grande vantagem para todo o tipo de biblioteca e aplicação.

Antes de descrever o que existe no TR1, vale a pena revisar as partes principais da biblioteca C++ especificada no C++98.

- **A Biblioteca de Templates Padrão (STL – Standard Template Library)**, incluindo contêineres (`vector`, `string`, `map`, etc); iteradores; algoritmos (`find`, `sort`, `transform`, etc.); objetos função (`less`, `greater`, etc.); e vários adaptadores de contêineres e de objetos função (`stack`, `priority_queue`, `mem_fun`, `not1`, etc).

- **Iostreams**, incluindo suporte para buffers definidos pelo usuário, entrada e saída internacionalizada e os objetos predefinidos `cin`, `cout`, `cerr` e `clog`.

- **Suporte para internacionalização**, incluindo a habilidade de ter múltiplas localidades ativas. Tipos como `wchar_t` (normalmente 16 por caractere) e `wstring` (strings de `wchar_t`s) facilitam o trabalho com Unicode.

- **Suporte para processamento numérico**, incluindo templates para números complexos (`complex`) e vetores de valores puros (`valarray`).

- **Uma hierarquia de exceções**, incluindo a classe-base `exception`, suas classes derivadas `logic_error` e `runtime_error` e diversas classes que herdam delas.

- **Biblioteca padrão do C89**. Tudo o que estava na biblioteca padrão 1989 também está em C++.

Se você não conhecer algum dos itens acima, sugiro que você tire um tempo para ler sua referência de C++ favorita e resolver essa situação. TR1 especifica 14 novos componentes (ou seja, partes das funcionalidades da biblioteca). Todos estão no espaço de nomes `std`, mais precisamente no espaço de nomes aninhados `tr1`. O nome completo do componente de TR1 `shared_ptr` (veja abaixo) é, então, `std::tr1::shared_ptr`. Neste livro, costumo omitir `std::` quando estou discutindo componentes da biblioteca padrão, mas sempre uso o prefixo de TR1 nos componentes com `tr1::`.

Este livro mostra exemplos dos seguintes componentes TR1:

- Os **ponteiros espertos** `tr1::shared_ptr` e `tr1::weak_ptr`. Os ponteiros `tr1::shared_ptr`s agem como ponteiros predefinidos, mas eles rastreiam quantos `tr1::shared_ptr`s apontam para um objeto. Isso é conhecido como *contagem de referências*. Quando o último desses ponteiros é destruído (ou seja, quando a contagem de referências para um objeto se torna zero), o objeto é automaticamente apagado. Isso funciona bem para impedir vazamentos de recursos em estruturas de dados acíclicas, mas, se dois ou mais objetos contiverem `tr1::shared_ptr`s de tal forma que um ciclo seja formado, o ciclo pode manter a contagem de referência de cada objeto acima de zero, mesmo quando todos os ponteiros externos ao ciclo tiverem sido destruídos (ou seja, quando não é possível alcançar o grupo de objetos como um todo). É nesse caso que entram os `tr1::weak_ptr`s, os quais são projetados para agir como ponteiros indutores de ciclos em estruturas de dados baseadas em `tr1::shared_ptr` que, de outra forma, seriam acíclicas. Os ponteiros `tr1::weak_ptr`s não participam da contagem de referências. Quando o último `tr1::shared_ptr` para um objeto for destruído, o objeto é apagado, mesmo que `tr1::weak_ptr`s continuem a apontar para ele. Tais `tr1::weak_ptr`s são automaticamente marcados como inválidos, entretanto.

  `tr1::shared_ptr` pode ser o componente mais útil em TR1. Eu o uso diversas vezes neste livro, inclusive no Item 13, quando explico porque ele é tão importante. (O livro não contém usos de `tr1::weak_ptr`, desculpem-me.)

- **`tr1::function`,** que possibilita a representação de qualquer *entidade chamável* (ou seja, qualquer função ou objeto função) cuja assinatura condiz com uma assinatura alvo. Se quiséssemos possibilitar o registro de funções *callback* que recebessem um `int` e retornassem uma `string`, faríamos o seguinte:

```
void registerCallback(std::string func(int));          // o tipo do parâmetro é uma função
                                                       // que recebe um int e
                                                       // retorna uma string
```

  O nome de parâmetro `func` é opcional, então `registerCallback` poderia ser declarado da seguinte forma:

```
void registerCallback(std::string (int));              // como acima; o nome
                                                       // do parâmetro é omitido
```

  Observe que, aqui, "`std::string(int)`" é uma assinatura de função.

  `tr1::function` possibilita que `registerCallback` seja muito mais flexível, aceitando como seu argumento qualquer entidade chamável que receba um `int` ou *qualquer coisa em que um* `int` *possa ser convertido* e que retorne uma string ou *qualquer coisa que possa ser*

convertida em uma string. `tr1::function` recebe como parâmetro de template sua assinatura de função-alvo:

```
void registerCallback(std::tr1::function<std::string (int)> func);
                                                  // o parâmetro "func"
                                                  // receberá qualquer entidade chamável
                                                  // com uma assinatura condizente
                                                  // com "std::string (int)"
```

Esse tipo de flexibilidade é surpreendentemente útil, algo que tentei da melhor forma possível mostrar no Item 35.

- **`tr1::bind`,** que faz tudo o que os vinculadores `bind1st` e `bind2nd` da STL fazem e muito mais. Ao contrário dos vinculadores pré-TR1, `tr1::bind` funciona tanto com funções membro constantes quanto com não constantes; funciona com parâmetros por referência; e trata ponteiros para funções sem ajuda, então, não é preciso usar `ptr_fun`, `mem_fun` ou `mem_fun_ref` antes de chamar `tr1::bind`. Simplesmente, `tr1::bind` é um recurso de vinculação de segunda geração muito melhor que seu predecessor. Mostro um exemplo de seu uso no Item 35.

Divido os componentes TR1 restantes em dois conjuntos. O primeiro, oferece funcionalidades autocontidas bastante discretas.

- **Tabelas de dispersão (hash tables)** usadas para implementar conjuntos, multiconjuntos, mapas e multimapas. Cada novo contêiner possui uma interface modelada quanto ao seu correspondente pré-TR1. O mais surpreendente sobre as tabelas de dispersão de TR1 são seus nomes: `tr1::unordered_set`, `tr1::unordered_multiset`, `tr1::unordered_map` e `tr1::unordered_multimap`. Esses nomes enfatizam que, ao contrário dos conteúdos de um `set`, `multiset`, `map` ou `multimap`, os elementos em um contêiner TR1 baseado em dispersão não estão em qualquer ordem previsível.

- **Expressões regulares**, incluindo a capacidade de realizar operações de busca e de substituição baseadas em expressões regulares em cadeias de caracteres para iterar por meio de cadeias de caracteres de casamento em casamento, etc.

- **Tuplas**, uma generalização interessante do template `pair` que já está na biblioteca padrão. Enquanto os objetos `pair` podem manter apenas dois objetos, os objetos `tr1::tuple` podem manter um número arbitrário deles. Programadores Expat Python e Eiffel, regojizem-se! Uma pequena parte de sua terra natal agora faz parte de C++.

- **`tr1::array`**, essencialmente um vetor "STLificado", ou seja, um vetor que suporta funções membro como início (`begin`) e fim (`end`). O tamanho de um `tr1::array` é fixado durante a compilação; o objeto não usa memória dinâmica.

- **`tr1::mem_fn`**, uma maneira sintaticamente uniforme de adaptar ponteiros de funções membros. Assim como `tr1::bind` fornece e estende as capacidades de `bind1st` e `bind2nd` do C++98, `tr1::mem_fn` fornece e estende as capacidades de `mem_fun` e `mem_fun_ref`.

- **`tr1::reference_wrapper`**, um recurso para fazer as referências agirem um pouco mais como objetos. Dentre outras coisas, isso possibilita criar contêineres que agem como se mantivessem referências. (Na realidade, os contêineres podem manter apenas objetos ou ponteiros.)

- Recursos de **geração de números aleatórios** que são muito superiores à função `rand` que C++ herda da biblioteca padrão de C.

- **Funções especiais matemáticas**, incluindo polinômios de Laguerre, Funções de Bessel, integrais elípticas completas e muito mais.

- **Extensões de compatibilidade com C99**, uma coleção de funções e de templates projetados para trazer muitos novos recursos de biblioteca do C99 para C++.

O segundo conjunto de componentes TR1 consiste em tecnologias de suporte para técnicas de programação de templates mais sofisticadas, incluindo a metaprogramação por templates (veja o Item 48).

- **Traits de tipo**, um conjunto de classes de trait (veja o Item 47) para fornecer informação em tempo de compilação acerca de tipos. Dado um tipo `T`, os *traits* de tipo de TR1 podem revelar se `T` é um tipo predefinido, se oferece um destrutor virtual, se é uma classe vazia (veja o Item 39), se é implicitamente conversível a algum outro tipo `U` e muito mais. Os traits de tipo de TR1 também podem revelar o alinhamento apropriado para um tipo, uma informação crucial para os programadores que estejam escrevendo funções personalizadas de alocação de memória (veja o Item 50).

- **`tr1::result_of`**, um template para deduzir os tipos de retorno de chamadas à função. Quando estiver escrevendo templates, é importante conseguir referenciar o tipo do objeto retornado de uma chamada a uma função (template), mas o tipo de retorno pode depender dos tipos dos parâmetros da função de maneiras complexas. `tr1::result_of` facilita a tarefa de referenciar os tipos de retornos de funções. `tr1::result_of` é usado em diversos locais no TR1 propriamente dito.

Embora as capacidades de algumas partes de TR1 (em especial, `tr1::bind` e `tr1::mem_fn`) forneça as mesmas capacidades de alguns componentes pré-TR1, o TR1 é uma adição pura à biblioteca padrão. Nenhum componente TR1 substitui um componente existente, então o código legado escrito com construções pré-TR1 continua sendo válido.

O TR1 propriamente dito é apenas um documento*. Para tirar vantagem da funcionalidade que ele especifica, você precisa acessar o código que o implementa. No fim das contas, o código virá juntamente com os compiladores, mas, enquanto escrevo este livro, existe uma boa chance de que, se você procurar componentes TR1 em suas implementações da biblioteca padrão, pelo menos alguns deles estejam faltando. Felizmente, existe outro lugar para realizar essa busca: 10 dos 14 componentes do TR1 se baseiam em bibliotecas disponíveis gratuitamente a partir de Boost (veja o Item 55); então, esse é um excelente recurso para funcionalidade similar a TR1. Eu digo "similar a TR1" porque, embora muitas das funcionalidades de TR1 se baseiem em bibliotecas Boost, existem locais nos quais a funcionalidade de Boost não é exatamente idêntica à especificação do TR1. É possível que, quando você ler isso, Boost não só tenha implementações em conformidade com TR1 para os componentes TR1 que evoluíram a partir de bibliotecas Boost, como também ofereça implementações dos quatro componentes TR1 que não foram baseados no trabalho de Boost.

Se você quiser usar as bibliotecas de Boost similares às de TR1 como subterfúgio até que os compiladores forneçam suas próprias implementações de TR1, pode usar um truque de espaço de nomes. Todos os componentes Boost estão no espaço de nomes `boost`, mas os componentes TR1 supostamente devem estar em `std::tr1`. Você pode dizer aos seus compiladores que tratem referências a `std::tr1` como se fossem referências a `boost`. Veja como fazer isso:

```
namespace std {
  namespace tr1 = ::boost;                  // o espaço de nomes std::tr1 é um apelido
}                                           // para o espaço de nomes boost
```

Tecnicamente, isso o coloca no mundo do comportamento indefinido, porque, como o Item 25 explica, você não pode adicionar nada ao espaço de nomes `std`. Na prática, você provavelmente não terá problema algum. Quando os compiladores fornecerem suas próprias implementações do TR1, tudo o que você precisará fazer é eliminar o apelido de espaço de nomes acima; o código que se refere ao espaço de nomes `std::tr1` continuará sendo válido.

Provavelmente, a parte mais importante de TR1 que não se baseia em bibliotecas Boost são as tabelas de dispersão, mas elas já estão disponíveis há muitos anos, de muitas fontes diferentes sob os nomes `hash_set`, `hash_multiset`, `hash_map` e `hash_multimap`. Existe uma boa chance de que as bibliotecas que vêm junto com seus compiladores já contenham esses templates. Se não contiverem, utilize seu mecanismo de busca preferido e pesquise esses nomes (bem como suas correspondentes TR1), porque certamente você encontrará diversas fontes para elas, tanto comerciais quanto gratuitas.

*Enquanto escrevo este livro, no início de 2005, o documento não está finalizado, e sua URL está sujeita a mudanças. Logo, sugiro que você consulte a Página de Informações sobre TR1 de C++ Eficaz, em http://aristeia.com/EC3E/TR1_info.html. Essa URL permanecerá estável.

Lembretes

» A funcionalidade principal da biblioteca padrão de C++ consiste na STL, em `iostreams` e em *locales*. A biblioteca padrão C99 também é incluída.

» TR1 adiciona suporte para ponteiros espertos (como `tr1::shared_ptr`), ponteiros de funções generalizados (`tr1::function`), contêineres baseados em dispersão, expressões regulares e dez outros componentes.

» O TR1 propriamente dito é apenas uma especificação. Para tirar proveito do TR1, você precisa de uma implementação. Uma fonte para implementações de componentes TR1 é chamada Boost.

Item 55: Familiarize-se com Boost

Você está buscando uma coleção de bibliotecas de alta qualidade, de código aberto, independente de plataforma e de compilador? Procure Boost. Está interessado em se unir a uma comunidade de desenvolvedores C++ ambiciosos e talentosos trabalhando no projeto e na implementação de uma biblioteca que representa o que há de melhor? Procure Boost. Quer ter uma ideia do que C++ será no futuro? Procure Boost. Boost é uma comunidade de desenvolvedores C++ e um conjunto de bibliotecas C++ disponível para download. Seu site é http:://boost.org. Você deve adicioná-lo aos favoritos.

Existem muitas organizações e sites sobre C++, é claro, mas Boost possui duas vantagens que nenhuma outra organização tem. Primeiro, ela tem uma relação forte e influente com o comitê de padronização de C++. Boost foi fundada pelos membros do comitê, e continua existindo uma grande circulação entre os membros de Boost e do comitê. Além disso, Boost sempre teve como um de seus objetivos agir como ambiente de testes para capacidades que poderiam ser adicionadas a C++ padrão. Um resultado desse relacionamento é que, das 14 novas bibliotecas introduzidas em C++ pelo TR1 (veja o Item 54), mais de dois terços se baseiam no trabalho feito em Boost.

Uma segunda característica especial de Boost é seu processo para aceitar bibliotecas, que se baseia em revisões por pares. Se quiser contribuir com uma biblioteca para Boost, você começa enviando um e-mail para a lista de desenvolvedores de Boost para levantar interesse acerca da biblioteca e iniciar o processo de exame preliminar de seu trabalho. Então, inicia-se um ciclo que o site resume como "Discutir, refinar, ressubmeter. Repetir até satisfazer."

Por fim, você decide que sua biblioteca está pronta para uma submissão formal. Um gerente de revisão confirma se sua biblioteca atende aos requisitos mínimos de Boost. Por exemplo, ela deve ser compilada em pelo menos dois compiladores (para demonstrar portabilidade nominal), e você deve atestar que a biblioteca pode ser disponibilizada por meio de uma licença aceitável (por exemplo, a biblioteca deve permitir o uso comercial e o não comercial gratuitamente). Então seu envio é disponibilizado para

a comunidade Boost para revisão oficial. Durante o período de revisão, voluntários analisam os materiais de sua biblioteca (código-fonte, documentos de projeto, documentação de usuário, etc) e consideram questões como as seguintes:

- O projeto e a implementação são bons? Até que ponto?
- O código é portável entre compiladores e sistemas operacionais?
- É provável que a biblioteca seja usada por público-alvo, ou seja, pessoas que trabalham no domínio que a biblioteca trata?
- A documentação é clara, completa e precisa?

Esses comentários são enviados para uma lista de e-mails de Boost, de forma que os revisores e outros possam ver e responder às colocações de cada um. No final do processo de revisão, o gerente de revisão decide se sua biblioteca é aceita, aceita condicionalmente ou rejeitada.

As revisões por pares fazem um bom trabalho, mantendo as bibliotecas mal escritas de fora de Boost, mas também ajudam a educar os autores das bibliotecas sobre considerações que devem ser feitas no projeto, na implementação e na documentação de bibliotecas de qualidade industrial com funcionamento em plataformas diferentes. Muitas bibliotecas requerem mais de uma revisão oficial antes de serem declaradas merecedoras de um aceite.

Boost contém dezenas de bibliotecas, e mais são adicionadas de maneira contínua. De tempos em tempos, algumas bibliotecas são removidas, em geral porque sua funcionalidade foi superada por uma nova biblioteca que oferece uma maior funcionalidade ou um melhor projeto (ou seja, uma que seja mais flexível ou mais eficiente).

As bibliotecas variam amplamente em termos de tamanho e de escopo. Em um extremo estão bibliotecas que, em teoria, requerem apenas algumas poucas linhas de código (mas são muito maiores após ser adicionado suporte para tratamento de erros e portabilidade). Uma dessas bibliotecas é chamada de **Conversion**, a qual fornece operadores de conversão explícita mais seguros ou mais convenientes. Sua função `numeric_cast`, por exemplo, lança uma exceção se a conversão de um valor numérico de um tipo para outro causar um transbordamento (positivo ou negativo) ou um problema similar, e `lexical_cast` permite converter explicitamente qualquer tipo que suporte `operator<<` em uma cadeia de caracteres – muito útil para diagnósticos, logging, etc. No outro extremo estão bibliotecas que oferecem capacidades extensas, para as quais já foram escritos muitos livros. Dentre essas bibliotecas, estão a **Biblioteca de Grafos Boost** (para programação com estruturas arbitrárias de grafos) e a **Biblioteca de Metaprogramação Boost** (*Boost MPL Library*).

As diversas bibliotecas de Boost tratam de uma abundância de tópicos, agrupados em uma dezena de categorias gerais. Essas categorias incluem:

- **Processamento de cadeias de caracteres e de texto**, incluindo bibliotecas para formatação similar a `printf` segura em relação a tipos, expressões regulares (a base para funcionalidade similar em TR1 – veja o Item 54), análise léxica e análise sintática.

- **Contêineres**, incluindo bibliotecas para vetores de tamanhos fixos com uma interface no estilo da STL (veja o Item 54), conjuntos de bits de tamanhos variados e vetores multidimensionais.

- **Objetos função e programação em ordem mais alta**, incluindo diversas bibliotecas que foram usadas como base para a funcionalidade do TR1. Uma biblioteca interessante é a biblioteca Lambda, que facilita tanto a criação instantânea de objetos função que você provavelmente nem se dará conta do que está fazendo:

```
using namespace boost::lambda;                          // torna a funcionalidade de
                                                        // boost:lambda visível

std::vector<int> v;

...

std::for_each(v.begin( ), v.end( ),                     // para cada elemento x em
              std::cout << _1 * 2 + 10 << "\n");        // v, imprima x*2+10;
                                                        // "_1" é o local de manutenção
                                                        // da biblioteca Lambda
                                                        // para o elemento atual
```

- **Programação genérica**, incluindo um amplo conjunto de classes de traits. (Veja o Item 47 para obter mais informações sobre traits.)

- **Metaprogramação por templates** (TMP – veja o Item 48), incluindo uma biblioteca para asserções em tempo de compilação, bem como a Biblioteca de Metaprogramação por Templates de Boost. Dentre as coisas interessantes da MPL está o suporte para estruturas de dados similares a STL de entidades em tempo de compilação como *tipos*, por exemplo,

```
// cria um contêiner similar a uma lista em tempo de compilação de três tipos (float,
// double e long double) e chama o contêiner de "floats"
typedef boost::mpl::list<float, double, long double> floats;

// cria uma lista de tipos em tempo de compilação consistindo nos tipos em
// "floats" mais "int" inserido na frente; chama o novo contêiner de "types"
typedef boost::mpl::push_front<floats, int>::type types;
```

  Esses contêineres de tipos (frequentemente conhecidos como *listas de tipos*, embora também possam se basear em `mpl::vector`, bem como em `mpl::list`) abrem as portas para uma ampla faixa de aplicações de TMP importantes e poderosas.

- **Matemática e funções numéricas**, incluindo bibliotecas para números racionais; octônios e quatérnios; computações de maior divisor comum e de menor multiplicador comum; e números aleatórios (outra biblioteca que influenciou funcionalidades relacionadas no TR1).

- **Correção e testes**, incluindo bibliotecas para formalizar interfaces de template implícitas (veja o Item 41) e facilitar a programação dirigida por testes.
- **Estruturas de dados**, incluindo bibliotecas para uniões seguras em relação a tipos (ou seja, armazenando "quaisquer" tipos variáveis) e a biblioteca de tuplas que levou à funcionalidade correspondente do TR1.
- **Suporte interlinguagens**, incluindo uma biblioteca para permitir interoperabilidade facilitada entre C++ e Python.
- **Memória**, incluindo a biblioteca Pool para alocadores de tamanho fixo de alto desempenho (veja o Item 50); e uma variedade de ponteiros espertos (veja o Item 13), incluindo (mas não limitado a) os ponteiros espertos em TR1. Um desses ponteiros espertos que não estão em TR1 é `scoped_array`, um ponteiro esperto parecido com `auto_ptr` para vetores alocados dinamicamente; o Item 44 mostra um exemplo de uso.
- **Miscelânea**, incluindo bibliotecas para verificação de CRC, manipulação de datas e horas, e para percorrer sistemas de arquivos.

Lembre-se, isso é apenas parte das bibliotecas que você encontrará em Boost. Essa exaustiva não é uma lista.

Boost oferece bibliotecas que fazem muitas coisas, mas ela não cobre todos os domínios de programação. Por exemplo, não existe uma biblioteca para desenvolvimento de interfaces gráficas com o usuário (GUIs), nem existe uma para a comunicação com bases de dados. Pelo menos não existem agora – não enquanto escrevo este livro. No momento em que você o estiver lendo, entretanto, talvez exista uma. A única maneira de saber com certeza é verificando. Eu sugiro que você faça isso agora: http://boost.org. Mesmo que não encontre exatamente o que está procurando, certamente você encontrará algo interessante lá.

Lembretes

» Boost é uma comunidade e um site para o desenvolvimento de bibliotecas C++ gratuitas, de código aberto, revisadas por pares. Boost tem um papel influente na padronização de C++.

» Boost oferece implementações de muitos componentes do TR1, mas também oferece muitas outras bibliotecas.

ÍNDICE

Os operadores são listados em *operator*. Ou seja, `operator<<` é listado em `operator<<`, não em `<<`, etc.

As classes, estruturas e templates de classes ou de estruturas de exemplo são indexados em *classes/templates de exemplo*. As funções e templates de funções de exemplo são indexados em *funções/templates* de exemplo.

C

D

E

F

G

H

I

J

K

L

M

N

0

P

Q

R

S

T

U

V

W

X

Z