Neste capítulo, exploramos as construções de linguagens de programação que suportam abstrações de dados. Dentre as novas ideias dos últimos 50 anos nas metodologias e no projeto de linguagens de programação, a abstração de dados é uma das mais profundas.

Começamos discutindo o conceito geral de abstração em programação e em linguagens de programação. A abstração de dados é então definida e ilustrada com um exemplo. Esse tópico é seguido por descrições do suporte para abstrações de dados em Ada, C++, Java, C# e Ruby. Implementações da mesma abstração de dados de exemplo são apresentadas em Ada, C++, Java C# e Ruby para mostrar as similaridades e as diferenças no projeto dos recursos de linguagem que suportam essas abstrações. A seguir, as capacidades de Ada, C++, Java 5.0 e C# 2005 para construir tipos de dados abstratos são discutidas.

Construções que suportam tipos de dados abstratos são encapsulamentos dos dados de operações em objetos do tipo. Encapsulamentos que contêm múltiplos tipos são necessários para a construção de programas maiores. Esses encapsulamentos e as questões associadas aos espaços de nomes também são discutidos neste capítulo.

As classes de C++, Java e C# tratam suas variáveis de instância e de classe de forma diferente das variáveis definidas em seus métodos. O escopo de uma variável definida em um método começa em sua definição. Entretanto, independentemente de onde uma variável de instância ou de classe é definida em uma classe, seu escopo é a classe inteira.

11.1 O CONCEITO DE ABSTRAÇÃO

Uma **abstração** é uma visão ou representação de uma entidade que inclui apenas os atributos mais significativos. De um modo geral, a abstração permite que alguém colete exemplares de entidades em grupos nos quais seus atributos comuns não precisam ser considerados. Por exemplo, suponha que definíssemos aves como criaturas com os seguintes atributos: duas asas, duas pernas, um rabo e penas. Então, se dissermos que um corvo é uma ave, uma descrição de um corvo não precisa incluir esses atributos. O mesmo ocorre para os piscos-de-peito-ruivo, pardais e pica-paus de barriga amarela. Esses atributos comuns nas descrições de espécies específicas de pássaros podem ser abstraídos. Dentro de uma espécie em particular, apenas os atributos que a distinguem precisam ser considerados, resultando em uma simplificação significativa das descrições. Uma visão menos abstrata de uma espécie, aquela de um pássaro, pode ser considerada quando for necessário ver um alto nível de detalhes.

No mundo das linguagens de programação, a abstração é uma arma contra a complexidade da programação; seu propósito é simplificar o processo de programação. É uma arma efetiva, pois permite que os programadores foquem em atributos essenciais, enquanto ignoram os subordinados.

Os dois tipos fundamentais de abstração nas linguagens de programação contemporâneas são a abstração de processos e a abstração de dados.

O **conceito de abstração de processo** está dentre os mais antigos no projeto de linguagens de programação – Plankalkül já suportava a abstração de processo. Todos os subprogramas são abstrações de processo, porque fornecem uma maneira pela qual um programa especifica um processo, sem fornecer os detalhes de como ele é realizado (ao menos no programa chamador). Por exemplo, quando um programa precisa ordenar um vetor de objetos de dados numéricos de algum tipo, ele normalmente usa um subprograma para o processo de ordenação. No momento em que um processo de ordenação é necessário, uma sentença como

```
sortInt(list, listLen)
```

é colocada no programa. Essa chamada é uma abstração do processo de ordenação real, cujo algoritmo não é especificado. A chamada é independente do algoritmo implementado no subprograma chamado.

No caso do subprograma `sortInt`, os únicos atributos essenciais são o nome do vetor a ser ordenado, o tipo de seus elementos, o tamanho do vetor e o fato de que a chamada a `sortInt` resulta na ordenação do vetor. O algoritmo em particular que `sortInt` implementa é um atributo não essencial para o usuário, que precisa ver apenas o nome e o protocolo do subprograma de ordenação para ser capaz de usá-lo.

A evolução da abstração de dados necessariamente seguiu a partir das abstrações de processo, visto que uma parte integral e central de todas as abstrações de dados são suas operações, definidas como abstrações de processos.

11.2 INTRODUÇÃO À ABSTRAÇÃO DE DADOS

Sintaticamente, um tipo de dados abstrato é um invólucro que inclui apenas a representação de dados de um tipo de dados específico e os subprogramas que fornecem as operações para esse tipo. Por meio de controles de acesso, detalhes desnecessários do tipo podem ser ocultados de unidades externas ao invólucro que o usam. Unidades de programa que usam um tipo de dados abstrato podem declarar variáveis de tal tipo, mesmo que a representação real seja deles ocultada. Um exemplar de um tipo de dados abstrato é chamado de um **objeto**.

Uma das motivações para a abstração de dados é similar à motivação para a abstração de processos: é uma arma contra a complexidade; uma forma de tornar programas grandes e/ou complicados mais gerenciáveis. Outras motivações e as vantagens dos tipos de dados abstratos são discutidas posteriormente nesta seção.

A programação orientada a objetos, descrita no Capítulo 12, é uma melhoria do uso de abstração de dados em desenvolvimento de software, e a abstração de dados é um de seus componentes mais importantes.

11.2.1 Ponto flutuante como um tipo de dados abstrato

O conceito de um tipo de dados abstratos não é uma evolução recente. Todos os tipos de dados pré-definidos, mesmo os do Fortran I, são abstratos, apesar de raramente serem chamados assim. Por exemplo, considere um tipo de dados de ponto flutuante. A maioria das linguagens inclui ao menos um desses tipos. Um tipo de ponto flutuante fornece uma maneira de criar variáveis para dados de ponto flutuante e um conjunto de operações aritméticas para manipular objetos do tipo.

Tipos de ponto flutuante em linguagens de alto nível empregam um conceito chave na abstração de dados: ocultamento de informação. O formato real do valor de dado em uma célula de memória de ponto flutuante é oculto do usuário, e as únicas operações disponíveis são as fornecidas pela linguagem. Não é permitido ao usuário criar novas operações em dados do tipo, exceto aquelas que puderem ser construídas usando as operações pré-definidas. O usuário não pode manipular diretamente as partes da representação real dos objetos porque essa representação é oculta. É esse o recurso que permite a portabilidade entre implementações de uma linguagem, mesmo que as implementações possam usar representações diferentes para tipos de dados em particular. Por exemplo, antes de o padrão de representações de ponto flutuante IEEE 754 aparecer no meio dos anos 1980, existiam diversas representações sendo usadas por diferentes arquiteturas de computadores. Entretanto, essa variação não impedia os programas que usavam tipos de ponto flutuante de serem portáveis para as várias arquiteturas.

11.2.2 Tipos de dados abstratos definidos pelo usuário

Um tipo de dados abstrato deve fornecer as mesmas características fornecidas por tipos definidos na linguagem, como um de ponto flutuante: (1) uma definição de tipo que permita às unidades de programa declararem variáveis do tipo, mas que oculte a representação de seus objetos; e (2) um conjunto de operações para manipular os objetos.

Agora, definimos formalmente um tipo de dados abstrato no contexto de tipos definidos pelo usuário. Um **tipo de dados abstrato** satisfaz as duas condições a seguir:

- As declarações do tipo e os protocolos das operações em objetos do tipo, que fornecem sua interface, são contidos em uma única unidade sintática. A interface do tipo não depende da representação dos objetos ou da implementação das operações. A implementação do tipo e de suas operações podem estar na mesma unidade sintática ou em uma unidade separada. Além disso, outras unidades de programa podem criar variáveis do tipo definido.

- A representação dos objetos do tipo é ocultada das unidades de programa que o usam, então as únicas operações diretas possíveis nesses objetos são aquelas fornecidas na definição do tipo.

A principal vantagem de empacotar as declarações do tipo e suas operações em uma única unidade sintática é fornecer um método de organizar um programa em unidades lógicas possíveis de serem compiladas separadamente. Ter a implementação do tipo e suas operações em uma unidade sintática diferente mantém as especificações e suas implementações de maneira separada. Unidades de programa que usam um tipo de dados abstrato específico, chamadas de **clientes** desse tipo, precisam ver a especificação, mas não pode ser permitido que elas vejam a implementação. Se tanto as declarações quanto as definições de tipos e operações estão na mesma unidade sintática, deve existir alguma forma de ocultar dos clientes as partes da unidade que especifiquem as definições.

A vantagem de a interface não depender da representação ou da implementação das operações é permitir que a representação e/ou a implementação sejam modificadas sem mudanças aos clientes do tipo.

Um benefício importante do ocultamento de informação é um aumento na confiabilidade. Clientes não podem manipular as representações subjacentes dos objetos diretamente, seja intencionalmente ou por acidente, aumentando a integridade de tais objetos. Objetos podem ser modificados apenas por meio das operações fornecidas.

11.2.3 Um exemplo

Suponha um tipo de dados abstrato construído para uma pilha que com as seguintes operações abstratas:

create(stack)	Cria e possivelmente inicializa um objeto de pilha
destroy(stack)	Libera o armazenamento para a pilha
empty(stack)	Um predicado ou função booleana que retorna verdadeiro (*true*) se a pilha especificada é vazia e falso (*false*) caso contrário.
push(stack, element)	Insere o elemento especificado na pilha especificada
pop(stack)	Remove o elemento do topo da pilha especificada
top(stack)	Retorna uma cópia do elemento do topo da pilha especificada

Note que algumas implementações de tipos de dados abstratos não requerem as operações de criação e destruição. Por exemplo, simplesmente definir uma variável como de um tipo de dados abstrato pode implicitamente criar a estrutura de dados subjacente e inicializá-la. O armazenamento para essa variável pode ser implicitamente liberado no final do escopo da variável.

Um cliente do tipo pilha poderia ter uma sequência de código como:

```
...
create(stk1);
push(stk1, color1);
push(stk1, color2);
if(! empty(stk1))
  temp = top(stk1);
...
```

Suponha que a implementação original da abstração pilha use uma representação com adjacência (que implementa uma pilha em um vetor). Posteriormente, por problemas de gerenciamento de memória com a representação usando adjacência, ela é modificada para uma representação baseada em lista encadeada. Como a abstração de dados foi usada, essa mudança pode ser feita no código que define o tipo pilha, mas nenhuma alteração será necessária em quaisquer dos clientes da abstração pilha. Em particular, a sequência de código de exemplo não precisa ser modificada. É claro, uma mudança no protocolo de qualquer operação pode requerer alterações nos clientes.

11.3 QUESTÕES DE PROJETO PARA TIPOS DE DADOS ABSTRATOS

Um recurso para definir tipos de dados abstratos em uma linguagem deve fornecer uma unidade sintática que envolva a declaração do tipo e os protótipos dos subprogramas que implementam as operações em objetos do tipo. Deve ser possível torná-los visíveis aos clientes da abstração, permitindo que declarem variáveis do tipo abstrato e manipulem seus valores. Apesar de o nome do tipo precisar ter visibilidade externa, a representação deve ser oculta. A representação do tipo e as definições dos subprogramas que implementam as operações podem aparecer dentro ou fora dessa unidade sintática.

Poucas, se é que existirão, operações gerais pré-definidas devem ser fornecidas para objetos de tipos de dados abstratos, além daquelas dadas com a definição de tipo. Simplesmente não existem muitas operações que se apliquem a uma ampla faixa de tipos de dados abstratos. Dentre essas, estão operações para atribuição e comparações para igualdade e diferença. Se a linguagem não permite que os usuários sobrecarreguem a atribuição, ela deve ser pré-definida. Comparações de igualdade e diferença devem ser pré-definidas em alguns casos, mas não em outros. Por exemplo, se o tipo é implementado como um ponteiro, a igualdade pode significar a igualdade de ponteiros, mas o usuário pode querer que seja a das estruturas referenciadas pelos ponteiros.

Algumas operações são necessárias por muitos tipos de dados abstratos, mas como não são universais, devem ser fornecidas pelo projetista do tipo. Dentre elas estão os iteradores, métodos de acesso, construtores e destrutores. Iteradores foram discutidos no Capítulo 8. Os métodos de acesso fornecem um caminho para dados ocultos para acesso direto pelos clientes. Os construtores são usados para a inicialização de partes de objetos recém-criados. Os destrutores servem para recuperar armazenamento no monte para ser usado por partes de objetos de tipos de dados abstratos em linguagens que não fazem recuperação implícita de armazenamento.

Conforme mencionado, o invólucro para um tipo de dados abstrato define um único tipo e suas operações. Muitas linguagens contemporâneas, incluindo C++, Java e C#, suportam tipos de dados abstratos diretamente. Um método alternativo, usado por Ada, é fornecer uma construção de encapsulamento mais generalizada que possa definir qualquer número de entidades, qualquer das quais pode ser especificada para ser visível de fora de sua unidade. Esses invólucros não são tipos de dados abstratos, mas generalizações, que podem ser usadas para definir tipos de dados abstratos. Apesar de discutirmos a construção de encapsulamento de Ada nesta seção, a tratamos como um encapsulamento mínimo para tipos de dados únicos. Encapsulamentos generalizados são discutidos na Seção 11.6.

Então, a primeira questão de projeto para tipos de dados abstratos é a forma do contêiner para a interface do tipo. A segunda é se os tipos de dados abstratos podem ser parametrizados. Por exemplo, se a linguagem suporta tipos de dados abstratos parametrizados, alguém deve projetar um tipo de dados abstrato para filas que poderiam armazenar elementos de qualquer tipo. Tipos de dados abstratos parametrizados são discutidos na Seção 11.5. A terceira questão de projeto é quais controles de acesso são fornecidos e como são especificados.

11.4 EXEMPLOS DE LINGUAGEM

O conceito de abstração de dados tem suas origens no SIMULA 67, apesar de essa linguagem não fornecer suporte completo para tipos de dados abstratos. Nesta seção, descrevemos o suporte para abstração fornecido por Ada, C++, Java, C# e Ruby.

11.4.1 Tipos de dados abstratos em Ada

Ada fornece uma construção de encapsulamento que pode ser usada para definir um tipo de dados abstrato único, incluindo a habilidade de ocultar sua representação. Ada foi uma das primeiras linguagens a oferecer suporte completo para esses tipos de dados.

11.4.1.1 Encapsulamento

As construções de encapsulamento em Ada são chamadas de **pacotes**, que podem ter duas partes, chamadas de **pacote de especificação** (que fornece a interface do encapsulamento) e **pacote de corpo** (que fornece a implementação da maioria das entidades nomeadas no pacote de especificação associado, ou de todas elas). Nem todos os pacotes têm a parte do corpo (os que encapsulam apenas tipos e constantes não têm ou não precisam de um corpo).

Um pacote de especificação e seu pacote de corpo associado compartilham o mesmo nome. A palavra reservada `body` em um cabeçalho de pacote o identifica como de corpo. Pacotes de especificação e de corpo podem ser compilados separadamente, desde que o de especificação seja compilado primeiro.

11.4.1.2 Ocultamento de informação

O projetista de um pacote Ada que define um tipo de dados pode escolher tornar o tipo inteiramente visível aos clientes ou fornecer apenas as informações de interface. É claro, se a representação não é oculta, o tipo definido não é um tipo de dados abstrato. Existem duas abordagens para ocultar a representação dos clientes no pacote de especificação. Uma é incluir duas seções no pacote de especificação – uma na qual as entidades são visíveis aos clientes e outra que oculta seus conteúdos. Para um tipo de dados abstrato, uma declaração aparece na parte visível da especificação, fornecendo apenas o nome do tipo e a informação de que sua representação é oculta. A representação aparece em uma parte da especificação chamada de **privada**, introduzida pela palavra reservada `private`. A cláusula privada sempre aparece no final do pacote de especificação. Ela é visível para o compilador, mas não para as unidades de programa cliente.

A segunda maneira de ocultar a representação é definir o tipo de dados abstrato como um ponteiro e fornecer a definição da estrutura apontada no pacote de corpo, cujo conteúdo inteiro é oculto para os clientes.

A seguir, temos um exemplo da primeira abordagem para ocultar a representação de um tipo de seus clientes. Suponha que um tipo de dados abstrato chamado `Node_Type` será definido em um pacote. `Node_Type` é declarado na parte visível do pacote de especificação sem seus detalhes de representação, como em

```
type Node_Type is private;
```

Tipos declarados como privados são chamados de **tipos privados** e têm operações pré-definidas para atribuição e comparações para igualdade e para diferença. Qualquer outra operação deve ser declarada no pacote de especificação que definiu o tipo.

Note que na cláusula privada do exemplo a seguir, a declaração de `Node_Type` é repetida, mas com a definição de tipo completa:

```
package Linked_List_Type is
  type Node_Type is private;
  ...
  private
    type Node_Type;
    type Ptr is access Node_Type;
    type Node_Type is
      record
      Info : Integer;
      Link : Ptr;
      end record;
  end Linked_List_Type;
```

A cláusula privada desse pacote tem tanto uma declaração quanto uma definição de `Node_Type`. A declaração é necessária por causa da referência a `Node_Type` na definição de `Ptr`, que deve preceder a definição de `Node_Type`. Como eles são definidos na cláusula privada, nem `Info` nem `Link` são visíveis para os clientes de `Linked_List_Type`.

Se nenhuma das entidades em um pacote será oculta, não existe um objetivo ou uma necessidade para a parte privada da especificação. É claro, tal pacote não pode definir um tipo de dados abstrato.

A razão pela qual a representação de um tipo aparece no pacote de especificação tem tudo a ver com questões de compilação. Um cliente pode ver apenas o pacote de especificação (não o pacote de corpo), mas o compilador deve ser capaz de alocar objetos do tipo exportado quando estiver compilando o cliente. Além disso, o cliente é compilável quando apenas o pacote de especificação para o tipo de dados abstrato tenha sido compilado e estiver presente. Logo, o compilador deve ser capaz de determinar o tamanho de um objeto a partir do pacote de especificação. Assim, a representação do tamanho deve ser visível ao compilador, mas não ao código do cliente. Essa é exatamente a situação especificada pela cláusula privada em um pacote de especificação.

De certa forma, é problemático que o pacote de especificação forneça parte dos detalhes de implementação (a definição dos dados) acima, enquanto os detalhes de implementação restantes (a definição das operações) estão no pacote de corpo. Seria muito mais claro se a especificação fornecesse apenas a interface e o corpo fornecesse todos os detalhes de implementação. Esse problema pode ser aliviado tornando o tipo de dados abstrato um ponteiro, como no pacote a seguir:

```
package Linked_List_Type is
  type Node is private;
  function Create_Node() return Node;
  private
    type Node_Record;
    type Node is access Node_Record;
end Linked_List_Type;
```

C++: nascimento, onipresença e críticas comuns

BJARNE STROUSTRUP

Bjarne Stroustrup é o projetista e implementador original de C++ e autor de *The C++ Programming Language* e *The Design and Evolution of C++*. Seus interesses em pesquisa incluem sistemas distribuídos, simulação, projeto, programação e linguagens de programação. Dr. Stroustrup é professor de Ciência da Computação na Faculdade de Engenharia da Universidade do Texas A&M e está ativamente envolvido na padronização ANSI/ISO do C++. Após mais de duas décadas na AT&T, mantém uma ligação com os laboratórios da companhia, pesquisando como membro do Laboratório de Pesquisa em Informação e Sistemas de Software. Bjarme é *ACM Fellow*, *AT&T Bell Laboratories Fellow* e *AT&T Fellow*. Em 1993, recebeu o prêmio Grace Murray Hopper da ACM "por seu trabalho pioneiro que levou às bases da linguagem de programação C++. Por meio dessas bases e de esforços contínuos do Dr. Stroustrup, C++ se tornou uma das linguagens de programação mais influentes na história da computação."

UM BREVE HISTÓRICO SOBRE VOCÊ E A COMPUTAÇÃO

No que você estava trabalhando, e quando, antes de se juntar ao Bell Labs no início dos anos 1980? No Bell Labs, estava pesquisando na área geral de sistemas distribuídos. Entrei lá em 1979. Antes disso, estava terminando meu doutorado nessa área na Universidade de Cambridge.

Você imediatamente começou a trabalhar no "C com Classes" (que mais tarde se tornou C++)? Trabalhei em alguns projetos relacionados à computação distribuída antes de iniciar o C com Classes e durante o desenvolvimento dele e de C++. Por exemplo, estava tentando encontrar uma maneira de distribuir o núcleo do UNIX entre vários computadores e ajudei diversos projetos a construírem simuladores.

Foi o interesse em matemática que levou você a essa profissão? Me inscrevi para um bacharelado em "matemática com ciência da computação" e meu mestrado é oficialmente em matemática. Eu – erroneamente – pensei que a computação fosse algum tipo de matemática aplicada. Fiz alguns anos de matemática e me considero um fraco, mas ainda isso é muito melhor do que não saber nada de matemática. No momento em que me inscrevi, nunca havia visto um computador. O que eu amo na computação é a programação, e não os campos mais matemáticos.

DISSECANDO UMA LINGUAGEM BEM-SUCEDIDA

Gostaria de começar de trás para frente, listando alguns itens que eu acredito terem o C++ onipresente, e ver sua reação. É "código aberto", não proprietário, e padronizado pela ANSI/ISO. O padrão ISO C++ é importante. Existem muitas implementações de C++ desenvolvidas e mantidas de forma independente. Sem um padrão para elas se adequarem e um processo de padronização para ajudar a coordenar a evolução de C++, surgiria um caos de dialetos.

Também é importante o fato de que existem implementações de código aberto e comerciais disponíveis. Além disso, para muitos usuários, é crucial que o padrão forneça uma medida de proteção contra a manipulação por parte dos fornecedores de implementações.

O processo de padronização da ISO é aberto e democrático. O comitê de C++ raramente se encontra com menos de 50 pessoas presentes e em geral mais de oito nações estão representadas em cada reunião. Não é apenas um fórum de vendedores.

C++ é ideal para a programação de sistemas (que, no momento do nascimento de C++, era o maior setor do mercado de desenvolvimento).

Sim, C++ é um forte concorrente para qualquer projeto de programação de sistemas. Ele também é eficaz para a programação de sistemas embarcados, atualmente o setor de maior crescimento. Outra área

de crescimento para C++ é a programação de alto desempenho numérica, de engenharia e científica.

Sua natureza orientada a objetos e a inclusão de classes e bibliotecas tornam a programação mais eficiente e transparente. C++ é uma linguagem de programação multiparadigmas. Ou seja, ela suporta diversos estilos de programação fundamentais (incluindo a programação orientada a objetos) e combinações desses estilos. Quando bem usadas, isso leva a bibliotecas mais limpas, flexíveis e eficientes do que poderiam ser fornecidas usando apenas um paradigma. Os contêineres e algoritmos da biblioteca padrão de C++, basicamente um *framework* de programação genérica, são um exemplo. Quando usadas com hierarquias de classes (orientadas a objetos), o resultado é uma combinação excelente de segurança de tipos, eficiência e flexibilidade.

Sua incubação no ambiente de desenvolvimento na AT&T. O AT&T Bell Labs forneceu um ambiente crucial para o desenvolvimento de C++. Os laboratórios eram uma fonte excepcionalmente rica de problemas desafiadores e um ambiente unicamente colaborativo para pesquisas práticas. C++ emergiu do mesmo laboratório de pesquisa de C e se beneficiou da mesma tradição intelectual, experiência e pessoas excepcionais. Por meio disso, a AT&T suportou a padronização de C++. Entretanto, C++ não teve uma campanha de marketing massiva, como muitas linguagens modernas. O Bell Labs simplesmente não trabalha dessa maneira.

Esqueci algo de sua lista principal de itens? Sem dúvida.

Agora, deixe-me parafrasear algumas das críticas a C++ e obter suas reações: C++ é uma linguagem imensa/selvagem. O problema "*hello world*" é 10 vezes maior em C++ que em C. C++ certamente não é uma linguagem pequena, mas poucas linguagens modernas o são. Se uma linguagem é pequena, você tende a precisar de bibliotecas gigantescas para fazer o trabalho e normalmente depende de convenções e extensões. Prefiro ter as partes-chave da complexidade inevitável na linguagem, em que elas podem ser vistas, ensinadas e efetivamente padronizadas, em vez de ocultas em algum lugar de um sistema. Para a maioria das necessidades, não considero C++ "selvagem". O programa "hello world" em C++ não é maior do que seu equivalente em C na minha máquina, e não deve ser na sua também. Na verdade, o código objeto para a versão de C++ do programa "hello world" é menor do que a versão de C na minha máquina. Não existe uma razão linguística pela qual uma versão deveria ser maior que a outra. Tudo é uma questão de como o implementador organizou as bibliotecas. Se uma versão é significativamente maior do que a outra, relate o problema para o implementador da versão maior.

Os críticos dizem que é mais difícil programar em C++ (comparado com C). Você mesmo comentou algo sobre atirar em seu próprio pé em relação a C *versus* C++. Sim, eu realmente disse algo como "é facil atirar em seu próprio pé com C; com C++ é mais difícil, mas, quando acontece, C++ explode sua perna inteira." No entanto, o que eu disse sobre C++ é, em um grau variável, verdadeiro para todas as linguagens poderosas. À medida que você protege as pessoas de perigos simples, elas começam enfrentar problemas novos e menos óbvios. Alguém que evita os problemas simples pode simplesmente estar se direcionando para um problema não tão simples. Uma das dificuldades de ambientes muito protetores e do excesso de suporte é que os problemas difíceis podem ser descobertos tarde demais. Além disso, um problema raro é mais difícil de encontrar do que um frequente porque você não suspeita dele.

C++ é adequado para sistemas embarcados de hoje, mas não para os sistemas de software para Internet atuais. C++ é adequado para sistemas embarcados hoje. Ele também é adequado a – e amplamente usado em – "software para a Internet" hoje. Por exemplo, dê uma olhada na minha página chamada "aplicações C++". Você notará que alguns dos maiores provedores de serviços Web, como Amazon, Adobe, Google, Quicken e Microsoft, dependem criticamente de C++. Jogos é uma área relacionada na qual se usa muito C++.

Esqueci alguma pergunta que você ouve com frequência? Claro.

Agora, todos os detalhes da implementação podem ser fornecidos no pacote de corpo, como em:

```
package body Linked_List_Type is
  type Node_Record is
    record
      Info : Integer;
      Link : Node;
    end record;
    ...
end Linked_List_Type;
```

Existem diversos problemas com essa versão de certa forma mais clara. Primeiro, existem as dificuldades inerentes de lidar com ponteiros. Segundo, comparações entre dois objetos do novo tipo de dados abstrato serão entre ponteiros, o que não produz os resultados esperados – porque os ponteiros são comparados, em vez de os objetos para os quais eles apontam. Outro problema de definir um tipo de dados abstrato como um ponteiro é a inabilidade do tipo de controlar a alocação e a liberação de objetos do tipo. Por exemplo, um cliente pode criar um ponteiro para um objeto (com uma declaração de variável) e usá-lo sem criar um objeto.

Uma alternativa aos tipos privados é uma forma mais restrita: os **tipos privados limitados**. Tipos privados limitados não baseados em ponteiros são descritos na seção privada de um pacote de especificação, assim como os tipos privados não baseados em ponteiros. A única diferença sintática é que os tipos privados limitados são declarados como `limited private` na parte visível da especificação do pacote. A diferença semântica é que objetos de um tipo que é declarado como privado limitado não têm operações pré-definidas. Tal tipo é útil quando as operações pré-definidas de atribuição e de comparação não são significativas nem úteis. Por exemplo, atribuições e comparações raramente são usadas para pilhas. Se a atribuição e as comparações forem necessárias, mas as versões pré-definidas não forem úteis, essas operações devem ser fornecidas pelo pacote de especificação. Essa é uma maneira de evitar os problemas de comparação quando o tipo de dados abstrato é um ponteiro. A operação de atribuição deve estar na forma de um procedimento normal, enquanto os operadores de igualdade e de diferença podem ser fornecidos pela sobrecarga desses operadores para o novo tipo.

11.4.1.3 Um exemplo

A seguir, temos o pacote de especificação para um tipo de dados abstrato pilha:

```
package Stack_Pack is
-- As entidades visíveis, ou interface pública
  type Stack_Type is limited private;
```

```
  Max_Size : constant := 100;
  function Empty(Stk : in Stack_Type) return Boolean;
  procedure Push(Stk : in out Stack_Type;
                 Element : in Integer);
  procedure Pop(Stk : in out Stack_Type);
  function Top(Stk : in Stack_Type) return Integer;
-- A parte que é oculta dos clientes
  private
    type List_Type is array (1..Max_Size) of Integer;
    type Stack_Type is
      record
      List : List_Type;
      Topsub : Integer range 0..Max_Size := 0;
      end record;
  end Stack_Pack;
```

Note que nenhuma operação de criação ou destruição é incluída, porque elas não são necessárias.

O pacote de corpo para `Stack_Pack` é:

```
with Ada.Text_IO; use Ada.Text_IO;
package body Stack_Pack is
  function Empty(Stk: in Stack_Type) return Boolean is
    begin
    return Stk.Topsub = 0;
    end Empty;

  procedure Push(Stk : in out Stack_Type;
      Element : in Integer) is
    begin
    if Stk.Topsub >= Max_Size then
      Put_Line("ERROR - Stack overflow");
    else
      Stk.Topsub := Stk.Topsub + 1;
      Stk.List(Topsub) := Element;
    end if;
  end Push;

  procedure Pop(Stk : in out Stack_Type) is
    begin
   if Stk.Topsub = 0
      then Put_Line("ERROR - Stack underflow");
      else Stk.Topsub := Stk.Topsub - 1;
    end if;
    end Pop;

  function Top(Stk : in Stack_Type) return Integer is
    begin
```

```
    if Stk.Topsub = 0
      then Put_Line("ERROR - Stack is empty");
      else return Stk.List(Stk.Topsub);
    end if;
    end Top;
  end Stack_Pack;
```

A primeira linha do código desse pacote de corpo contém duas cláusulas: **with** e **use**. A cláusula **with** torna os nomes definidos em pacotes externos visíveis; nesse caso `Ada.Text_IO`, que fornece funções para entrada e saída de texto. A cláusula **use** elimina a necessidade de qualificação explícita de referências a entidades a partir do pacote nomeado. As questões de acesso a encapsulamentos externos e qualificações de nomes serão discutidas na Seção 11.6.

O pacote de corpo deve ter as definições de subprogramas com cabeçalhos que casem com os de subprogramas no pacote de especificação associado. O pacote de especificação promete que esses subprogramas serão definidos no pacote de corpo associado.

O seguinte procedimento, `Use_Stacks`, é um cliente do pacote `Stack_Pack`. Ele ilustra como o pacote poderia ser usado.

```
with Stack_Pack;
use Stack_Pack;
procedure Use_Stacks is
  Topone : Integer;
  Stack : Stack_Type;   -- Cria um objeto Stack_Type
  begin
  Push(Stack, 42);
  Push(Stack, 17);
  Topone := Top(Stack);
  Pop(Stack);
  ...
  end Use_Stacks;
```

Uma pilha é um exemplo tolo para a maioria das linguagens contemporâneas, porque o suporte para pilhas é incluído em suas bibliotecas de classe padrão. Entretanto, pilhas fornecem um exemplo simples que podemos usar para permitir comparações das linguagens discutidas nesta seção.

11.4.2 Tipos de dados abstratos em C++

C++ foi criado com a adição de recursos à linguagem C. As primeiras adições importantes foram as que suportam a programação orientada a objetos. Como um dos componentes primários da programação orientada a objetos são os tipos de dados abstratos, C++ obviamente deve suportá-los.

Enquanto Ada fornece um encapsulamento que pode ser usado para simular tipos de dados abstratos, C++ fornece duas construções bastante simi-

lares uma a outra, a classe (`class`) e a estrutura (`struct`), que suportam mais diretamente os tipos de dados abstratos. Como essas estruturas são mais comumente usadas quando apenas dados são incluídos, não as discutimos aqui.

Classes em C++ são tipos; conforme mencionado, os pacotes em Ada são encapsulamentos mais generalizados que podem definir qualquer número de tipos. Uma unidade de programa que ganha visibilidade para um pacote Ada pode acessar qualquer uma de suas entidades públicas diretamente por seus nomes. Uma unidade de programa C++ que declara uma instância de uma classe também pode acessar qualquer uma das entidades públicas nessa classe, mas apenas por meio de uma instância da classe. Essa é uma maneira mais limpa e direta de fornecer tipos de dados abstratos.

11.4.2.1 Encapsulamento

Os dados definidos em uma classe C++ são chamados de **membros de dados**; as funções (métodos) definidas em uma classe são chamadas de **funções membro**. Membros de dados e funções membro aparecem em duas categorias: classes e instâncias. Neste capítulo, apenas os membros de instância de uma classe são discutidos. Todas as instâncias de uma classe compartilham um conjunto único de funções membro, mas cada instância tem seu próprio conjunto dos membros de dados da classe. Instâncias de classe podem ser estáticas, dinâmicas da pilha ou dinâmicas do monte. Se estáticas ou dinâmicas da pilha, são referenciadas diretamente com variáveis de valores. Se dinâmicas do monte, são referenciadas por ponteiros. Instâncias de classe dinâmicas da pilha são sempre criadas pela elaboração de uma declaração de objeto. Além disso, seu tempo de vida termina quando o final do escopo de sua declaração é alcançado. Objetos de classe dinâmicos do monte são criados com o operador `new` e destruídos com `delete`. Tanto classes da pilha quanto dinâmicas do monte podem ter ponteiros como membros de dados que referenciam dados dinâmicos do monte – então, mesmo que uma instância seja dinâmica da pilha, ela pode incluir membros de dados que referenciam dados dinâmicos do monte.

Uma função membro de uma classe pode ser definida de duas maneiras: a definição completa pode aparecer na classe ou apenas seu cabeçalho. Quando tanto o cabeçalho quanto o corpo de uma função membro aparecem na definição da classe, a função é implicitamente internalizada. Isso significa que seu código é colocado no código do chamador, em vez de requerer o processo usual de chamada e retorno. Se apenas o cabeçalho de uma função membro aparece na definição da classe, sua definição completa aparece fora da classe e é compilada separadamente. A razão para permitir que as funções membro sejam internalizadas era economizar tempo em aplicações de tempo real, nas quais a eficiência em tempo de execução é de fundamental importância. A desvantagem de internalizar funções membro é que isso polui a interface da definição da classe, resultando em uma redução na legibilidade.

Colocar as definições das funções membro fora da definição da classe separa a especificação da implementação, um objetivo comum da programação moderna.

11.4.2.2 Ocultamento de informação

Uma classe C++ pode conter tanto entidades ocultas quanto visíveis (ocultas dos clientes ou visíveis para os clientes da classe). Entidades ocultas são colocadas em uma cláusula **`private`**, e entidades visíveis, ou públicas, aparecem em **`public`**. Logo, a cláusula **`public`** descreve a interface para objetos da classe. Existe ainda uma terceira categoria de visibilidade, **`protected`**, discutida no contexto de herança no Capítulo 12.

11.4.2.3 Construtores e destrutores

C++ permite que o usuário inclua funções chamadas de **construtores** em definições de classe, usadas para inicializar os membros de dados de novos objetos sendo criados. Um construtor pode também alocar os dados dinâmicos do monte referenciados pelos membros ponteiros do novo objeto. Construtores são implicitamente chamados quando um objeto do tipo da classe é criado. Um construtor tem o mesmo nome da classe cujos objetos ele inicializa. Os construtores podem ser sobrecarregados, mas é claro que cada construtor de uma classe deve ter um perfil de parâmetros único.

Uma classe C++ pode também incluir uma função chamada de um **destrutor**, implicitamente chamado quando o tempo de vida de uma instância da classe termina. Conforme mencionado, instâncias de classe dinâmicas da pilha podem conter membros ponteiros que referenciam dados dinâmicos do monte. A função destrutora para essa instância pode incluir um operador **`delete`** nos membros ponteiros para liberar o espaço no monte que eles referenciam. Os destrutores são usados como uma ajuda à depuração, simplesmente mostrando ou imprimindo os valores de algum ou de todos os membros de dados do objeto antes de esses membros serem liberados. O nome de um destrutor é o nome da classe, precedido por til (~).

Nem construtores nem destrutores têm tipos de retorno, e nenhum deles usa sentenças **`return`**. Os dois podem ser chamados explicitamente.

11.4.2.4 Um exemplo

Nosso exemplo de um tipo de dados abstrato em C++ é, mais uma vez, uma pilha:

```
#include <iostream.h>
class Stack {
  private:  //** Esses membros são visíveis apenas para outros
            //** membros e amigos (veja Seção 11.6.4)
    int *stackPtr;
```

```
    int maxLen;
    int topPtr;
  public:   //** Esses membros são visíveis para os clientes
    Stack() {   //** Um construtor
      stackPtr = new int [100];
      maxLen = 99;
      topPtr = -1;
    }
    ~Stack() {delete [] stackPtr;};  //** Um destrutor
    void push(int number) {
      if (topPtr == maxLen)
        cerr << "Error in push--stack is full\n";
      else stackPtr[++topPtr] = number;
    }
    void pop() {
      if (topPtr == -1)
        cerr << "Error in pop--stack is empty\n";
      else topPtr--;
    }
    int top() {return (stackPtr[topPtr]);}
    int empty() {return (topPtr == -1);}
}
```

Discutimos poucos aspectos dessa definição de classe, porque não é necessário entender todos os detalhes do código. Objetos da classe `Stack` são dinâmicos da pilha, mas incluem um ponteiro que referencia dados dinâmicos do monte. A classe `Stack` tem três membros de dados – `stackPtr`, `maxLen` e `topPtr` – todos privados. O membro `stackPtr` é usado para referenciar os dados dinâmicos do monte, a matriz que implementa a pilha. A classe tem também quatro funções membro públicas – `push`, `pop`, `top` e `empty` – assim como um construtor e um destrutor. Todas as definições de funções membro são incluídas nessa classe, apesar de elas poderem ter sido definidas externamente. Como os corpos das funções são incluídos, são todos implicitamente internalizados. O construtor usa o operador **`new`** para alocar um vetor com 100 elementos inteiros do monte (**`int`**). Ele também inicializa `maxLen` e `topPtr`. O propósito da função destrutora é liberar o armazenamento para o vetor usado para implementar a pilha quando o tempo de vida de um objeto `Stack` termina. Esse vetor foi alocado pelo construtor.

Um programa de exemplo que usa o tipo de dados abstrato `Stack` é

```
void main() {
  int topOne;
  Stack stk;  //** Cria uma instância da classe Stack
  stk.push(42);
  stk.push(17);
```

```
  topOne = stk.top();
  stk.pop();
  ...
}
```

A seguir, temos uma definição da classe `Stack` com apenas os protótipos das funções membro. Esse código é armazenado em um arquivo de cabeçalho com a extensão do nome do arquivo `.h`. As definições das funções membro seguem a definição da classe. Elas usam o operador de resolução de escopo, `::`, para indicar a classe à qual pertencem. Essas definições são armazenadas em um arquivo de código com a extensão de nome `.cpp`.

```
// Stack.h - o arquivo de cabeçalho para a classe Stack
#include <iostream.h>
class Stack {
  private:  //** Esses membros são visíveis apenas para outros
            //** membros e amigos (veja Seção 11.6.4)
    int *stackPtr;
    int maxLen;
    int topPtr;
  public:   //** Esses membros são visíveis para os clientes
    Stack();  //** Um construtor
    ~Stack();  //** Um destrutor
    void push(int);
    void pop();
    int top();
    int empty();
}

// Stack.cpp - o arquivo de implementação para a classe Stack
#include <iostream.h>
#include "Stack.h"
using std::cout;
Stack::Stack() {  //** Um construtor
  stackPtr = new int [100];
  maxLen = 99;
  topPtr = -1;
}

Stack::~Stack() {delete [] stackPtr;};  //** Um destrutor

void Stack::push(int number) {
  if (topPtr == maxLen)
    cerr << "Error in push--stack is full\n";
  else stackPtr[++topPtr] = number;
}

void Stack::pop() {
  if (topPtr == -1)
```

```
    cerr << "Error in pop--stack is empty\n";
  else topPtr--;
}

int Stack::top() {return (stackPtr[topPtr]);}

int Stack::empty() {return (topPtr == -1);}
```

11.4.2.5 Avaliação

O suporte de C++ para tipos de dados abstratos, por meio de sua construção de classes, é similar ao poder de expressão do suporte de Ada, por seus pacotes. Ambos fornecem mecanismos efetivos para o encapsulamento e o ocultamento de informação de tipos de dados abstratos. A diferença primária é que classes são tipos, enquanto os pacotes de Ada são encapsulamentos mais gerais. Além disso, a construção de classe foi projetada para mais do que abstração de dados, como discutido no Capítulo 12.

11.4.3 Tipos de dados abstratos em Java

O suporte de Java para tipos de dados abstratos é similar ao de C++. Existem, no entanto, algumas diferenças importantes. Todos os tipos de dados definidos pelo usuário em Java são classes (Java não inclui *structs*), e todos os objetos são alocados do monte e acessados por meio de variáveis de referência. Outra diferença é que os métodos em Java devem ser definidos completamente em uma classe. Um corpo de método deve aparecer com seu cabeçalho de método correspondente. Logo, um tipo de dados abstrato é declarado e definido em uma única unidade sintática. Um compilador Java pode internalizar qualquer método não sobrescrito. As definições são ocultas dos clientes tornando-as privadas.

Em vez de ter cláusulas privadas e públicas em suas definições de classe, em Java os modificadores de acesso podem ser anexados às definições de métodos e de variáveis.

A seguir, temos uma definição de classe em Java para nosso exemplo da pilha:

```
import java.io.*;
class StackClass {
  private int [] stackRef;
  private int maxLen,
              topIndex;
  public StackClass() {  // Um construtor
    stackRef = new int [100];
    maxLen = 99;
    topIndex = -1;
  }
  public void push(int number) {
```

```
    if (topIndex == maxLen)
     System.out.println("Error in push—stack is full");
    else stackRef[++topIndex] = number;
  }
  public void pop() {
    if (topIndex == -1)
      System.out.println("Error in pop—stack is empty");
    else --topIndex;
  }
  public int top() {return (stackRef[topIndex]);}
  public boolean empty() {return (topIndex == -1);}
}
```

Uma classe de exemplo que usa `StackClass` é mostrada a seguir:

```
public class TstStack {
  public static void main(String[] args) {
    StackClass myStack = new StackClass();
    myStack.push(42);
    myStack.push(29);
    System.out.println("29 is: " + myStack.top());
    myStack.pop();
    System.out.println("42 is: " + myStack.top());
    myStack.pop();
    myStack.pop();  // Produz uma mensagem de erro
  }
}
```

Uma diferença óbvia é a falta de um destrutor na versão Java, desnecessário por causa da coleção de lixo implícita em Java.

Nosso exemplo não ilustra uma das diferenças importantes entre o suporte para tipos de dados abstratos de C++ e de Java. Entretanto, como será discutido na Seção 11.6, existem mais diferenças entre os encapsulamentos de C++ e os de Java. Além disso, quando consideramos outros aspectos da programação orientada a objetos, como é feito no Capítulo 12, mais diferenças entre as classes Java e as de C++ serão discutidas.

11.4.4 Tipos de dados abstratos em C#

Lembre que C# é baseado tanto em C++ quanto em Java e inclui algumas construções novas.

C# usa os modificadores de acesso **`private`**, **`public`** e **`protected`**[1] exatamente como eles são usados em Java. Entretanto, inclui também dois modificadores que Java não tem, **`internal`** e **`protected internal`**. O modificador **`internal`** é descrito na Seção 11.6, onde encapsulamentos generalizados são discutidos.

Assim como em Java, todas as instâncias de classe em C# são dinâmicas do monte. Construtores padrão, que fornecem valores iniciais para dados

[1] O modificador de acesso **`protected`** é discutido no Capítulo 12.

de instância, são pré-definidos para todas as classes. Esses construtores fornecem valores iniciais típicos, como 0 para tipos **int** e **false** para tipos booleanos (**boolean**). Um usuário pode fabricar um construtor para qualquer classe que definir. Esse construtor pode atribuir valores iniciais a alguns ou todos os dados de instâncias da classe. Qualquer variável de instância não inicializada em um construtor definido pelo usuário recebe um valor do construtor padrão.

Como C# usa coleção de lixo para a maioria de seus objetos do monte, os destrutores são raramente usados.

Apesar de os princípios de tipos de dados abstratos ditarem que membros de dados de objetos sejam ocultos dos clientes, surgem muitas situações nas quais eles devem acessar esses membros. A solução comum é fornecer métodos de acesso, leitores e escritores (*getters* e *setters*), que permitem aos clientes acessar indiretamente os dados chamados ocultos – uma solução melhor do que simplesmente tornar os dados públicos, fornecendo acesso direto. As razões pelas quais os métodos de acesso são melhores:

1. Acesso apenas de leitura pode ser fornecido, tendo um método de leitura, mas nenhum método de escrita correspondente.
2. Restrições podem ser incluídas nos escritores. Por exemplo, se o valor de dado deve ser restrito de forma a estar em uma determinada faixa, o escritor pode garantir isso.
3. A implementação real do membro de dados pode ser modificada sem afetar os clientes se os leitores e escritores forem a única forma de acesso.

C# fornece propriedades, herdadas do Delphi, como uma maneira de implementar leitores e escritores sem requerer chamadas a métodos explícitas. Propriedades fornecem acesso implícito a dados de instância privados específicos. Por exemplo, considere a seguinte classe simples e o código cliente:

```
public class Weather {
  public int DegreeDays {  //** DegreeDays é uma propriedade
    get {
      return degreeDays;
    }
    set {
      if(value < 0 || value > 30)
        Console.WriteLine(
             "Value is out of range: {0}", value);
      else
        degreeDays = value;
    }
  }
  private int degreeDays;
  ...
  }
...
Weather w = new Weather();
```

```
int degreeDaysToday, oldDegreeDays;
...
w.DegreeDays = degreeDaysToday;
...
oldDegreeDays = w.DegreeDays;
```

Na classe `Weather`, é definida a propriedade `DegreeDays`, que fornece um método de leitura escrita e um método de escrita para acesso ao membro de dados privado `degreeDays`. No código cliente logo a seguir da definição da classe, `degreeDays` é tratado como uma variável membro pública, apesar de o acesso a ela estar disponível apenas pela propriedade. Note o uso da variável implícita **`value`** no método de escrita. Esse é o mecanismo pelo qual o novo valor da propriedade é referenciado.

Conforme mencionado na Seção 11.4.2, C++ inclui tanto classes quanto estruturas (*structs*), construções praticamente idênticas – a única diferença é que o modificador de acesso padrão para classes é **`private`**, enquanto para estruturas é **`public`**. C# também tem estruturas, mas elas são bem diferentes das de C++. Em C#, estruturas são, em certo sentido, classes leves. Elas podem ter construtores, propriedades, métodos e atributos de dados e implementar interfaces, mas não suportam herança. Outra diferença fundamental entre estruturas e classes em C# é que estruturas são tipos de valores, em oposição a tipos de referência. Elas são alocadas na pilha de tempo de execução, em vez de no monte. Se forem passadas como parâmetros, elas são passadas por valor. Todos os tipos de valores em C#, incluindo seus tipos primitivos, são estruturas. Apesar de isso parecer estranho, objetos de estruturas são criados com o mesmo operador **`new`** usado para criar objetos de classe.

Estruturas são usadas em C# primariamente para implementar tipos relativamente simples e pequenos que nunca precisarão ser tipos base para herança. Elas também são usadas quando é conveniente para os objetos de um tipo serem alocados na pilha em vez de no monte.

11.4.5 Tipos de dados abstratos em Ruby

Ruby fornece suporte completo para tipos de dados abstratos por meio de suas classes. Em termos de capacidades, as classes de Ruby são similares às de C++ e de Java.

Em Ruby, uma classe é definida em uma sentença composta aberta com a palavra reservada **`class`**. Variáveis locais têm nomes com a forma de nomes de variáveis em outras linguagens de programação. Os nomes das variáveis de instância começam com o sinal arroba (@). Classes podem ter variáveis de classes (instanciadas com a classe, em vez de com a instância), cujos nomes devem começar com dois sinais arroba (@@). Métodos de instância têm a mesma sintaxe que as funções em Ruby: começam com a palavra reservada **`def`** e são fechados com **`end`**. Métodos de classe são distinguidos de métodos de instância por meio da inserção do nome da classe no início do nome do método

com um ponto como separador. Por exemplo, em uma classe chamada `Stack`, um nome de método de classe teria `Stack.` antes do nome. Construtores em Ruby são chamados `initialize`.

Se uma classe precisa de mais de um construtor, todos devem ter nomes únicos. O construtor `initialize` é implicitamente chamado quando o método de classe **`new`** é chamado. Outros construtores são definidos como outros métodos, mas apenas chamam **`new`** com os parâmetros a serem enviados para `initialize`, que será implicitamente chamado. Por exemplo, considere o seguinte construtor em uma classe chamada `Circle`:

```
def initialize(red, green, blue, radius)
  @red, @green, @blue, @radius = red, green, blue, radius
end
```

Um segundo construtor que cria um objeto da classe `Circle` com o raio de 5,0 é mostrado a seguir:

```
def Circle.r5circle(red, green, blue):
  new(red, green, blue, 5.0)
end
```

Métodos de uma classe podem ser marcados como privados ou públicos, sendo públicos o padrão[2]. Acessos privados e públicos em Ruby têm o mesmo significado que em Java. Todos os membros de dados devem ser privados para suportar ocultamento de informação.

Classes em Ruby são dinâmicas no sentido que seus membros podem ser adicionados em qualquer momento, simplesmente incluindo definições de classe adicionais que especificam novos membros. Métodos também podem ser removidos de uma classe, ao se fornecer outra definição de classe na qual o método a ser removido é enviado ao `remove_method` como um parâmetro. As classes dinâmicas de Ruby são outro exemplo de um projetista de linguagem trocando a legibilidade (e a confiabilidade) por flexibilidade. Para determinar a definição atual de uma classe, é necessário encontrar suas definições no programa e considerar todas elas.

A seguir, está o exemplo da pilha escrito em Ruby:

```
# Stack.rb - define e testa uma pilha de tamanho máximo
#            igual a 100, implementada como um vetor

class StackClass

# Constructor

  def initialize
    @stackRef = Array.new
```

[2] Ruby também suporta o modo de acesso protegido, conforme discutido no Capítulo 12.

```
    @maxLen = 100
     @topIndex = -1
  end

# push method

  def push(number)
    if @topIndex == @maxLen
      puts "Error in push - stack is full"
    else
      @topIndex = @topIndex + 1
      @stackRef[@topIndex] = number
    end
  end

# pop method

  def pop
    if @topIndex == -1
      puts "Error in pop - stack is empty"
    else
      @topIndex = @topIndex - 1
    end
  end

# top method

  def top
    @stackRef[@topIndex]
  end
# empty method
  def empty
    @topIndex == -1
  end
end  # of Stack class

# Test code for StackClass

myStack = StackClass.new
myStack.push(42)
myStack.push(29)
puts "Top element is (should be 29): #{myStack.top}"
myStack.pop
puts "Top element is (should be 42): #{mystack.top}"
myStack.pop
# O pop a seguir deve produzir uma
#  mensagem de erro - a pilha está vazia

myStack.pop
```

Lembre que a notação `#{variável}` converte o valor da variável para uma cadeia, então inserida na cadeia na qual aparece. Essa classe define uma estrutura de pilha que pode armazenar objetos de qualquer tipo. Em Ruby, tudo são objetos e as matrizes são, na verdade, matrizes de referências a objetos, o que torna essa pilha mais flexível do que os exemplos similares em Ada, C++ e Java. Além disso, simplesmente passando o tamanho máximo desejado para o construtor, os objetos dessa classe podem ter qualquer tamanho máximo. Visto que as matrizes em Ruby têm tamanho dinâmico, a classe poderia ser modificada para implementar objetos da pilha que não são restritos a tamanho algum, exceto aquele imposto pela capacidade de memória da máquina.

11.5 TIPOS DE DADOS ABSTRATOS PARAMETRIZADOS

Em geral, é conveniente ser capaz de parametrizar tipos de dados abstratos. Por exemplo, devemos ser capazes de projetar um tipo de dados abstrato pilha que possa armazenar quaisquer tipos escalares em vez de ter de escrever uma abstração de pilha separada para cada tipo escalar diferente. Note que essa é apenas uma questão para linguagens estaticamente tipadas. Em uma linguagem dinamicamente tipada, como Ruby, qualquer pilha pode implicitamente armazenar elementos de qualquer tipo. Nas quatro seções seguintes, são discutidas as capacidades de Ada, C++, Java 5.0 e C# 2005 para construir tipos de dados abstratos parametrizados.

11.5.1 Ada

Procedimentos genéricos em Ada foram discutidos e ilustrados no Capítulo 9. Pacotes também podem ser genéricos, então podemos construir tipos de dados abstratos genéricos, ou parametrizados.

O exemplo do tipo de dados abstrato pilha em Ada mostrado na Seção 11.4.1 sofre de duas restrições: (1) as pilhas desse tipo podem armazenar apenas elementos do tipo inteiro e (2) as pilhas podem ter apenas 100 elementos no máximo. Ambas as restrições podem ser eliminadas usando um pacote genérico, que pode ser instanciado para outros tipos de elementos e de qualquer tamanho desejado. (Essa é uma instanciação genérica, bem diferente da instanciação de uma classe para criar um objeto). A seguinte especificação de pacote descreve a interface de um tipo de dados abstrato pilha genérico com esses recursos:

```
generic
  Max_Size : Positive;  -- Um parâmetro genérico para
                        -- o tamanho da pilha
  type Element_Type is private;  -- Um parâmetro genérico
                                 -- para o tipo de elemento
```

```
package Generic_Stack is
-- As entidades visíveis, ou interface pública
  type Stack_Type is limited private;
  function Empty(Stk : in Stack_Type) return Boolean;
  procedure Push(Stk : in out Stack_Type;
                 Element : in Element_Type);
  procedure Pop(Stk : in out Stack_Type);
  function Top(Stk : in Stack_Type) return Element_Type;
-- A parte oculta
private
  type List_Type is array (1..Max_Size) of Element_Type;
  type Stack_Type is
    record
    List : List_Type;
    Topsub : Integer range 0..Max_Size := 0;
    end record;
  end Generic_Stack;
```

O pacote de corpo para `Generic_Stack` é o mesmo que para `Stack_Pack` na seção anterior, exceto que o tipo do parâmetro formal `Element` em `Push` e `Top` é `Element_Type` em vez de `Integer`. A seguinte sentença instancia `Generic_Stack` para uma pilha de 100 elementos do tipo `Integer`:

```
package Integer_Stack is new Generic_Stack(100, Integer);
```

Alguém poderia também construir um tipo de dados abstrato para uma pilha de tamanho 500 para elementos `Float`, como em

```
package Float_Stack is new Generic_Stack(500, Float);
```

Essas instanciações constroem duas versões de código fonte diferentes de `Generic_Stack` em tempo de compilação.

11.5.2 C++

C++ também suporta tipos de dados abstratos parametrizados. Para tornar o exemplo da classe pilha em C++ da Seção 11.4.2 genérico no tamanho da pilha, apenas a função construtora precisa ser modificada, como no código:

```
Stack(int size) {
  stkPtr = new int [size];
  maxLen = size - 1;
  top = -1;
}
```

A declaração para um objeto pilha agora pode aparecer como a seguir:

```
Stack stk(150);
```

A definição de classe para `Stack` pode incluir ambos os construtores, de forma que os usuários possam usar a pilha de tamanho padrão ou especificar outro.

O tipo do elemento da pilha pode ser genérico tornando a classe uma classe *template*. Então, o tipo do elemento pode ser um parâmetro de *template*. A definição da classe *template* para um tipo pilha é:

```
#include <iostream.h>
template <class Type>  // Type é o parâmetro de template
class Stack {
  private:
    Type *stackPtr;
    int maxLen;
    int topPtr;
  public:
// Um construtor para pilhas de 100 elementos
    Stack() {
      stackPtr = new Type [100];
      maxLen = 99;
      topPtr = -1;
    }
// A constructor for a given number of elements
    Stack(int size) {
      stackPtr = new Type [size];
      maxLen = size - 1;
      topPtr = -1;
    }
    ~Stack() {delete stackPtr;};  // Um destrutor
    void push(Type number) {
      if (topPtr == maxLen)
        cout << "Error in push—stack is full\n";
      else stackPtr[++ topPtr] = number;
    }
    void pop() {
      if (topPtr == -1)
        cout << "Error in pop—stack is empty\n";
      else topPtr --;
    }
    Type top() {return (stackPtr[topPtr]);}
    int empty() {return (topPtr == -1);}
}
```

Como em Ada, classes *template* em C++ são instanciadas em tempo de compilação. A diferença é que em C++ as instanciações são implícitas: uma nova instanciação é criada sempre que é criado um objeto que requeira uma versão da classe *template* até então não existente.

11.5.3 Java 5.0

Java 5.0 suporta uma forma de tipos de dados abstratos parametrizados na qual os parâmetros genéricos devem ser classes. Lembre que tais parâmetros são brevemente discutidos no Capítulo 9.

Os tipos genéricos mais comuns são de coleção, como `LinkedList` e `ArrayList`, que estavam na biblioteca de classes Java antes de o suporte para tipos genéricos ser adicionado. Os tipos de coleção armazenam objetos da classe `Object`, então podem armazenar quaisquer objetos (mas não tipos primitivos). Logo, os tipos de coleção sempre foram capazes de armazenar múltiplos tipos (desde que fossem classes). Existem três problemas acerca disso. Primeiro, sempre que um objeto é removido da coleção, ele deve ser convertido para o tipo apropriado. Segundo, não existe verificação de erros quando elementos são adicionados à coleção. Terceiro, os tipos de coleção não podem armazenar tipos primitivos. Então, para armazenar valores `int` em um `ArrayList`, o valor primeiro deve ser colocado em um objeto da classe `Integer`. Por exemplo, considere o código:

```
//* Cria um objeto da classe ArrayList
ArrayList myArray = new ArrayList();
//* Cria um elemento
myArray.add(0, new Integer(47));
//* Obtém o primeiro elemento
Integer myInt = (Integer)myArray.get(0);
```

Em Java 5.0, as classes de coleção, cujas mais usadas são `List`, `ArrayList` e `Queue`, tornaram-se classes genéricas. Essas classes são instanciadas chamando-se **`new`** no construtor da classe e passando a ele o parâmetro genérico entre os sinais de menor e maior (`<>`). Por exemplo, a classe `ArrayList` pode ser instanciada para armazenar objetos `Integer` com a sentença:

```
ArrayList <Integer> myArray = new ArrayList <Integer>();
```

Essa nova classe resolve dois dos problemas com as coleções anteriores a Java 5.0. Apenas objetos da classe `Integer` podem ser colocados na coleção `myArray`. Além disso, não existe a necessidade de converter um objeto sendo removido da coleção. Entretanto, ainda não é possível instanciar uma coleção genérica que armazena valores primitivos.

Lembre que no Capítulo 9 descrevemos que Java 5.0 suporta classes coringa. Por exemplo, `Collection<?>` é uma classe coringa para todas as classes de coleção. Isso permite que métodos sejam escritos de forma a aceitar qualquer tipo de coleção como parâmetro. Como uma coleção pode ser genérica, a classe `Collection<?>` é, em certo sentido, um tipo genérico de uma classe genérica.

Algum cuidado deve ser tomado com objetos do tipo coringa. Por exemplo, como os componentes de um objeto particular desse tipo têm um tipo, outros objetos de outros tipos não podem ser adicionados à coleção. Por exemplo, considere

```
Collection<?> c = new ArrayList<String>();
```

Seria ilegal usar o método `add` para colocar algo nessa coleção a menos que seu tipo fosse `String`.

Os usuários podem definir classes genéricas em Java 5.0. Esse é um processo bastante direto e as classes resultantes se comportam exatamente como as classes genéricas pré-definidas.

11.5.4 C# 2005

Como no caso de Java, a primeira versão de C# definia classes de coleção que armazenavam objetos de qualquer classe. Elas eram `ArrayList`, `Stack` e `Queue`, que tinham os mesmos problemas que as classes de coleção de Java antes da versão 5.0.

As classes genéricas foram adicionadas em C# em sua versão 2005. As cinco coleções genéricas pré-definidas são `Array`, `List`, `Stack`, `Queue` e `Dictionary` (`Dictionary` implementa dispersões). Exatamente como em Java 5.0, essas classes eliminam os problemas de permitir tipos mistos em coleções e requerer conversões quando os objetos são removidos das coleções.

Como em Java 5.0, os usuários podem definir classes genéricas em C# 2005. Uma capacidade das coleções genéricas definidas pelo usuário de C# é que qualquer uma delas permite que seus elementos sejam indexados (acessados por meio de índices). Apesar de os índices normalmente serem inteiros, uma alternativa é usar cadeias como índices.

Uma capacidade que Java 5.0 fornece que C# 2005 não fornece são as classes coringa.

11.6 CONSTRUÇÕES DE ENCAPSULAMENTO

As primeiras cinco seções deste capítulo discutem tipos de dados abstratos, que são encapsulamentos mínimos[3]. Esta seção descreve os encapsulamentos de múltiplos tipos necessários para programas maiores.

11.6.1 Introdução

Quando o tamanho de um programa ultrapassa a fronteira de algumas poucas mil linhas de código, dois problemas práticos se tornam evidentes. Do ponto de vista do programador, ter tal programa aparecendo como uma única coleção e subprogramas ou definições de tipos de dados abstratos não impõe um

[3] No caso de Ada, o encapsulamento pacote pode ser usado para tipos individuais e também para múltiplos tipos.

nível adequado de organização no programa para mantê-lo intelectualmente gerenciável. O segundo problema prático para programas grandes é a recompilação. Para os relativamente pequenos, recompilar o programa inteiro após cada modificação não é caro. Para programas grandes, o custo da recompilação é significativo. Então, existe uma necessidade óbvia de encontrar maneiras para evitar a recompilação das partes de um programa não afetadas por uma mudança. A solução para esses problemas é organizar os programas em coleções de códigos e dados logicamente relacionados, cada uma podendo ser compilada sem a recompilação do resto do programa. Um **encapsulamento** é tal coleção.

Encapsulamentos são colocados em bibliotecas e disponibilizados para reúso em programas além daqueles para os quais eles foram escritos. As pessoas têm escrito programas com mais alguns milhares de linhas nos últimos 40 anos, então as técnicas para fornecer encapsulamentos têm evoluído.

11.6.2 Subprogramas aninhados

Em linguagens que permitem subprogramas aninhados, os programas podem ser organizados por definições de subprogramas aninhadas dentro de subprogramas maiores que os usam. Isso pode ser feito em Ada, Fortran 95, Python e Ruby. Como discutido no Capítulo 5, entretanto, esse método de organizar programas, que usa escopo estático, está longe de ser o ideal. Logo, mesmo em linguagens que permitem subprogramas aninhados, eles não são usados como uma construção de organização de encapsulamento primária.

11.6.3 Encapsulamento em C

C não fornece um suporte forte para tipos de dados abstratos, apesar de tanto tipos de dados abstratos quanto encapsulamentos de múltiplos tipos poderem ser simulados.

Em C, uma coleção de funções e definições de dados relacionadas pode ser colocada em um arquivo, que pode ser compilado independentemente. Esse arquivo, que age como uma biblioteca, tem uma implementação de suas entidades. A interface para esse arquivo, incluindo os dados, tipos e declarações de funções, é colocada em um separado chamado de **arquivo de cabeçalho**. Representações de tipo podem ser ocultas declarando-as no arquivo de cabeçalho como ponteiros para tipos `struct`. As definições completas para tais tipos `struct` precisam aparecer apenas no arquivo de implementação. Essa abordagem tem as mesmas desvantagens do uso de ponteiros como tipos de dados abstratos em pacotes Ada – ou seja, os problemas inerentes de ponteiros e a confusão em potencial com a atribuição e com as comparações de ponteiros.

O arquivo de cabeçalho, em forma de fonte, e a versão compilada do arquivo de implementação são repassados para os clientes. Quando tal biblioteca é usada, o arquivo de cabeçalho é incluído no código cliente usando uma especificação de pré-processador `#include`, de forma que as referências às funções e aos dados no código cliente possam ser verificadas em relação aos seus tipos. A especificação `#include` também documenta o fato de que o programa cliente depende do arquivo de implementação da biblioteca. Essa abordagem separa a especificação e a implementação de um encapsulamento.

Apesar de esses encapsulamentos funcionarem, eles criam algumas inseguranças. Por exemplo, um usuário poderia simplesmente recortar e colar as definições do arquivo de cabeçalho no programa cliente, em vez de usar `#include`. Isso funcionaria, porque `#include` copia o conteúdo de seu arquivo operando para o arquivo no qual ele aparece. Entretanto, existem dois problemas com essa abordagem. Primeiro, a documentação de dependência entre o programa cliente e a biblioteca (e seu arquivo de cabeçalho) é perdida. Segundo, o autor da biblioteca pode modificar os arquivos de cabeçalho e os de implementação, mas o cliente poderia tentar usar o novo arquivo de implementação (sem se dar conta de que ele mudou), mas com o de cabeçalho antigo, o usuário havia copiado em seu programa cliente. Por exemplo, uma variável `x` poderia ter sido definida como do tipo `int` no arquivo de cabeçalho antigo, que o código cliente ainda usa, apesar de o código de implementação ter sido recompilado com o novo arquivo de cabeçalho, que define `x` como `float`. Então, o código de implementação foi compilado com `x` sendo um `int`, mas o código cliente foi compilado com `x` sendo um `float`. O ligador não detecta esse erro.

Então, é responsabilidade do usuário garantir que tanto o arquivo de cabeçalho quanto o de implementação estão atualizados. Em geral, isso é feito com um utilitário `make`.

11.6.4 Encapsulamento em C++

C++ fornece dois tipos de encapsulamento – arquivos de cabeçalho e de implementação podem ser definidos como em C, ou cabeçalhos de classes e definições podem ser definidas. Em função da complexa interação dos *templates* C++ e da compilação separada, os arquivos de cabeçalho das bibliotecas de *template* de C++ geralmente incluem a definição completa de recursos, em vez de apenas declarações de dados e protocolos de subprogramas; isso ocorre em parte devido ao uso do ligador C para programas C++.

Quando classes que não contêm *templates* são usadas para encapsulamentos, o arquivo de cabeçalho de classe tem apenas os protótipos das funções membros, com as definições das funções fornecidas fora da classe em um arquivo de código, como no último exemplo na Seção 11.4.2.4. Isso separa claramente a interface da implementação.

Um problema de projeto de linguagem resultante de se ter classes, mas nenhuma construção generalizada de encapsulamento, é não ser sempre natural associar operações de objetos com objetos individuais. Por exemplo, suponha que tivéssemos um tipo de dados abstrato para matrizes e outro para vetores e precisássemos de uma operação de multiplicação entre um vetor e uma matriz. O código de multiplicação deve ter acesso aos membros de dados tanto da classe do vetor quanto da classe da matriz, mas nenhuma das classes é a casa natural para o código. Além disso, independentemente de qual for escolhido, o acesso aos membros do outro é um problema. Em C++, essas situações podem ser manipuladas permitindo que funções não membro sejam "amigas" (*friends*) de uma classe. Funções amigas têm acesso às entidades privadas da classe em que são declaradas amigas. Para a operação de multiplicação entre matriz e vetor, uma solução em C++ é definir a operação fora das classes da matriz e do vetor, mas defini-la como amiga de ambas. O seguinte código esqueleto ilustra esse cenário:

```
class Matrix;  //** Uma declaração de classe
class Vector {
  friend Vector multiply(const Matrix&, const Vector&);
  ...
};
class Matrix {  //** The class definition
  friend Vector multiply(const Matrix&, const Vector&);
  ...
};
//** A função que usa tanto objetos do tipo Matrix quanto do
tipo Vector
Vector multiply(const Matrix& m1, const Vector& v1) {
  ...
}
```

Além de funções, classes inteiras podem ser definidas como amigas de uma classe; então, todos os membros privados da classe são visíveis para todos os membros da classe amiga.

11.6.5 Pacotes em Ada

Pacotes de especificação em Ada podem incluir qualquer número de declarações de dados ou de subprogramas em suas seções pública e privada. Logo, podem incluir interfaces para qualquer número de tipos de dados abstratos, assim como para quaisquer outros recursos de programas. Então, o pacote é uma construção de encapsulamento de múltiplos tipos.

Pacotes Ada podem ser compilados separadamente. As duas partes do pacote, especificação e corpo, também podem ser compiladas separadamente se o pacote de especificação for compilado primeiro. Um programa inteiro que use qualquer número de pacotes externos pode ser compilado separadamente, desde que as especificações de todos os pacotes usados tenham sido

compiladas. Os corpos dos pacotes usados podem ser compilados após o programa cliente.

Considere a situação descrita na Seção 11.6.4 dos tipos vetor e matriz e a necessidade para métodos com acesso às partes privadas de ambos, manipulada por meio de funções amigas em C++. Em Ada, tanto o tipo matriz quanto o tipo vetor podem ser definidos em um único pacote Ada, o que diminui a necessidade de funções amigas.

11.6.6 Montagens em C#

C# inclui uma construção de encapsulamento maior do que a classe. Nesse caso, a construção é aquela usada pelas linguagens de programação .NET: a **montagem** (*assembly*), uma coleção de um ou mais arquivos que aparecem para os programas aplicativos como uma única biblioteca de ligação dinâmica ou um executável (EXE). Cada arquivo define um módulo, que pode ser desenvolvido separadamente. Uma **biblioteca de ligação dinâmica** (DLL) é uma coleção de classes e métodos individualmente ligados a um programa que está executando quando necessário durante a execução. Logo, apesar de um programa ter acesso a todos os recursos em uma DLL em particular, apenas as partes realmente usadas são carregadas e ligadas ao programa. As DLLs têm sido parte do ambiente de programação do Windows desde sua a primeira aparição do sistema operacional. Além do código objeto para seus recursos, uma montagem .NET inclui um manifesto, com definições de topo para cada classe que ela contém, definições de outros recursos da montagem, uma lista de todas as montagens referenciadas na montagem e um número de versão da montagem.

No mundo .NET, a montagem é a unidade básica de implantação de software. Montagens podem ser privadas – disponíveis para apenas uma aplicação – ou públicas – disponíveis para qualquer aplicação.

Conforme mencionado, C# tem um modificador de acesso, `internal`. Um membro interno (`internal`) de uma classe é visível para todas as classes na montagem em que ele aparece.

Como uma montagem é uma unidade executável autocontida, ela pode ter apenas um ponto de entrada.

11.7 NOMEANDO ENCAPSULAMENTOS

Temos considerado os encapsulamentos como contêineres sintáticos para recursos de software relacionados logicamente – em particular, tipos de dados abstratos. O propósito desses encapsulamentos é fornecer uma maneira de organizar programas em unidades lógicas para a compilação, permitindo partes de programas serem recompiladas após mudanças isoladas. Existe outro tipo de encapsulamento necessário para construir grandes programas: um encapsulamento de nomeação.

Um grande programa pode ser escrito por muitos desenvolvedores, trabalhando de maneira independente, talvez até em localizações geográficas diferentes. Isso requer que as unidades lógicas do programa sejam independentes, embora ainda assim seja possível trabalhar em conjunto. Isso também cria um problema de nomeação: como desenvolvedores trabalhando independentemente criam nomes para suas variáveis, métodos e classes sem acidentalmente usar nomes já em uso por outro programador em uma parte diferente do mesmo sistema de software?

As bibliotecas são a origem do mesmo tipo de problemas de nomeação. Nas últimas duas décadas, grandes sistemas de software se tornaram mais dependentes de bibliotecas de software de suporte. Praticamente todo software escrito em linguagens de programação contemporâneas requer o uso de bibliotecas padrão grandes e complexas e de bibliotecas específicas para a aplicação em questão. Esse uso disseminado de múltiplas bibliotecas tem necessitado de novos mecanismos para gerenciar nomes. Por exemplo, quando um desenvolvedor adiciona novos nomes a uma biblioteca existente ou cria uma nova biblioteca, ele não pode usar um novo nome que entre em conflito com um já definido em um programa aplicativo do cliente ou em alguma outra biblioteca. Sem alguma assistência linguística, isso é praticamente impossível, porque não existe uma maneira de o autor da biblioteca saber que nomes um programa cliente usa ou quais são definidos por outras bibliotecas que o programa cliente possa usar.

Encapsulamentos de nomeação definem escopos de nome que ajudam a evitar conflitos. Cada biblioteca pode criar seu próprio encapsulamento para prevenir que seus nomes entrem em conflito com aqueles definidos em outras bibliotecas ou em código cliente. Cada parte lógica de um sistema de software pode criar um encapsulamento com o mesmo propósito.

11.7.1 Espaços de nome em C++

C++ inclui uma especificação, **`namespace`**, que ajuda os programas a gerenciarem o problema de espaços de nome globais. Alguém pode colocar cada biblioteca em seu próprio espaço de nomes e qualificá-los no programa com o nome do espaço de nomes quando eles são usados fora desse espaço. Por exemplo, suponha que existe um arquivo de cabeçalho de um tipo de dados abstrato que implementa pilhas. Se existe a preocupação de que algum outro arquivo de biblioteca possa definir um nome usado no tipo de dados abstrato pilha, o arquivo que define a pilha pode ser colocado em seu próprio espaço de nomes. Isso é feito colocando todas as declarações para a pilha em um bloco de espaço de nomes, como a seguir:

```
namespace MyStack {
  // Declarações de pilha
}
```

O arquivo de implementação para o tipo de dados abstrato pode referenciar os nomes declarados no arquivo de cabeçalho com o operador de resolução de escopo, `::`, como em

```
MyStack::topPtr
```

O arquivo de implementação também pode aparecer em uma especificação de bloco de espaço de nomes idêntica àquela usada no arquivo de cabeçalho, tornando todos os nomes declarados no arquivo de cabeçalho diretamente visíveis. Isso é definitivamente mais simples, mas levemente menos legível, porque é menos óbvio onde um nome específico no arquivo de implementação é declarado.

O código cliente pode ganhar acesso aos nomes no espaço de nomes do arquivo de cabeçalho de uma biblioteca de três maneiras. Uma é qualificá-los a partir da biblioteca com o nome do espaço de nomes. Por exemplo, uma referência à variável `topPtr` poderia aparecer como

```
MyStack::topPtr
```

que é exatamente a maneira pela qual o código de implementação poderia referenciá-la.

As outras duas abordagens usam a diretiva **`using`**. Ela pode ser usada para qualificar nomes individuais de um espaço de nomes, como em

```
using MyStack::topPtr;
```

que torna `topPtr` visível, mas não quaisquer outros nomes do espaço de nomes `MyStack`.

A diretiva **`using`** pode também ser usada para qualificar todos os nomes de um espaço de nomes, como em:

```
using namespace MyStack;
```

Código que inclui essa diretiva pode acessar diretamente os nomes definidos no espaço de nomes, como em

```
p = topPtr;
```

Esteja ciente de que os espaços de nomes são um recurso complicado de C++, e que introduzimos apenas a parte mais simples da história aqui.

C# inclui espaços de nomes bastante parecidos com aqueles de C++.

11.7.2 Pacotes em Java

Java inclui uma construção de encapsulamento de nomeação: o pacote. Pacotes podem conter mais de uma definição de classe, e as classes em um pacote são amigas parciais umas das outras. *Parcial* aqui significa que as en-

tidades definidas em uma classe em um pacote públicas ou protegidas (veja o Capítulo 12) ou que não têm um especificador de acesso são visíveis para todas as outras classes no pacote. Um pacote pode ter apenas uma definição de classe pública*.

Entidades sem modificadores de acesso são ditas como tendo **escopo de pacote**, porque são visíveis ao longo do pacote. Java, dessa forma, tem menos necessidade para declarações amigas explícitas e não inclui as funções amigas ou classes amigas de C++.

Os recursos definidos em um arquivo são especificados como em um pacote em particular com uma declaração de pacote, como em

```
package myStack;
```

A declaração de pacote deve aparecer como a primeira linha do arquivo. Os recursos de cada arquivo que não incluem uma declaração de pacote são implicitamente colocados no mesmo pacote não nomeado.

Os clientes de um pacote podem referenciar os nomes definidos no pacote com nomes completamente qualificados. Por exemplo, se o pacote `myStack` define uma variável chamada `topPtr`, ela pode ser referenciada em um cliente de `myStack` como `myStack.topPtr`**. Como esse procedimento pode rapidamente se tornar desajeitado quando os pacotes são aninhados, Java fornece a declaração **`import`**, que permite referências mais curtas aos nomes definidos em um pacote. Por exemplo, suponha que o cliente inclua o seguinte:

```
import myStack.*;
```

Agora, a variável `topPtr`, assim como outros nomes definidos no pacote `myStack`, pode ser referenciada apenas por seus nomes. Para acessar apenas um nome do pacote, ele pode ser dado na declaração de importação, como em

```
import myStack.topPtr;
```

Note que o **`import`** de Java é apenas um mecanismo de abreviação. Nenhum outro recurso externo oculto é tornado disponível com **`import`**. Na verdade, em Java nada é implicitamente oculto se puder ser encontrado pelo compilador ou pelo carregador de classes (usando o nome do pacote e a variável de ambiente `CLASSPATH`).

---

* N. de R. T.: Na verdade, uma unidade de compilação pode ter apenas uma definição de classe pública. Um pacote pode conter diversas definições de classes públicas.

** N. de R. T.: Na verdade, variáveis não podem ser referenciadas dessa forma, pois em Java não é possível definir variáveis globais. Normalmente, nomes de classes dentro dos pacotes são nomeados assim.

O **import** de Java documenta as dependências do pacote no qual ele aparece com os pacotes nomeados no **import**. Essas dependências são menos óbvias quando **import** não é usado.

11.7.3 Pacotes em Ada

Pacotes em Ada, em geral usados para encapsular bibliotecas, são definidos em hierarquias, que correspondem às de arquivos nas quais eles são armazenados. Por exemplo, se `subPack` é um pacote definido como um filho do `pack`, o arquivo de código de `subPack` apareceria em um subdiretório do diretório que armazenou `pack`. As bibliotecas de classe padrão de Java também definidas em uma hierarquia de pacotes e são armazenadas em uma hierarquia de diretórios correspondente.

Como discutido na Seção 11.4.1, os pacotes também definem espaços de nomes. A visibilidade a um pacote a partir de uma unidade de programa é obtida com a cláusula **with**. Por exemplo,

```
with Ada.Text_IO;
```

torna os recursos e o espaço de nomes do pacote `Ada.Text_IO` disponível. Acessos aos nomes definidos no espaço de nomes de `Ada.Text_IO` devem ser qualificados. Por exemplo, o procedimento `Put` de `Ada.Text_IO` deve ser acessado como

```
Ada.Text_IO.Put
```

Para acessar os nomes em `Ada.Text_IO` sem qualificação, a cláusula **use** pode ser usada, como em

```
use Ada.Text_IO;
```

Com essa cláusula, o procedimento `Put` de `Ada.Text_IO` pode ser acessado simplesmente como `Put`. O **use** de Ada é similar ao **import** de Java.

11.7.4 Módulos em Ruby

As classes de Ruby servem como encapsulamentos de nomeação, como o fazem as classes de outras linguagens que suportam programação orientada a objetos. Ruby tem um encapsulamento de nomeação adicional, chamado de *módulo*, que normalmente define coleções de métodos e constantes. Então, os módulos são convenientes para encapsular bibliotecas de métodos e constantes, cujos nomes estão em um espaço de nomes separado, de forma que não existem conflitos de nomes com outros nomes em um programa que usa o módulo. Módulos são diferentes das classes uma vez que não podem ser instanciados ou estendidos por herança e nem definem variáveis.

Métodos definidos em um módulo incluem o nome do módulo em seus nomes. Por exemplo, considere o seguinte esqueleto de definição de um módulo:

```
module MyStuff
  PI = 3.114159265
  def MyStuff.mymethod1(p1)
  ...
  end
  def MyStuff.mymethod2(p2)
  ...
  end
end
```

Assumindo que o módulo `MyStuff` é armazenado em seu próprio arquivo, um programa que quer usar a constante e os métodos de `MyStuff` deve primeiro obter acesso ao módulo. Isso é feito com o método `require`, que recebe o nome do arquivo na forma de um literal do tipo cadeia como parâmetro. Então, as constantes e métodos do módulo podem ser acessados pelo nome do módulo. Considere o seguinte código que usa nosso módulo de exemplo, `MyStuff`, armazenado no arquivo chamado `MyStuffMod`:

```
require 'myStuffMod'
...
MyStuff.mymethod1(x)
...
```

Os módulos são discutidos mais detalhadamente no Capítulo 12.

RESUMO

O conceito de tipos de dados abstratos, e seu uso em projeto de programas, foi um marco no desenvolvimento da programação como uma disciplina de engenharia. Apesar de o conceito ser relativamente simples, seu uso não se tornou conveniente e seguro até que linguagens foram projetadas para suportá-lo.

Os dois recursos principais de tipos de dados abstratos são o empacotamento de objetos de dados com suas operações associadas e o ocultamento de informações. Uma linguagem pode suportar tipos de dados abstratos ou simulá-los com encapsulamentos mais gerais.

Ada fornece encapsulamentos chamados pacotes que podem ser usados para simular tipos de dados abstratos. Os pacotes normalmente têm duas partes: uma especificação, que apresenta a interface cliente, e um corpo, que fornece a implementação de um tipo de dados abstrato. Representações de tipos de dados podem aparecer no pacote de especificação, mas serem ocultas de clientes ao colocá-las na cláusula privada do pacote. O tipo abstrato propriamente dito é definido como privado na parte pública do pacote de especificação. Tipos privados têm operações pré-definidas para atribuição e comparação para igualdade e desigualdade.

A abstração de dados em C++ é fornecida pelas classes. As classes são tipos, e as instâncias podem ser dinâmicas da pilha ou dinâmicas do monte. Uma função membro (método) pode ter sua definição completa aparecendo na classe ou apenas o protocolo dado na classe e a definição colocada em outro arquivo, que pode ser compilado separadamente. Classes em C++ têm três cláusulas, cada uma pré-fixada com um modificador de acesso: `private`, `public` ou `protected`. Tanto construtores quanto destrutores podem ser dados em definições de classes. Objetos alocados do monte devem ser explicitamente liberados com `delete`.

Abstrações de dados em Java são similares às de C++, exceto que todos os objetos em Java são alocados do monte e acessados por meio de variáveis de referência. Além disso, todos os objetos são coletados quando viram lixo. Em vez de ter modificadores de acesso anexados às cláusulas, em Java os modificadores aparecem em declarações individuais (ou definições).

C# suporta tipos de dados abstratos tanto com classes quanto com estruturas. Suas estruturas são tipos de valores e não suportam herança. De modo contrário, as classes C# são similares àquelas de Java.

Ruby suporta tipos de dados abstratos com suas classes. As classes de Ruby diferem daquelas da maioria das outras linguagens no sentido de que são dinâmicas – membros podem ser adicionados, apagados ou modificados durante a execução.

Ada, C++, Java 5.0 e C# 2005 permitem que seus tipos de dados abstratos sejam parametrizados – Ada por meio de seus pacotes genéricos, C++ por suas classes *template*, e Java 5.0 e C# 2005 com suas classes de coleção.

Para suportar a construção de grandes programas, algumas linguagens contemporâneas incluem construções de encapsulamento de múltiplos tipos, que podem conter uma coleção de tipos relacionados logicamente. Um encapsulamento pode fornecer controle de acesso às suas entidades e um método de organizar programas facilitando a recompilação.

C++, C#, Java, Ada e Ruby fornecem encapsulamentos de nomeação. Para Ada e Java, eles são chamados de pacotes; para C++ e C#, são espaços de nomes; para Ruby, são módulos. Parcialmente por causa da disponibilidade dos pacotes, Java não tem funções ou classes amigas. Em Ada, os pacotes podem ser usados como encapsulamentos de nomeação.

QUESTÕES DE REVISÃO

1. Quais são os dois tipos de abstrações em linguagens de programação?
2. Defina *tipo de dados abstrato*.
3. Quais são as vantagens das duas partes da definição de *tipo de dados abstrato*.
4. Quais são os requisitos de projeto de linguagem para uma linguagem que suporte tipos de dados abstratos?
5. Quais são as questões de projeto de linguagem para tipos de dados abstratos?
6. Explique como o ocultamento de informações é fornecido por um pacote Ada.
7. Para quem a parte privada de um pacote de especificação em Ada é visível?
8. Qual é a diferença entre os tipos `private` e `limited private` em Ada?
9. O que é um pacote de especificação em Ada? E o que é um pacote de corpo?
10. Qual é o uso da cláusula `with` em Ada?

11. Qual é o uso da cláusula `use` em Ada?
12. Qual é a diferença fundamental entre uma classe C++ e um pacote em Ada?
13. De onde os objetos em C++ são alocados?
14. Em que diferentes locais a definição de uma função membro em C++ pode aparecer?
15. Qual é o propósito de um construtor em C++?
16. Quais são os tipos legais de retorno de um construtor?
17. Onde todos os métodos Java são definidos?
18. Como os objetos de classe em C++ são criados?
19. De onde os objetos de classe Java são alocados?
20. Por que Java não tem destrutores?
21. Onde as classes Java são alocadas?
22. O que é uma função amiga? O que é uma classe amiga?
23. Cite uma razão pela qual Java não tem funções amigas ou classes amigas.
24. Descreva as diferenças fundamentais entre structs e classes em C#.
25. Como um objeto struct é criado em C#?
26. Explique as três razões pelas quais os métodos de acesso para tipos privados são melhores do que tornar os tipos públicos.
27. Quais são as diferenças entre structs em C++ e structs em C#?
28. Porque Java não precisa de uma cláusula `use`, como em Ada?
29. Qual é o nome de todos os construtores em Ruby?
30. Qual é a diferença fundamental entre as classes de Ruby e aquelas de C++ e de Java?
31. Como as instâncias das classes genéricas de Ada são criadas?
32. Como as instâncias das classes *template* de C++ são criadas?
33. Descreva os dois problemas que aparecem na construção de grandes programas que levaram ao desenvolvimento de construções de encapsulamento.
34. Que problemas podem ocorrer ao usar C para definir tipos de dados abstratos?
35. O que é um espaço de nomes em C++ e qual é o seu propósito?
36. O que é um pacote Java e qual o seu propósito?
37. Descreva uma montagem no .NET.
38. Que elementos podem aparecer em um módulo Ruby?

CONJUNTO DE PROBLEMAS

1. Alguns engenheiros de software acreditam que todas as entidades importadas devem ser qualificadas pelo nome da unidade de programa exportadora. Você concorda? Justifique sua resposta.
2. Suponha que alguém projetou um tipo de dados abstrato pilha no qual a função `top` retorna um caminho de acesso (ou ponteiro) em vez de retornar uma cópia do elemento do topo. Essa não é uma abstração de dados verdadeira. Por quê? Dê um exemplo que ilustre o problema.
3. Escreva uma análise das similaridades e das diferenças entre pacotes em Java e espaços de nomes em C++.
4. Quais são as desvantagens de projetar um tipo de dados abstrato como um ponteiro?

5. Por que a estrutura de tipos de dados abstratos não ponteiros precisa ser dada nos pacotes de especificação de Ada?
6. Que perigos são evitados em Java e em C# ao ter coleta de lixo implícita, em relação a C++?
7. Discuta as vantagens das propriedades em C#, em relação a escrever métodos de acesso em C++ ou Java.
8. Explique os perigos da abordagem de C para encapsulamento.
9. Por que C++ não eliminou os problemas discutidos no Problema 8?
10. Por que os destrutores não são necessários em Java, mas essenciais em C++?
11. Quais são os argumentos a favor e contra a política de internalizar métodos?
12. Descreva uma situação na qual uma estrutura C# seja preferível a uma classe C#.
13. Explique por que encapsulamentos de nomeação são importantes para desenvolver grandes programas.
14. Descreva as três maneiras pelas quais um cliente pode referenciar um nome a partir de um espaço de nomes C++.
15. O espaço de nomes da biblioteca padrão de C#, `System`, não é implicitamente disponível para os programas em C#. Você acha que essa é uma boa ideia? Defenda sua resposta.
16. Quais são as vantagens e desvantagens da habilidade de modificar objetos em Ruby?
17. Compare os pacotes de Java com os módulos de Ruby.

EXERCÍCIOS DE PROGRAMAÇÃO

1. Projete o exemplo do tipo abstrato pilha da Seção 11.2.3 em Fortran usando um único subprograma com múltiplas entradas para a definição de tipo e as operações.
2. Como a implementação do Exercício de Programação 1 em Fortran pode ser comparada à implementação de Ada deste capítulo em termos de confiabilidade e flexibilidade?
3. Projete um tipo de dados abstrato para uma matriz com elementos inteiros em uma linguagem que você conheça, incluindo operações para adição, subtração e multiplicação de matrizes.
4. Projete um tipo de dados abstrato fila para elementos de ponto flutuante em uma linguagem que você conheça, incluindo operações para inserir, remover e esvaziar a fila. A operação de remoção remove o elemento e retorna o seu valor.
5. Modifique a classe C++ para o tipo abstrato mostrado na Seção 12.4.2 para usar uma representação de lista encadeada e teste-a com o mesmo código que aparece neste capítulo.
6. Modifique a classe Java para o tipo abstrato mostrado na Seção 12.4.3 para usar uma representação de lista encadeada e teste-a com o mesmo código que aparece neste capítulo.
7. Escreva um tipo de dados abstrato para números complexos, incluindo operações para adição, subtração, multiplicação, divisão, extração de cada uma das partes de um número complexo e a construção de um número complexo a partir

de duas constantes, variáveis ou expressões de ponto flutuante. Use Ada, C++, Java, C# ou Ruby.

8. Escreva um tipo de dados abstrato para filas cujos elementos armazenem nomes de 10 caracteres. Os elementos da fila devem ser dinamicamente alocados do monte. As operações da fila são a inserção, a remoção e o esvaziamento. Use Ada, C++, Java, C# ou Ruby.
9. Escreva um tipo de dados abstrato para uma fila cujos elementos possam ser de qualquer tipo primitivo. Use Java 5.0, C# 2005, C++ ou Ada.
10. Escreva um tipo de dados abstrato para uma fila cujos elementos incluam tanto uma cadeia de 20 caracteres quanto uma prioridade inteira. Essa fila deve ter os seguintes métodos: inserir, que recebe uma cadeia e um inteiro como parâmetros, remover, que retorna a cadeia da fila que tem a prioridade mais alta, e esvaziar. A fila não deve ser mantida na ordem de prioridade dos elementos, então operações de remoção devem sempre buscar toda a fila.
11. Um deque é uma fila de duas extremidades, com operações de adição e remoção de elementos em ambas. Modifique a solução do Problema 9 para implementar um deque.