

Programação Orientada a Objetos com C++

Ivan Luiz Marques Ricarte

Departamento de Engenharia de Computação e Automação Industrial

Faculdade de Engenharia Elétrica

Universidade Estadual de Campinas

© 1995, DCA/FEE/UNICAMP

Conteúdo

1	Introdução a Orientação a Objetos	3
1.1	Definições	3
1.2	Conceitos Básicos	5
1.2.1	Abstração	5
1.2.2	Encapsulação	5
1.2.3	Compartilhamento	6
1.3	O Modelo de Objetos	6
1.3.1	Objetos e Classes	7
1.3.2	Ligações e Associações	10
1.3.3	Agregação	11
1.3.4	Generalização e Herança	12
1.4	Sugestões de desenvolvimento	13
2	Fundamentos de C++	15
2.1	Origens de C++	15
2.2	Similaridades com C	16
2.3	Particularidades de C++	17
2.3.1	Comentários	17
2.3.2	Operador de escopo	19
2.3.3	Entrada e Saída	20
2.3.4	Definição de variáveis	23
2.4	Tipos Compostos	25
2.4.1	Conversão de tipos	27
2.5	Apontadores	28
2.5.1	Revisão de apontadores	28
2.5.2	Alocação Dinâmica	30
2.5.3	Apontadores para funções	32
2.6	Funções	33
2.6.1	Protótipos	34
2.6.2	Passagem por referência	34
2.6.3	Parâmetros <i>default</i>	35

2.6.4	Número variável de argumentos	36
2.6.5	Sobrecarga de nome de funções	38
3	Encapsulação	40
3.1	Classes e Encapsulação	40
3.1.1	Definição de classes	41
3.1.2	Ocultamento da informação	43
3.1.3	Funções em linha	47
3.1.4	Construtores e Destrutores	47
3.1.5	Objetos e funções	49
3.2	Arranjos e apontadores	49
3.2.1	Arranjos	49
3.2.2	Apontadores	50
3.2.3	O apontador <i>this</i>	50
3.2.4	Alocação dinâmica de objetos	50
3.3	Sobrecarga de operadores	51
3.4	Amizade	52
3.5	Tratamento de exceções	52
4	Herança	55
4.1	Funcionamento básico	55
4.2	Controle de acesso	59
4.3	Herança múltipla	59
4.4	Herança de construtores e destrutores	60
5	Polimorfismo	62
5.1	Funções virtuais	62
5.2	Funções virtuais puras e classes abstratas	64
5.3	Gabaritos	65
5.3.1	Gabaritos de funções	65
5.3.2	Gabarito de classes	66
6	Desenvolvimento de Aplicações	68
6.1	Projeto e Implementação	68
6.2	Empacotamento de classes	69
6.2.1	Cabeçalhos	70
6.2.2	Implementação	71
6.3	Estilo de programação C++	71

Capítulo 1

Introdução a Orientação a Objetos

O termo *orientação a objetos* pressupõe uma organização de software em termos de coleção de objetos discretos incorporando estrutura e comportamento próprios. Esta abordagem de organização é essencialmente diferente do desenvolvimento tradicional de software, onde estruturas de dados e rotinas são desenvolvidas de forma apenas fracamente acopladas.

Neste capítulo, as primeiras noções de orientação a objetos serão introduzidas. Esta breve visão geral do paradigma permitirá entender melhor os conceitos associados à programação orientada a objetos e, em particular, às construções da linguagem C++.

1.1 Definições

Um *objeto* é uma entidade do mundo real que tem uma *identidade*. Objetos podem representar entidades concretas (um arquivo no meu computador, uma bicicleta) ou entidades conceituais (uma estratégia de jogo, uma política de escalonamento em um sistema operacional). Cada objeto ter sua identidade significa que dois objetos são distintos mesmo que eles apresentem exatamente as mesmas características.

Embora objetos tenham existência própria no mundo real, em termos de linguagem de programação um objeto necessita um mecanismo de identificação. Esta *identificação de objeto* deve ser única, uniforme e independente do conteúdo do objeto. Este é um dos mecanismos que permite a criação de coleções de objetos, as quais são também objetos em si.

A estrutura de um objeto é representada em termos de *atributos*. O comportamento de um objeto é representado pelo conjunto de *operações*

que podem ser executadas sobre o objeto. Objetos com a mesma estrutura e o mesmo comportamento são agrupados em *classes*. Uma classe é uma abstração que descreve propriedades importantes para uma aplicação e simplesmente ignora o resto.

Cada classe descreve um conjunto (possivelmente infinito) de objetos individuais. Cada objeto é dito ser uma *instância* de uma classe. Assim, cada instância de uma classe tem seus próprios valores para cada atributo, mas dividem os nomes dos atributos e métodos com as outras instâncias da classe. Implicitamente, cada objeto contém uma referência para sua própria classe — em outras palavras, ele sabe o que ele é.

Polimorfismo significa que a mesma operação pode se comportar de forma diferente em classes diferentes. Por exemplo, a operação *move* quando aplicada a uma janela de um sistema de interfaces tem um comportamento distinto do que quando aplicada a uma peça de um jogo de xadrez. Um *método* é uma implementação específica de uma operação para uma certa classe.

Polimorfismo também implica que uma operação de uma mesma classe pode ser implementada por mais de um método. O usuário não precisa saber quantas implementações existem para uma operação, ou explicitar qual método deve ser utilizado: a linguagem de programação deve ser capaz de selecionar o método correto a partir do nome da operação, classe do objeto e argumentos para a operação. Desta forma, novas classes podem ser adicionadas sem necessidade de modificação de código já existente, pois cada classe apenas define os seus métodos e atributos.

No mundo real, alguns objetos e classes podem ser descritos como casos especiais, ou *especializações*, de outros objetos e classes. Por exemplo, a classe de computadores pessoais com processador da linha 80x86 é uma especialização de computadores pessoais, que por sua vez é uma especialização de computadores. Não é desejável que tudo que já foi descrito para computadores tenha de ser repetido para computadores pessoais ou para computadores pessoais com processador da linha 80x86.

Herança é o mecanismo do paradigma de orientação a objetos que permite compartilhar atributos e operações entre classes baseada em um relacionamento hierárquico. Uma classe pode ser definida de forma genérica e depois refinada sucessivamente em termos de *subclasses* ou *classes derivadas*. Cada subclasse incorpora, or *herda*, todas as propriedades de sua *superclasse* (ou *classe base*) e adiciona suas propriedades únicas e particulares. As propriedades da classe base não precisam ser repetidas em cada classe derivada. Esta capacidade de fatorar as propriedades comuns de diversas classes em uma superclasse pode reduzir dramaticamente a repetição de código em um projeto ou programa, sendo uma das principais vantagens da abordagem de

orientação a objetos.

1.2 Conceitos Básicos

A abordagem de orientação a objetos favorece a aplicação de diversos conceitos considerados fundamentais para o desenvolvimento de bons programas, tais como abstração e encapsulação. Tais conceitos não são exclusivos desta abordagem, mas são suportados de forma melhor no desenvolvimento orientado a objetos do que em outras metodologias.

1.2.1 Abstração

Abstração consiste de focalizar nos aspectos essenciais inerentes a uma entidade e ignorar propriedades “acidentais.” Em termos de desenvolvimento de sistemas, isto significa concentrar-se no que um objeto é e faz antes de se decidir como ele será implementado. O uso de abstração preserva a liberdade para tomar decisões de desenvolvimento ou de implementação apenas quando há um melhor entendimento do problema a ser resolvido.

Muitas linguagens de programação modernas suportam o conceito de abstração de dados; porém, o uso de abstração juntamente com polimorfismo e herança, como suportado em orientação a objetos, é um mecanismo muito mais poderoso.

O uso apropriado de abstração permite que um mesmo modelo conceitual (orientação a objetos) seja utilizado para todas as fases de desenvolvimento de um sistema, desde sua análise até sua documentação.

1.2.2 Encapsulação

Encapsulação, também referido como *esconder informação*, consiste em separar os aspectos externos de um objeto, os quais são acessíveis a outros objetos, dos detalhes internos de implementação do objeto, os quais permanecem escondidos dos outros objetos. O uso de encapsulação evita que um programa torne-se tão interdependente que uma pequena mudança tenha grandes efeitos colaterais.

O uso de encapsulação permite que a implementação de um objeto possa ser modificada sem afetar as aplicações que usam este objeto. Motivos para modificar a implementação de um objeto podem ser por exemplo melhoria de desempenho, correção de erros e mudança de plataforma de execução.

Assim como abstração, o conceito de encapsulação não é exclusivo da abordagem de orientação a objetos. Entretanto, a habilidade de se combinar

estrutura de dados e comportamento em uma única entidade torna a encapsulação mais elegante e mais poderosa do que em linguagens convencionais que separam estruturas de dados e comportamento.

1.2.3 Compartilhamento

Técnicas de orientação a objetos promovem compartilhamento em diversos níveis distintos. Herança de estrutura de dados e comportamento permite que estruturas comuns sejam compartilhadas entre diversas classes derivadas similares sem redundância. O compartilhamento de código usando herança é uma das grandes vantagens da orientação a objetos. Ainda mais importante que a economia de código é a clareza conceitual de reconhecer que operações diferentes são na verdade a mesma coisa, o que reduz o número de casos distintos que devem ser entendidos e analisados.

O desenvolvimento orientado a objetos não apenas permite que a informação dentro de um projeto seja compartilhada como também oferece a possibilidade de reaproveitar projetos e código em projetos futuros. As ferramentas para alcançar este compartilhamento, tais como abstração, encapsulação e herança, estão presentes na metodologia; uma estratégia de reuso entre projetos é a definição de bibliotecas de elementos reusáveis. Entretanto, orientação a objetos não é uma fórmula mágica para alcançar reusabilidade; para tanto, é preciso planejamento e disciplina para pensar em termos genéricos, não voltados simplesmente para a aplicação corrente.

1.3 O Modelo de Objetos

Um modelo de objetos busca capturar a estrutura estática de um sistema mostrando os objetos existentes, seus relacionamentos, e atributos e operações que caracterizam cada classe de objetos. É através do uso deste modelo que se enfatiza o desenvolvimento em termos de objetos ao invés de mecanismos tradicionais de desenvolvimento baseado em funcionalidades, permitindo uma representação mais próxima do mundo real.

Uma vez que as principais definições e conceitos da abordagem de orientação a objetos estão definidos, é possível introduzir o *modelo de objetos* que será adotado ao longo deste texto. O modelo apresentado é um subconjunto do modelo OMT (Object Modeling Technique), proposto por Rumbaugh e outros¹. OMT também introduz uma representação diagramática para este modelo, a qual será também apresentada aqui.

¹James Rumbaugh *et al.*: *Object-Oriented Modeling and Design*. Prentice-Hall International Editions, 1991.

1.3.1 Objetos e Classes

Objeto é definido neste modelo como um conceito, abstração ou coisa com limites e significados bem definidos para a aplicação em questão. Objetos têm dois propósitos: promover o entendimento do mundo real e suportar uma base prática para uma implementação computacional. Não existe uma maneira “correta” de decompor um problema em objetos; esta decomposição depende do julgamento do projetista e da natureza do problema. Todos objetos têm identidade própria e são distinguíveis.

Uma *classe de objetos* descreve um grupo de objetos com propriedades (atributos) similares, comportamento (operações) similares, relacionamentos comuns com outros objetos e uma semântica comum. Por exemplo, *Pessoa* e *Companhia* são classes de objetos. Cada pessoa tem um nome e uma idade; estes seriam os atributos comuns da classe. Companhias também podem ter os mesmos atributos nome e idade definidos. Entretanto, devido à distinção semântica elas provavelmente estariam agrupados em outra classe que não *Pessoa*. Como se pode observar, o agrupamento em classes não leva em conta apenas o compartilhamento de propriedades.

Todo objeto sabe a que classe ele pertence, ou seja, a classe de um objeto é um atributo implícito do objeto. Este conceito é suportado na maior parte das linguagens de programação orientada a objetos, tais como C++.

OMT define dois tipos de diagramas de objetos, diagramas de classes e diagramas de instâncias. Um diagrama de classe é um *esquema*, ou seja, um padrão ou gabarito que descreve as muitas possíveis instâncias de dados. Um diagrama de instâncias descreve como um conjunto particular de objetos está relacionado. Diagramas de instâncias são úteis para apresentar exemplos e documentar casos de testes; diagramas de classes têm uso mais amplo. A Figura 1.1 apresenta a notação adotada para estes diagramas.

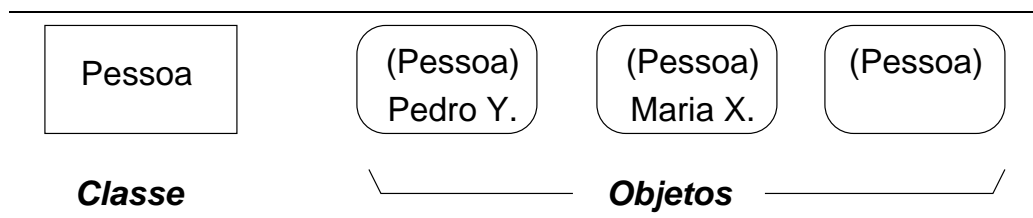


Figura 1.1: Representação diagramática de OMT para classes e objetos. Um diagrama de classe é apresentado à esquerda. Um possível diagrama de instâncias é apresentado à direita.

O agrupamento de objetos em classes é um poderoso mecanismo de ab-

stração. Desta forma, é possível generalizar definições comuns para uma classe de objetos, ao invés de repetí-las para cada objeto em particular. Esta é uma das formas de reutilização e economia que a abordagem de orientação a objetos suporta.

Atributos

Um atributo é um valor de dado assumido pelos objetos de uma classe. Nome, idade e peso são exemplos de atributos de objetos *Pessoa*. Cor, peso e modelo são possíveis atributos de objetos *Carro*. Cada atributo tem um valor para cada instância de objeto. Por exemplo, o atributo *idade* tem valor “29” no objeto *Pedro Y*. Em outras palavras, Pedro Y tem 29 anos de idade. Diferentes instâncias de objetos podem ter o mesmo valor para um dado atributo.

Cada nome de atributo é único para uma dada classe, mas não necessariamente único entre todas as classes. Por exemplo, ambos *Pessoa* e *Companhia* podem ter um atributo chamado *endereço*.

No diagrama de classes, atributos são listados no segundo segmento da caixa que representa a classe. O nome do atributo pode ser seguido por detalhes opcionais, tais como o tipo de dado assumido e valor *default*. A Figura 1.2 mostra esta representação.

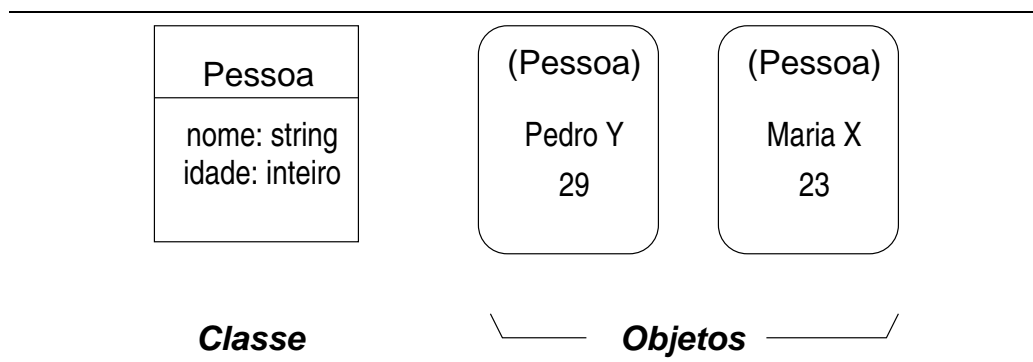


Figura 1.2: Representação diagramática de OMT para classes e objetos com atributos. Um diagrama de classe com atributos é apresentado à esquerda. Um possível diagrama de instâncias com os respectivos valores é apresentado à direita.

Não se deve confundir identificadores internos de objetos com atributos do mundo real. Identificadores de objetos são uma conveniência de implementação, e não têm nenhum significado para o domínio da aplicação. Por

exemplo, CIC e RG não são identificadores de objetos, mas sim verdadeiros atributos do mundo real.

Operações e Métodos

Uma operação é uma função ou transformação que pode ser aplicada a ou por objetos em uma classe. Por exemplo, *abrir*, *salvar* e *imprimir* são operações que podem ser aplicadas a objetos da classe *Arquivo*. Todos objetos em uma classe compartilham as mesmas operações.

Toda operação tem um objeto-alvo como um argumento implícito. O comportamento de uma operação depende da classe de seu alvo. Como um objeto “sabe” qual sua classe, é possível escolher a implementação correta da operação. Além disto, outros argumentos (parâmetros) podem ser necessários para uma operação.

Uma mesma operação pode se aplicar a diversas classes diferentes. Uma operação como esta é dita ser *polimórfica*, ou seja, ela pode assumir distintas formas em classes diferentes.

Um *método* é a implementação de uma operação para uma classe. Por exemplo, a operação *imprimir* pode ser implementada de forma distinta, dependendo se o arquivo a ser impresso contém apenas texto ASCII, é um arquivo de um processador de texto ou binário. Todos estes métodos executam a mesma operação — imprimir o arquivo; porém, cada método será implementado por um diferente código.

A *assinatura* de um método é dada pelo número e tipos de argumentos do método, assim como por seu valor de retorno. Uma estratégia de desenvolvimento recomendável é manter assinaturas coerentes para métodos implementando uma dada operação, assim como um comportamento consistente entre as implementações.

Em termos de diagramas OMT, operações são listadas na terceira parte da caixa de uma classe. Cada nome de operação pode ser seguida por detalhes opcionais, tais como lista de argumentos e tipo de retorno. A lista de argumentos é apresentada entre parênteses após o nome da operação. Uma lista de argumentos vazia indica que a operação não tem argumentos; da ausência da lista de argumentos não se pode concluir nada. O tipo de resultado vem após a lista de argumentos, sendo precedido por dois pontos (:). Caso a operação retorne resultado, este não deve ser omitido — esta é a forma de distingui-la de operações que não retornam resultado. Exemplos de representação de operações em OMT são apresentados na Figura 1.3.

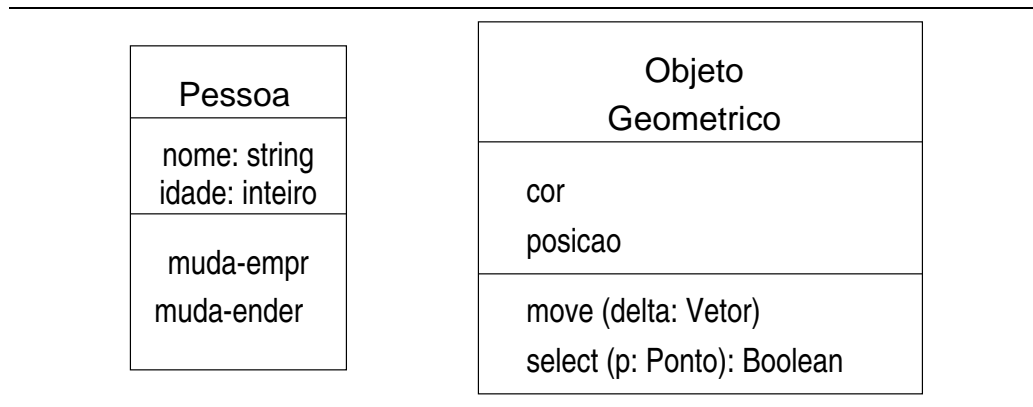


Figura 1.3: Representação diagramática de OMT para classes com atributos e operações.

1.3.2 Ligações e Associações

Ligações e associações são os mecanismos para estabelecer relacionamentos entre objetos e classes. Uma ligação é uma conexão física ou conceitual entre duas instâncias de objetos. Por exemplo, Pedro Y *trabalha-para* Companhia W. Uma ligação é uma instância de uma associação. Uma associação descreve um grupo de ligações com estrutura e semântica comuns, tal como “uma pessoa *trabalha-para* uma companhia.” Uma associação descreve um conjunto de ligações potenciais da mesma forma que uma classe descreve um conjunto de objetos potenciais.

A notação de diagramas OMT para associação é uma linha conectando duas classes. Uma ligação é representada como uma linha conectando objetos. Nomes de associações são usualmente apresentada em itálico. Se entre um par de classes só existe uma única associação cujo sentido deva ser óbvio, então o nome da associação pode ser omitido. A Figura 1.4 apresenta um exemplo de diagrama OMT com associações.

Alguns atributos podem dizer respeito a associações, e não a classes. Para tais casos, OMT introduz o conceito de *atributo de ligação*. Quando a associação tem ainda operações associadas, então ela pode ser modelada como uma classe que está “conectada” à associação. Um exemplo deste caso é apresentado na Figura 1.5.

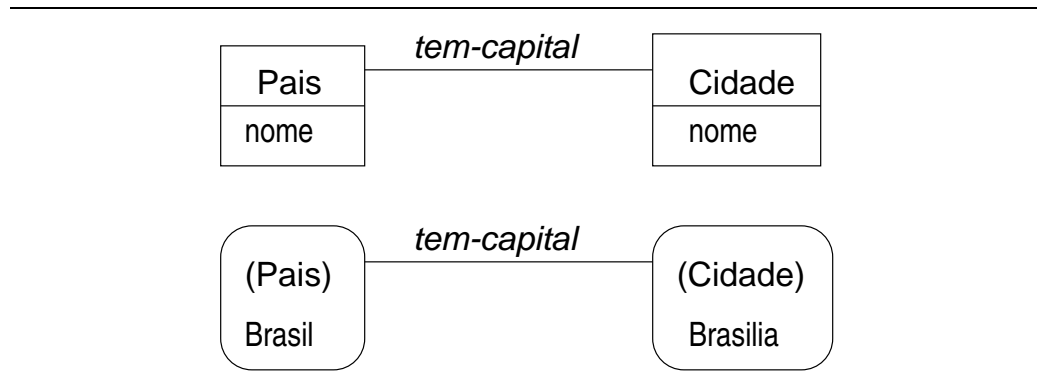


Figura 1.4: Representação diagramática de OMT para associações entre classes (topo) e ligações entre objetos (abaixo).

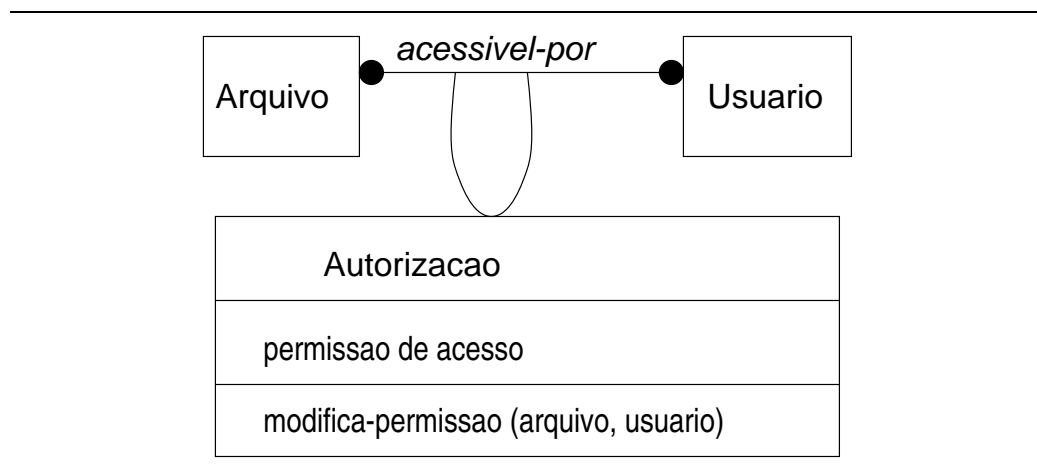


Figura 1.5: Representação diagramática de OMT para associações entre classes com atributos. Neste caso, os atributos da associação estão representados através de uma classe explícita, *Autorização*. O círculo preto no final da linha da associação indica que mais de um objeto de uma classe podem estar associados a cada objeto da outra classe. Um círculo vazado indicaria que possivelmente nenhum objeto poderia estar associado, ou seja, o conceito de associação opcional.

1.3.3 Agregação

Uma *agregação* é um relacionamento do tipo “uma-parte-de,” nos quais objetos representando os componentes de alguma coisa são associados com objetos representando uma *montagem*. Por exemplo, o texto de um documento

pode ser visto como um conjunto de parágrafos, e cada parágrafo é um conjunto de sentenças (Figura 1.6).

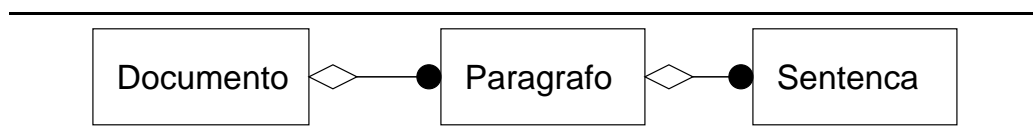


Figura 1.6: Representação diagramática de OMT para agregação.

Agregação é uma forma de associação com alguma semântica adicional. Por exemplo, agregação é *transitiva*: se A é parte de B e B é parte de C , então A é parte de C . Agregação também é *anti-simétrica*: se A é parte de B , então B não é parte de A .

1.3.4 Generalização e Herança

Generalização e herança são abstrações poderosas para compartilhar similaridades entre classes e ao mesmo tempo preservar suas diferenças.

Generalização é o relacionamento entre uma classe e um ou mais versões refinadas (especializadas) desta classe. A classe sendo refinada é chamada de superclasse ou *classe base*, enquanto que a versão refinada da classe é chamada uma subclasse ou *classe derivada*. Atributos e operações comuns a um grupo de classes derivadas são colocadas como atributos e operações da classe base, sendo compartilhados por cada classe derivada. Diz-se que cada classe derivada *herda* as características de sua classe base. Algumas vezes, generalização é chamada de relacionamento *is-a* (é-um), porque cada instância de uma classe derivada é também uma instância da classe base.

Generalização e herança são transitivas, isto é, podem ser recursivamente aplicadas a um número arbitrário de níveis. Cada classe derivada não apenas herda todas as características de todos seus ancestrais como também pode acrescentar seus atributos e operações específicos.

A Figura 1.7 mostra a notação diagramática de OMT para representar generalização, um triângulo com o vértice apontado para a classe base. Um *discriminador* pode estar associado a cada associação do tipo generalização; este é um atributo do tipo enumeração que indica qual a propriedade de um objeto está sendo abstraída pelo relacionamento de generalização. Este discriminador é simplesmente um nome para a base de generalização.

Uma classe derivada pode *sobrepôr*² uma característica de sua classe base

²Override.

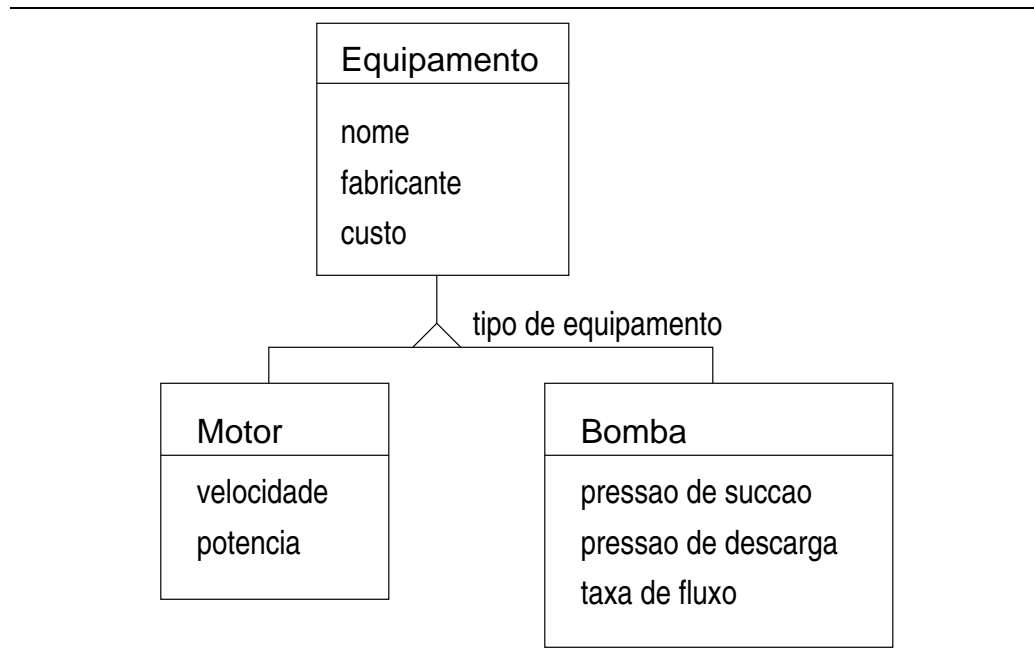


Figura 1.7: Representação diagramática de OMT para generalização.

definindo uma característica própria com o mesmo nome. A característica local (da classe derivada) irá refinar e substituir a característica da classe base. Uma característica pode ser sobreposta, por exemplo, por questões de refinamento de especificação ou por questões de desempenho.

Entre as características que podem ser sobrepostas estão valores *default* de atributos e métodos de operação. Uma boa estratégia de desenvolvimento não deve sobrepor uma característica de forma inconsistente com a semântica da classe base.

1.4 Sugestões de desenvolvimento

Na construção de um modelo para uma aplicação, as seguintes sugestões devem ser observadas a fim de se obter resultados claros e consistentes:

1. Não comece a construir um modelo de objetos simplesmente definindo classes, associações e heranças. A primeira coisa a se fazer é entender o problema a ser resolvido.
2. Tente manter seu modelo simples. Evite complicações desnecessárias.

3. Escolha nomes cuidadosamente. Nomes são importantes e carregam conotações poderosas. Nomes devem ser descritivos, claros e não deixar ambiguidades. A escolha de bons nomes é um dos aspectos mais difíceis da modelagem.
4. Não “enterre” apontadores ou outras referências a objetos dentro de objetos como atributos. Ao invés disto, modele estas referências como associações. Isto torna o modelo mais claro e independente da implementação.
5. Tente evitar associações que envolvam três ou mais classes de objetos. Muitas vezes, estes tipos de associações podem ser decompostos em termos de associações binárias, tornando o modelo mais claro.
6. Não transfira os atributos de ligação para dentro de uma das classes.
7. Tente evitar hierarquias de generalização muito profundas.
8. Não se surpreenda se o seu modelo necessitar várias revisões; isto é o normal.
9. Sempre documente seus modelos de objetos. O diagrama pode especificar a estrutura do modelo, mas nem sempre é suficiente para descrever as razões por trás da definição do modelo. Uma explicação escrita pode clarificar pontos tais como significado de nomes e explicar a razão para cada classe e relacionamento.
10. Nem sempre todas as construções OMT são necessárias para descrever uma aplicação. Use apenas aquelas que forem adequadas para o problema analisado.

Capítulo 2

Fundamentos de C++

Neste capítulo será iniciado o estudo da linguagem C++ e sua aplicação à programação orientada a objetos. Aqui, particularmente, serão abordados os princípios da linguagem, sem necessariamente ater-se aos conceitos de orientação a objetos.

Não se pretende aqui ensinar princípios de programação. Conhecimentos de programação serão assumidos, de forma que será possível se concentrar nos detalhes da linguagem C++.

2.1 Origens de C++

C++ é uma extensão da linguagem de programação C. As extensões de C++ sobre C foram primeiramente introduzidas por Bjarne Stroustrup em 1980 nos Laboratórios Bell de New Jersey. Inicialmente, a linguagem era chamada “C com classes”, mas o nome foi alterado para C++ em 1983.

A motivação para o desenvolvimento de C++ foi *complexidade*. Grandes sistemas implementados com a linguagem C, na ordem de 25000 a 100000 linhas de código, são difíceis de controlar ou mesmo entender sua totalidade. C++ surgiu para permitir que esta barreira seja quebrada. O objetivo de C++ é permitir que programadores possam gerenciar e compreender programas maiores e mais complexos.

A maior parte das adições introduzidas por Stroustrup suportam a programação orientada a objetos. Algumas das características de C++ foram inspiradas em outra linguagem, Simula67. Apesar de não ser exatamente uma linguagem orientada a objetos, Simula67 suportava diversos dos conceitos de abstração que são fundamentais para a programação orientada a objetos. A primeira linguagem orientada a objetos com repercussão significativa foi Smalltalk, desenvolvida no final da década de 70. Entretanto, seu

uso foi e continua sendo muito restrito — C++ é a linguagem adotada pela maior parte de empresas e centros desenvolvendo software em grande escala.

Quando C++ foi inventada, Stroustrup sabia que era importante manter o espírito original de C, incluindo sua eficiência, flexibilidade e a filosofia de que o programador, e não a linguagem, é o encarregado, e queria ainda ao mesmo tempo adicionar suporte para a programação orientada a objetos. Estes objetivos foram alcançados com C++. C++ ainda oferece ao programador a liberdade e o controle de C acoplada com o poder dos objetos. As características de orientação a objetos de C++, nas palavras de Stroustrup, “permite que programas sejam estruturados para clareza, extensibilidade e facilidade de manutenção sem perda de eficiência.”

Embora C++ fosse inicialmente projeto para ajudar na gerência de programas muito grandes, seu uso não se limita apenas a estes casos. Na verdade, os atributos orientados a objetos de C++ podem ser efetivamente aplicados a praticamente qualquer tarefa de programação. C++ já vem sendo utilizado em projetos tais como editores, bancos de dados, sistemas pessoais de arquivos e programas de comunicação. Com a mesma eficiência de C, C++ pode ser usado para construir software de sistemas com ótimo desempenho.

2.2 Similaridades com C

C++ tem muito a ver com a linguagem C, da qual ela é derivada. Os tipos e formatos de expressões, operadores e comandos da linguagem são os mesmos. Arranjos, *strings*, apontadores e funções são também definidos e manipulados como em C.

Os tipos de dados básicos em C++ são os mesmos que são definidos pelo padrão ANSI para a linguagem C (usualmente referenciado como ANSI-C). Para fins de referência, estes tipos são apresentados na Tabela 2.1.

Deve-se observar que o padrão ANSI admite que estes tipos têm tamanho e faixas de valores variáveis. Entretanto, a sequência de ordenação de tamanhos — por exemplo, um *char* é menor que um *int* que é menor que um *long* — deve ser independente da plataforma em que o programa será compilado.

Assim como para a programação em C, cuidados especiais devem ser tomados para garantir a portabilidade dos programas desenvolvidos. Quando conhecer o tamanho de uma variável de um dado tipo é importante para a correta execução do programa, o operador *sizeof* deve ser utilizado. Este operador retorna o número de bytes que um tipo de dado ou variável ocupa na plataforma de execução. Por exemplo, em um computador da linha PC *sizeof(int)* retorna o valor 2 para a maior parte dos compiladores.

A ausência de um tipo de dado (por exemplo, o valor de retorno de uma

Tipo	Exemplo de Aplicação
unsigned char	Números pequenos, caracteres ASCII estendido
char	Números muito pequenos, caracteres ASCII
enum	Conjuntos ordenados de valores
unsigned int	Números e laços grandes
short int	Contagem, números pequenos, controle de laço
int	Contagem, números pequenos, controle de laço
unsigned long	Distâncias astronômicas
long	Números grandes, populações
float	Científico, precisão pequena
double	Científico, precisão maior
long double	Científico e Financeiro, grande precisão

Tabela 2.1: Tipos de dados em C++.

função que não retorna nada) é indicado pelo uso da palavra chave *void*.

2.3 Particularidades de C++

Nesta seção, as particularidades de C++ em relação à proposta de ANSI-C são apresentadas.

2.3.1 Comentários

C++ introduz uma construção para facilitar a representação de comentários de uma única linha — a *barra dupla* `//`, que pode começar em qualquer posição de uma linha e considera como comentário o restante da linha. Apesar de o padrão C de comentários — início com `/*` e final com `*/` — ainda ser válido, o método da barra dupla deve ser adotado como parte das boas regras de programação C++.

O exemplo a seguir demonstra a utilização de comentários em C++.

```
#include <iostream.h>           /* This is the stream definition file */

void print_it(const int data_value);

main()
{
    const int START = 3;         // The value of START cannot be changed
    const int STOP = 9;          // The value of STOP cannot be changed
    volatile int CENTER = 6;     /* The value of CENTER may be changed
```

```

                                by something external to this
                                program.                               */
int index;                      /* A normal C variable                */

    for (index = START ; index < STOP ; index++)
        print_it(index);
}  /* End of program */

void print_it(const int data_value)
{
    cout << "The value of the index is " << data_value << "\n";
}

```

Este programa também demonstra o uso do operador << para apresentação de dados no console; este operador será analisado em mais detalhes adiante.

O resultado da execução deste programa deve ser:

```

The value of the index is  3
The value of the index is  4
The value of the index is  5
The value of the index is  6
The value of the index is  7
The value of the index is  8

```

Observe também neste exemplo o uso das palavras chave *const* e *volatile*. Estas palavras chave fazem parte de ANSI-C, apesar de não serem definidas originalmente na linguagem C¹.

A palavra chave *const* é usada para definir uma constante — o compilador não permitirá que o valor de uma variável “const” seja modificado, seja por engano, seja deliberadamente. Uma vez que o valor destas variáveis não podem ser modificados no programa, elas *devem* ser inicializadas no momento de sua declaração. Observe que esta construção substitui com vantagens a forma tradicionalmente adotada na linguagem C para este fim — o uso da diretiva *#define* do pré-processador.

Neste exemplo, *const* também é usada na lista de argumentos da função — isto indica que o valor pela função recebido também não pode ser modificado.

A palavra chave *volatile* indica que a variável pode ser modificada não apenas pelo programador mas também por outros mecanismos — tais como

¹Por “definição original de C” nos referimos à definição de Kernigham e Ritchie, também referenciada como “K&R C”, apresentada no livro *The C Programming Language*.

interrupções do sistema. Se uma variável for ao mesmo tempo constante e volátil, ela só poderá ser modificada pelo sistema, e nunca pelo programador.

Outra observação com relação a este program está no uso de protótipos, a declaração que explicita os tipos de dados para cada argumento de uma função. Protótipos, opcionais em ANSI-C, são mandatórios em C++.

2.3.2 Operador de escopo

Outra construção que é nova em C++ é o operador de escopo (::). Não há correspondente para este operador em K&R ou ANSI-C: ele permite acessar uma variável global mesmo que exista uma variável local com o mesmo nome. O seguinte exemplo mostra o uso deste operador.

```
#include <iostream.h>

int index = 13;

main()
{
    float index = 3.1415;

    cout << "The local index value is " << index << "\n";
    cout << "The global index value is " << ::index << "\n";

    ::index = index + 7;                                // 3 + 7 should result in 10

    cout << "The local index value is " << index << "\n";
    cout << "The global index value is " << ::index << "\n";
}
```

O resultado da execução deste programa seria:

```
The local index value is 3.1415
The global index value is 13
The local index value is 3.1415
The global index value is 10
```

Deve-se observar que, apesar de disponível em C++, este tipo de construção não é recomendado; o próprio uso de variáveis globais não é considerado boa prática de programação. Entretanto, caso seja realmente necessário, o mecanismo de acesso a variáveis globais está disponível.

2.3.3 Entrada e Saída

C++, assim como C, não oferece operações de entrada e saída como parte da linguagem em si. Ao invés disto, ela oferece uma biblioteca que adiciona as funções de entrada e saída de forma elegante, usando o conceito de *streams*.

O próximo exemplo mostra, ainda que de forma muito simples, o uso dos métodos de entrada e saída. Algumas variáveis são definidas e enviadas para o dispositivo de saída padrão (o monitor) através do stream *cout*. Observe como, ao contrário do que acontece com *printf* em C, o programador não tem de dizer ao sistema que tipo de dado está sendo enviado para a saída.

```
#include <iostream.h>
#include <string.h>

main()
{
    int index;
    float distance;
    char letter;
    char name[25];

    index = -23;
    distance = 12.345;
    letter = 'X';
    strcpy(name, "John Doe");

    cout << "The value of index is " << index << "\n";
    cout << "The value of distance is " << distance << "\n";
    cout << "The value of letter is " << letter << "\n";
    cout << "The value of name is " << name << "\n";

    index = 31;
    cout << "The decimal value of index is " << dec << index << "\n";
    cout << "The octal value of index is " << oct << index << "\n";
    cout << "The hex value of index is " << hex << index << "\n";
    cout << "The character letter is " << (char)letter << "\n";

    cout << "Input a decimal value --> ";
    cin >> index;
    cout << "The hex value of the input is " << index << "\n";
}
```

O resultado da execução deste programa seria:

```
The value of index is -23
The value of distance is 12.345
The value of letter is X
The value of name is John Doe
The decimal value of index is 31
The octal value of index is 37
The hex value of index is 1f
The character letter is X
Input a decimal value --> 999
The hex value of the input is 3e7
```

O operador `<<`, chamado de *operador de inserção*, diz para o sistema para enviar a seguinte variável ou constante para a saída, mas deixa o sistema decidir como apresentar o dado. Observe a consistência que este operador apresenta: em uma mesma linha de código, ele é utilizado para apresentar um string de caracteres e números (inteiros, ponto flutuante), sem que o programador tenha que se preocupar qual a rotina que será chamada para a apresentação. O sistema é encarregado de decidir qual a rotina apropriada. Este comportamento já sugere algumas das vantagens da orientação a objetos, como observado no Capítulo 1.

A grande vantagem da biblioteca de *streams* de C++ é que ela suporta boa parte das necessidades de entrada e saída de forma simples e eficiente. Os compiladores C++ podem também oferecer algumas facilidades adicionais de formatação para esta biblioteca, mas em último caso a rotina tradicional *printf* pode continuar sendo usada juntamente com *cout* sempre que necessário. Entretanto, deve-se lembrar que *printf* não é uma rotina muito eficiente, pois ela deve estar preparada para resolver todos os casos possíveis de formatação de saída — ou seja, é uma rotina complexa. Algumas das facilidades de formatação são apresentadas no exemplo acima: o mesmo dado é apresentado em formatos decimal, octal e hexadecimal.

Além do stream *cout*, há também o stream *cin* que é usado para ler dados do dispositivo de entrada padrão. Neste caso, o operador `>>`, chamado de *operador de extração*, é utilizado. Como no uso de *cout*, os operadores *dec*, *oct* e *hex* podem selecionar a base de numeração adotada para a entrada; como usual, a base decimal é o padrão quando nada é especificado.

Há também um stream *cerr* pré-definido, que envia dados para o dispositivo de erros padrão. Estes três streams, *cout*, *cin* e *cerr* correspondem aos apontadores para stream *stdout*, *stdin* e *stderr* da linguagem C, e são automaticamente abertos e fechados pelo sistema.

Os mesmos mecanismos de manipulação de streams podem ser utilizados para manipular arquivos do usuário. O próximo exemplo ilustra as capaci-

dades de manipulação de arquivos através da biblioteca de *streams* de C++.

```
#include <iostream.h>
#include <fstream.h>
#include <process.h>

void main()
{
    ifstream infile;
    ofstream outfile;
    ofstream printer;
    char filename[20];

    cout << "Enter the desired file to copy ----> ";

    cin >> filename;

    infile.open(filename, ios::nocreate);
    if (!infile) {
        cout << "Input file cannot be opened.\n";
        exit(1);
    }

    outfile.open("copy");
    if (!outfile) {
        cout << "Output file cannot be opened.\n";
        exit(1);
    }

    printer.open("PRN");
    if (!printer) {
        cout << "There is a problem with the printer.\n";
        exit(1);
    }

    cout << "All three files have been opened.\n";

    char one_char;

    printer << "This is the beginning of the printed copy.\n\n";

    while (infile.get(one_char)) {
        outfile.put(one_char);
        printer.put(one_char);
    }
}
```

```
    }

    printer << "\n\nThis is the end of the printed copy.\n";

    infile.close();
    outfile.close();
    printer.close();

}
```

Neste exemplo, um arquivo é aberto para leitura, outro para escrita, e um terceiro é aberto para a impressora. A diferença entre estes arquivos e aqueles do exemplo anterior é que aqueles já estavam abertos pelo sistema, enquanto que estes têm de ser explicitamente abertos e fechados. (Apesar de um arquivo ser fechado automaticamente ao fim da execução de um programa que o abriu, fechar os arquivos que foram abertos é sempre uma boa prática de programação.)

Como resultado da execução deste programa (caso o arquivo de entrada exista e não haja problemas com a impressora), o arquivo de entrada seria copiado para um outro arquivo com nome `copy` e seria também impresso através da impressora.

2.3.4 Definição de variáveis

Assim como em C, variáveis globais e estáticas em C++ são automaticamente inicializadas para o valor 0 quando declaradas caso um valor diferente não seja especificado. Variáveis automáticas, que são declaradas dentro do escopo de uma função, não são inicializadas pelo sistema, e receberão o valor que (por acaso) esteja na posição de memória correspondente àquela variável — em outras palavras, recebe “lixo”. O próximo exemplo ilustra estas diferenças.

```
#include <iostream.h>

int index;

main()
{
    int stuff;
    int &another_stuff = stuff;                // A synonym for stuff

    stuff = index + 14;                        //index was initialized to zero
    cout << "stuff has the value " << stuff << "\n";
}
```



```
stuff = 17;
cout << "another_stuff has the value " << another_stuff << "\n";

int more_stuff = 13;                                //not automatically initialized

cout << "more_stuff has the value " << more_stuff << "\n";

for (int count = 3; count < 8; count++) {
    cout << "count has the value " << count << "\n";
    char count2 = count + 65;
    cout << "count2 has the value " << count2 << "\n";
}

static unsigned goofy;                             //automatically initialized to zero

cout << "goofy has the value " << goofy << "\n";
}
```

O resultado da execução deste programa seria

```
stuff has the value 14
another_stuff has the value 17
more_stuff has the value 13
count has the value 3
count2 has the value D
count has the value 4
count2 has the value E
count has the value 5
count2 has the value F
count has the value 6
count2 has the value G
count has the value 7
count2 has the value H
goofy has the value 0
```

Observe o uso do símbolo & neste exemplo. Neste caso, *another_stuff* é definido como uma *variável de referência*, o que é também uma particularidade de C++.

A variável de referência não é usualmente usada neste contexto; o objetivo neste caso é apenas de ilustrar sua funcionalidade. Esta variável não funciona exatamente como as outras variáveis porque ela funciona como se fosse um apontador derreferenciado. Seguindo sua inicialização, a variável de referência se torna um sinônimo para a variável *stuff*; mudanças no valor de

stuff se refletirão no valor de *another_stuff*, pois efetivamente elas se referem ambas para a mesma variável. O sinônimo pode ser usado para acessar a variável para qualquer propósito legal da linguagem.

Deve-se observar que a variável de referência *deve* ser inicializada para referenciar outra variável quando ela é declarada; caso contrário, o compilador deve acusar um erro. Após a inicialização, a variável de referência não pode ser modificada para indicar uma variável diferente.

No contexto em que foi mostrado neste exemplo, o uso de variáveis de referência pode levar à confusão de código. Entretanto, mais adiante serão apresentados casos onde seu uso leva, de fato, a codificação mais clara e elegante.

O exemplo anterior também ilustra uma outra característica de C++, a possibilidade de se definir variáveis (como *more_stuff* e *goofy*) no meio de uma função. Em C, todas variáveis de uma função deviam ser definidas antes do primeiro comando; em C++, variáveis podem ser definidas mais próximas dos pontos onde elas serão efetivamente utilizadas. Tais variáveis têm escopo a partir do ponto onde são declaradas até o final do bloco onde foram definidas.

Observe também neste exemplo o uso desta característica no comando de laço *for*: a variável *count* é definida no apenas no início do laço, no ponto onde ela será utilizada. Neste caso, como esta variável foi definida fora do bloco do laço, seu escopo permanece até o fim da função (fim do bloco onde foi definida). Já a variável *count2*, definida dentro do bloco do laço, é liberada a cada fim de execução do laço e então realocada na próxima execução do laço (cinco vezes).

2.4 Tipos Compostos

Tipos compostos em C++ são enumeração, estrutura, união e classes.

O uso de enumeração e estrutura é similar ao uso em linguagem C, exceto por uma pequena diferença: não é necessário repetir a palavra chave (*enum* e *struct*, respectivamente) para declarar uma variável deste tipo composto. Por exemplo, se um tipo de enumeração foi previamente declarado como:

```
enum game_result {win,lose,tie};
```

então uma variável deste tipo pode ser declarada simplesmente como

```
game_result outcome;
```

ou seja, *outcome* é uma variável que pode apenas assumir os valores *win*, *lose* ou *tie*.

Uniãos em C++ apresentam ainda outra característica não presente em C: uma união pode ser *livre*, isto é. não receber um nome. Esta é uma característica prática quando embutindo uniões dentro de estruturas: quando a união tem obrigatoriamente um nome, como exigido em ANSI-C, isto acrescenta um nível extra de indireção. Em C++, este nível extra não é necessário.

A definição de classes traz o maior potencial de C++, e este assunto será abordado em mais detalhes adiante. Neste momento, entretanto, é interessante notar que classes podem ser definidas e acessadas praticamente da mesma maneira que estruturas. O exemplo abaixo ilustra estas similaridades através da definição de uma classe muito simples.

```
#include <iostream.h>

class animal {
public:
    int weight;
    int feet;
};

main()
{
    animal dog1, dog2, chicken;
    animal cat1;
    class animal cat2;

    dog1.weight = 15;
    dog2.weight = 37;
    chicken.weight = 3;

    dog1.feet = 4;
    dog2.feet = 4;
    chicken.feet = 2;

    cout << "The weight of dog1 is " << dog1.weight << "\n";
    cout << "The weight of dog2 is " << dog2.weight << "\n";
    cout << "The weight of chicken is " << chicken.weight << "\n";
}
```

O resultado da execução deste programa deve ser

```
The weight of dog1 is 15
The weight of dog2 is 37
The weight of chicken is 3
```

Caso *animal* tivesse sido declarado como uma estrutura, então *dog1*, *dog2*, *chicken*, *cat1* e *cat2* seriam variáveis de um tipo de estrutura. Como *animal* é uma classe, então estas variáveis são chamadas de *objetos*. Embora a diferença possa parecer mínima neste exemplo, ficará claro ao longo deste texto que outras diferenças, muito mais fundamentais, existem.

Observe também o uso da palavra chave *public* na definição da classe. Ela é necessária porque todas as variáveis em uma classe são, por *default*, privadas; isto é, elas não poderiam ser acessadas de forma direta como foi feito neste exemplo.

2.4.1 Conversão de tipos

A forma tradicional (ANSI-C) de conversão de tipos, através de *casts*, é também aceita em C++. Entretanto, uma nova forma é introduzida em C++: a conversão de forma equivalente à chamada de uma função.

O exemplo abaixo ilustra estes formatos de conversão. O primeiro bloco mostra a forma tradicional (por *casts*), enquanto o segundo bloco da função ilustra a forma funcional introduzida por C++.

```
#include <iostream.h>
```

```
main()
```

```
{
```

```
int a = 2;
```

```
float x = 17.1, y = 8.95, z;
```

```
char c;
```

```
    c = (char)a + (char)x;
```

```
    c = (char)(a + (int)x);
```

```
    c = (char)(a + x);
```

```
    c = a + x;
```

```
    z = (float)((int)x * (int)y);
```

```
    z = (float)((int)x * (int)y);
```

```
    z = (float)((int)(x * y));
```

```
    z = x * y;
```

```
    c = char(a) + char(x);
```

```
    c = char(a + int(x));
```

```
    c = char(a + x);
```

```
    c = a + x;
```

```
z = float(int(x) * int(y));  
z = float(int(x) * int(y));  
z = float(int(x * y));  
z = x * y;  
}
```

Observe que não é uma boa regra de programação misturar estas duas formas de conversão. Uma delas deve ser adotada e utilizada ao longo de todo a codificação de um programa.

2.5 Apontadores

Apontadores são extremamente importantes tanto em C quanto em C++. Na prática, é praticamente impossível desenvolver um programa com algum grau de complexidade ou tamanho significativo em uma destas linguagens sem o uso de apontadores.

Nesta seção, particularidades de apontadores em C++ são apresentadas ao longo de uma revisão deste tópico.

2.5.1 Revisão de apontadores

O programa abaixo ilustra um exemplo simples do uso de apontadores. O objetivo deste exemplo é simplesmente rever conceitos básicos relativos a apontadores.

```
#include <iostream.h>  
  
main()  
{  
  int *pt_int;  
  float *pt_float;  
  int pig = 7, dog = 27;  
  float x = 1.2345, y = 32.14;  
  void *general;  
  
  pt_int = &pig;  
  *pt_int += dog;  
  cout << "Pig now has the value of " << *pt_int << "\n";  
  general = pt_int;  
  
  pt_float = &x;  
  y += 5 * (*pt_float);  
}
```

```
cout << "y now has the value of " << y << "\n";
general = pt_float;

const char *name1 = "John";           // Value cannot be changed
char *const name2 = "John";          // Pointer cannot be changed
}
```

Como em ANSI-C, em C++ um apontador é declarado através da presença de um asterisco (*) precedendo o nome da variável. O apontador então é um apontador para uma variável daquele tipo específico e não deve ser usado para variáveis de outros tipos.

O processo de obter o valor de uma variável através de um apontador é usualmente conhecido como *derreferência*. Como em ANSI-C, o modo de derreferenciar um apontador em C++ é também através do uso do asterisco precedendo o nome do apontador.

O resultado da execução do programa deste exemplo deve ser

```
Pig now has the value of 34
y now has the value of 38.3125
```

A penúltima linha desta função ilustra a definição de um apontador para um valor constante, ou seja, o valor para qual o apontador *name1* aponta não pode ser alterado. É também possível definir um apontador constante, o qual não pode ser alterado. Um exemplo deste caso, *name2*, é apresentado na linha seguinte do exemplo.

O uso destas duas construções pode suportar verificação adicional em tempo de compilação e ao mesmo tempo melhorar a qualidade de código. Caso você saiba que um apontador nunca será movido devido a sua natureza, você deve defini-lo como um apontador constante. Caso você saiba que um valor nunca será alterado, ele deve ser definido como constante, de forma que o compilador poderá avisar em caso de tentativas acidentais de mudanças em seu valor.

Este mesmo exemplo ainda ilustra o uso de apontadores para *void*, uma construção já adotada em ANSI-C mas ainda pouco usada. Um apontador para *void* pode receber o valor de qualquer outro tipo de apontador. No exemplo, o apontador *general* recebe primeiro um endereço de um tipo *int* e posteriormente de um tipo *float*. Esta construção permite que o programador use um apontador que pode apontar para muitos tipos diferentes a fim de transferir informação em um programa.

2.5.2 Alocação Dinâmica

Em C, alocação dinâmica é tratada por rotinas de uma biblioteca adicional. Estas rotinas tornaram-se tão utilizadas que sentiu-se, durante o projeto de C++, que sua funcionalidade devia fazer parte da linguagem de maneira a melhorar sua eficiência. Por este motivo, os operadores *new* e *delete* tornaram-se parte integrante de C++, assim como um operador de adição ou atribuição.

O exemplo abaixo ilustra o uso destes operadores.

```
#include <iostream.h>

struct date {
    int month;
    int day;
    int year;
};

main()
{
    int index, *point1, *point2;

    point1 = &index;
    *point1 = 77;
    point2 = new int;
    *point2 = 173;
    cout << "The values are " << index << " " <<
        *point1 << " " << *point2 << "\n";

    point1 = new int;
    point2 = point1;
    *point1 = 999;
    cout << "The values are " << index << " " <<
        *point1 << " " << *point2 << "\n";

    delete point1;

    float *float_point1, *float_point2 = new float;

    float_point1 = new float;
    *float_point2 = 3.14159;
    *float_point1 = 2.4 * (*float_point2);
    delete float_point2;
    delete float_point1;

    date *date_point;
```

```
date_point = new date;
date_point->month = 10;
date_point->day = 18;
date_point->year = 1938;
cout << date_point->month << "/" << date_point->day << "/" <<
                                     date_point->year << "\n";

delete date_point;

char *c_point;

c_point = new char[37];
delete [] c_point;
c_point = new char[sizeof(date) + 133];
delete [] c_point;
}
```

Observe que o operador *new* requer um modificador, o qual deve ser um tipo. Neste exemplo, *point2* aponta para uma variável inteira que foi dinamicamente alocada e pode, a partir daquele momento, ser usada como uma variável normal.

Posteriormente neste exemplo *point1* também passa a apontar para uma variável dinamicamente alocada, e *point2* recebe este mesmo endereço. Observe que neste ponto o endereço a que *point2* originalmente se referia é perdido, de forma que não é possível desalocar *point2*. Por este motivo, o operador *delete* é apenas aplicado a *point1*. Após a aplicação do operador *delete*, o espaço usado pela variável dinâmica torna-se disponível para outras alocações; portanto, os apontadores liberados desta forma não devem ser referenciados depois.

Caso o argumento para o operador *delete* seja `NULL`, o operador nada fará. Efetivamente, este operador só pode ser aplicado a variáveis que tenham sido alocadas com o operador *new*. Se ele for aplicado a qualquer outro tipo de variável, então qualquer coisa pode acontecer — até mesmo um *crash* do sistema, se assim for definido pelo autor do compilador.

O exemplo também mostra o uso destes operadores com outros tipos de variáveis, neste caso *floats* e *structs*.

Finalmente, neste exemplo ilustra-se a utilização destes operadores para alocar e desalocar blocos de tamanho arbitrários. Neste caso, aloca-se um arranjo de 37 caracteres (em geral, 37 bytes) e também um arranjo cujo tamanho é 133 bytes maior que o tamanho da estrutura *date*.

Note que é *muito* importante a indicação, quando for o caso, de que se está removendo um arranjo que foi dinamicamente alocado com *new*. Esta indicação é feita usando-se o modificador `[]` após o operador *delete*. A

ausência do modificador não será acusada como erro nem em tempo de compilação nem em tempo de execução; porém, ela pode comprometer a correta execução de programas mais complexos do que o apresentado neste exemplo. Esta responsabilidade é puramente do programador.

O resultado da execução deste programa é:

```
The values are 77  77  173
The values are 77  999  999
10/18/1938
```

As funções de C para o gerenciamento de memória dinâmica, *malloc()*, *calloc()* e *free()*, podem ainda ser utilizadas em C++. Entretanto, seu uso não deve ser misturado com os novos operadores introduzidos por C++, ou os resultados podem ser imprevisíveis. Novos programas devem utilizar exclusivamente os novos operadores, uma vez que estes são muito mais eficientes.

2.5.3 Apontadores para funções

Apontadores para funções funcionam da mesma forma que em C. Este conceito não é um dos mais utilizados em C, mas é importante em C++ para entender mais adiante os conceitos de ligação dinâmica e polimorfismo. Por este motivo, uma breve revisão é apresentada aqui na forma do seguinte exemplo.

```
#include <stdio.h>

void print_stuff(float data_to_ignore);
void print_message(float list_this_data);
void print_float(float data_to_print);
void (*function_pointer)(float);

main()
{
    float pi = 3.14159;
    float two_pi = 2.0 * pi;

    print_stuff(pi);
    function_pointer = print_stuff;
    function_pointer(pi);
    function_pointer = print_message;
    function_pointer(two_pi);
    function_pointer(13.0);
    function_pointer = print_float;
```

```
    function_pointer(pi);
    print_float(pi);
}

void print_stuff(float data_to_ignore)
{
    printf("This is the print stuff function.\n");
}

void print_message(float list_this_data)
{
    printf("The data to be listed is %f\n", list_this_data);
}

void print_float(float data_to_print)
{
    printf("The data to be printed is %f\n", data_to_print);
}
```

Neste exemplo, *function_pointer* é um apontador para uma função que não retorna nada (tipo de retorno *void*) e requer um único parâmetro formal, uma variável do tipo *float*. As três funções declaradas neste exemplo se encaixam neste padrão, sendo portanto “candidatas” a serem apontadas por *function_pointer*.

É interessante notar neste exemplo que o nome de uma função pode ser atribuído diretamente a um apontador para uma função. Isto sugere, de forma correta, que o nome de uma função é na verdade um apontador para aquela função — apenas este apontador é constante, e portanto não pode ser alterado. Deste modo, após atribuir um valor para um apontador de função, o nome do apontador também pode ser usado para chamar a função.

2.6 Funções

Esta seção discute as melhorias que foram feitas com relação a funções em C++. Estas modificações tornaram a programação mais conveniente e oferece ao compilador melhores mecanismos para verificação de erros.

2.6.1 Protótipos

Um aspecto fundamental em relação a funções C++ é a obrigatoriedade da *prototipação*. Prototipação permite que o compilador realize uma checagem de tipos para chamadas de função em tempo de compilação, de forma que alguns erros de programação podem ser detectados.

Um protótipo é um modelo limitado de algo mais completo que virá depois. Neste caso, o que virá depois é a função completa, e o protótipo dá um modelo da interface para a função que pode ser usado para verificar a corretude do número e tipos de parâmetros para a função. Se o protótipo não estivesse presente, não haveria como verificar se a função está sendo chamada corretamente ou não — nem a tempo de compilação, nem a tempo de ligação; entretanto, o comportamento do programa em tempo de execução pode ser muito estranho caso a chamada não esteja correta.

Uma forma de escrever um protótipo é simplesmente copiar o cabeçalho de uma função para o início do programa onde ela será chamada e acrescentar um `;` no final da linha. Os nomes das variáveis podem opcionalmente estar presentes; neste caso, eles são simplesmente comentários para o programador, sendo ignorados pelo compilador.

2.6.2 Passagem por referência

Passagem por referência é mais uma particularidade de C++, não presente em ANSI-C. O mecanismo de referência já foi mencionado anteriormente, mas em um contexto “não-aconselhável”; aqui, o mecanismo será apresentado na forma em que ele é mais útil.

Passagem por referência permite passar uma variável para uma função e retornar as mudanças feitas pela função para o programa anterior. O mesmo efeito pode ser obtido com apontadores em ANSI-C, mas o mecanismo de referência é mais claro e elegante.

O seguinte exemplo ilustra a combinação do mecanismo de referência e passagem de argumentos para funções.

```
#include <iostream.h>
#include <stdio.h>

void fiddle(int in1, int &in2);

main()
{
    int count = 7, index = 12;
```

```
    cout << "The values are ";
    printf("%3d %3d\n", count, index);

    fiddle(count, index);

    cout << "The values are ";
    printf("%3d %3d\n", count, index);
}

void fiddle(int in1, int &in2)
{
    in1 = in1 + 100;
    in2 = in2 + 100;
    cout << "The values are ";
    printf("%3d %3d\n", in1, in2);
}
```

Observe que o segundo argumento da função *fiddle* é precedido por &. A consequência é que, para este argumento, a variável “verdadeira” do programa anterior será usada; o normal, passagem por valor, é que apenas uma cópia do valor corrente da variável seja passada para a função. Este efeito explica o resultado obtido pela execução deste programa, apresentado a seguir.

```
The values are    7  12
The values are  107 112
The values are    7 112
```

2.6.3 Parâmetros *default*

C++ permite a especificação de valores que argumentos de função assumirão caso a função seja chamada sem estes parâmetros — são os *parâmetros default*.

O exemplo abaixo ilustra o uso deste mecanismo em uma função com três argumentos, dois dos quais podem assumir valores *default*, mas pelo menos o primeiro deve estar sempre presente. No corpo da função, há chamadas para a função com um, dois e três argumentos.

```
#include <iostream.h>
#include <stdio.h>

int get_volume(int length, int width = 2, int height = 3);

main()
```

```
{
int x = 10, y = 12, z = 15;

    cout << "Some box data is " << get_volume(x, y, z) << "\n";
    cout << "Some box data is " << get_volume(x, y) << "\n";
    cout << "Some box data is " << get_volume(x) << "\n";

    cout << "Some box data is ";
    cout << get_volume(x, 7) << "\n";
    cout << "Some box data is ";
    cout << get_volume(5, 5, 5) << "\n";

}

int get_volume(int length, int width, int height)
{
    printf("%4d %4d %4d ", length, width, height);
    return length * width * height;
}
```

O resultado da execução deste programa deve ser:

Some box data is	10	12	15	1800
Some box data is	10	12	3	360
Some box data is	10	2	3	60
Some box data is	10	7	3	210
Some box data is	5	5	5	125

Em alguns compiladores, a saída pode surgir “embaralhada” nas três primeiras linhas. Quando isto acontece, é porque a rotina de apresentação de dados na função *get_volume* é executada antes da apresentação dos dados na rotina *main*.

2.6.4 Número variável de argumentos

Em algumas raras ocasiões, pode ser desejável definir uma função com um número variável de argumentos — um bom exemplo deste tipo de necessidade é a função *printf*. Como em ANSI-C, C++ pode usar macros definidas em *stdarg.h* para atingir este objetivo. Entretanto, como C++ é mais rígido com relação à prototipagem de funções, é preciso um mecanismo que indique esta liberdade no número de argumentos. Este mecanismo é o uso de três pontos ... no protótipo da função.

O exemplo a seguir ilustra o uso deste mecanismo.

```
#include <iostream.h>
#include <stdarg.h>

// Declare a function with one required parameter
void display_var(int number, ...);

main()
{
    int index = 5;
    int one = 1, two = 2;

    display_var(one, index);
    display_var(3, index, index + two, index + one);
    display_var(two, 7, 3);
}

void display_var(int number, ...)
{
    va_list param_pt;

    va_start(param_pt, number);                // Call the setup macro

    cout << "The parameters are ";
    for (int index = 0 ; index < number ; index++)
        cout << va_arg(param_pt, int) << " ";    // Extract a parameter
    cout << "\n";
    va_end(param_pt);                          // Closing macro
}
```

Neste exemplo, o protótipo indica que um argumento do tipo *int* é requerido na chamada da função; após este argumento, o compilador não faz mais nenhuma verificação de tipos de dados sendo passado para a função. É tarefa do programador garantir que os dados sendo passados são do tipo esperado.

O resultado da execução deste exemplo simples é apresentado abaixo.

```
The parameters are 5
The parameters are 5 7 6
The parameters are 7 3
```

Este mecanismo, apesar de presente por questões de completude da linguagem, deve ser evitado. Seu uso pode levar à obscuridade de código, tornando entendimento e manutenção tarefas mais difíceis.

2.6.5 Sobrecarga de nome de funções

Em ANSI-C, duas funções devem ter nomes obrigatoriamente únicos. Em C++, uma vez que o mecanismo de prototipagem permite identificar uma função por seu nome e seus argumentos, é possível repetir nome de funções — é o mecanismo de *sobrecarga*. Este mecanismo é frequentemente utilizado na programação orientada a objetos.

O exemplo abaixo ilustra o uso deste mecanismo. Neste caso, três funções são definidas com o mesmo nome. No momento da chamada, o compilador seleciona a função correta com base em seu nome e no número de e tipos dos parâmetros formais.

```
#include <iostream.h>

overload do_stuff;                                // This is optional

int do_stuff(const int);                          // This squares an integer
int do_stuff(float);                              // This triples a float & returns int
float do_stuff(const float, float);               // This averages two floats

main()
{
    int index = 12;
    float length = 14.33;
    float height = 34.33;

    cout << "12 squared is " << do_stuff(index) << "\n";
    cout << "24 squared is " << do_stuff(2 * index) << "\n";
    cout << "Three lengths is " << do_stuff(length) << "\n";
    cout << "Three heights is " << do_stuff(height) << "\n";
    cout << "The average is " << do_stuff(length,height) << "\n";
}

int do_stuff(const int in_value)                  // This squares an integer
{
    return in_value * in_value;
}

int do_stuff(float in_value)                      // Triples a float & return int
{
    return (int)(3.0 * in_value);
}
```

```
float do_stuff(const float in1, float in2)    // This averages two floats
{
    return (in1 + in2)/2.0;
}
```

O resultado da execução deste programa deve ser:

```
12 squared is 144
24 squared is 576
Three lengths is 42
Three heights is 102
The average is 24.330002
```

Observe que o tipo de retorno da função não é um critério de seleção para determinar qual função será chamada.

A palavra chave *overload* dá uma indicação que o nome a seguir, neste caso *do_stuff*, será conscientemente sobrecarregado, e que isto não está acontecendo por engano. Entretanto, esta palavra chave era apenas obrigatória em versões mais antigas de C++; correntemente, este mecanismo é tão utilizado que usar sobrecarga é a situação padrão, não sendo necessário indicar mais seu uso. A opção de uso desta palavra é mantido por questões de compatibilidade com códigos de versões anteriores.

Deve-se ainda observar que a seleção da função é feita em tempo de compilação, e não em tempo de execução. Desta forma, não há prejuízo no desempenho do programa ao se utilizar este mecanismo. Mesmo que cada uma destas funções tivesse um nome único não haveria diferença no tamanho do programa ou em seu tempo de execução.

Capítulo 3

Encapsulação

Este capítulo apresenta o início das verdadeiras técnicas de programação orientada a objetos em C++. O tópico abordado aqui é *encapsulação*, que é basicamente uma técnica na linha de “dividir para conquistar.”

Encapsulação é o processo no qual se baseia a formação de objetos. Um objeto encapsulado é muito similar ao conceito de Tipos Abstratos de Dados, suportado em outras linguagens. Encapsulação envolve o uso de classes, e sem encapsulação não há programação orientada a objetos. Há mais em programação orientada a objetos do que encapsulação, mas este é um princípio fundamental.

A idéia por trás de encapsulação é proteger a informação — código, dados — por trás de uma barreira, de forma a evitar que programadores usando aquele objeto venham porventura corromper esta informação devido a pequenos enganos a que todos estão sujeitos. Este é o mecanismo conhecido como *ocultamento da informação*¹. Outra contribuição é a possibilidade de isolar erros em seções pequenas de código, o que facilita encontrá-los e corrigí-los.

3.1 Classes e Encapsulação

A definição de classes já foi brevemente discutida na Seção 2.4. Naquela seção foi apresentada uma classe que era funcionalmente equivalente a uma estrutura. Nesta seção, aspectos de encapsulação na definição de classes serão abordados.

¹Information hiding.

3.1.1 Definição de classes

Uma definição de classe é basicamente uma definição de um tipo de dado. Uma classe contém um conjunto de bits representando um estado e um conjunto de operações que permitem modificar o estado. Em geral, as operações são *públicas* e os dados internos da classe são *privados* — as únicas modificações possíveis nos dados são realizadas através das operações que a classe deixa disponível para que o público use.

A informação que permanece escondida do usuário da classe está contida na *seção privativa* da classe. De fora da classe, é como se aqueles dados não existissem: eles não podem ser acessados ou modificados diretamente.

É possível que dados internos sejam acessados externamente; para tanto, deve-se colocar estes dados na *seção pública* da classe. (Este foi o mecanismo adotado no exemplo da seção 2.4.) A seção pública inicia-se com o rótulo *public*, seguida pelas declarações (dados ou funções) que poderão ser acessadas externamente.

Funções declaradas dentro de uma classe são chamadas de *funções membros*, uma vez que elas são membros da classe da mesma forma que os dados declarados na classe. Funções membros são definidas da mesma forma que funções normais — a única diferença visível é que o nome da função é precedido pelo nome da classe, sendo os nomes separados por `::`. O uso do nome da classe como prefixo para o nome da função permite que outras classes usem funções membros com o mesmo nome.

Uma outra diferença existe com relação a funções membros, sendo esta mais significativa: dados privados da classe podem ser acessados por elas, tanto para leitura como para alteração. A implementação de uma função membro de uma classe pode fazer o que quiser com os dados privados da classe, mas nada com os dados privados de outras classes. Este é mais um motivo pelo qual o nome da classe deve preceder o nome da função membro na definição.

Em geral, a parte privativa da classe contém apenas dados, enquanto que a parte pública contém apenas declarações de funções. Entretanto, nada impede que dados sejam declarados também na parte pública e funções na parte privativa.

O exemplo a seguir apresenta, de forma muito simplificada, os conceitos apresentados acima. Neste programa, uma classe contendo um único dado — o inteiro *data_store* — é declarada, e todo acesso ao estado da classe (o valor deste inteiro) é restrito a suas duas funções membro, *set* e *get_value*. Estas duas funções são os *métodos* da classe, na terminologia de orientação a objetos apresentada no Capítulo 1.

```
#include <iostream.h>
```

```
class one_datum {
    int data_store;
public:
    void set(int in_value);
    int get_value(void);
};

void one_datum::set(int in_value)
{
    data_store = in_value;
}

int one_datum::get_value(void)
{
    return data_store;
}

main()
{
    one_datum dog1, dog2, dog3;
    int piggy;

    dog1.set(12);
    dog2.set(17);
    dog3.set(-13);
    piggy = 123;

    // dog1.data_store = 115;      This is illegal in C++
    // dog2.data_store = 211;      This is illegal in C++

    cout << "The value of dog1 is " << dog1.get_value() << "\n";
    cout << "The value of dog2 is " << dog2.get_value() << "\n";
    cout << "The value of dog3 is " << dog3.get_value() << "\n";
    cout << "The value of piggy is " << piggy << "\n";
}
```

O resultado da execução deste programa é

```
The value of dog1 is 12
The value of dog2 is 17
The value of dog3 is -13
The value of piggy is 123
```

Observe que, dentro da definição da classe, o que está presente na verdade são os protótipos dos métodos. Esta é mais uma razão pela qual protótipos são importantes em C++.

No início da função *main* é mostrado como objetos da classe definida são criados — da mesma forma que variáveis dos tipos padrão da linguagem. O processo de chamar de um dos métodos da classe aplicado a um objeto da classe é conhecido na terminologia de orientação a objetos como *enviar uma mensagem*. A maneira de se enviar uma mensagem em C++ é ilustrada neste exemplo. Por exemplo, a linha onde se lê `dog1.set(12)` pode ser interpretada como “envie uma mensagem para o objeto `dog1` instruindo-o para setar (seu valor interno para) 12.” Repare como esta forma de acesso assemelha-se sintaticamente ao acesso de dados internos de uma estrutura: o nome do objeto, um ponto (`.`), e o nome do método.

O exemplo também ilustra, sob a forma de comentários, a “forma ilegal” de acesso aos dados internos dos objetos. Caso se tentasse compilar um programa com um comando daquela forma, um erro seria acusado pelo compilador C++. Observe ainda que não é possível “misturar” os dados de dois objetos — cada objeto tem seus dados exclusivos, e mesmo que eles sejam de uma mesma classe não há como misturá-los.

3.1.2 Ocultamento da informação

A fim de entender realmente os benefícios associados com a encapsulação, iremos analisar a seguir um exemplo onde encapsulação não é utilizada, e possíveis problemas que podem ocorrer neste caso. Posteriormente, este problema será revisitado para analisar uma solução com encapsulação.

Neste problema, duas entidades são consideradas. A primeira delas é uma entidade do tipo retângulo, que tem uma altura e uma largura. A área do retângulo pode ser calculada como o produto destes dois valores. A segunda entidade é uma entidade do tipo mastro, que tem um comprimento e uma profundidade (o quanto o mastro foi enterrado); o comprimento total do mastro é a soma destes dois valores.

A solução usando ANSI-C seria representar cada uma das entidades como uma estrutura. Esta solução é apresentada a seguir.

```
#include <iostream.h>

int area(int rec_height, int rec_width);

struct rectangle {
    int height;
```

```
    int width;
};

struct pole {
    int length;
    int depth;
};

int area(int rec_height, int rec_width)           //Area of a rectangle
{
    return rec_height * rec_width;
}

main()
{
    rectangle box, square;
    pole flag_pole;

    box.height = 12;
    box.width = 10;
    square.height = square.width = 8;

    flag_pole.length = 50;
    flag_pole.depth = 6;

    cout << "The area of the box is " <<
            area(box.height, box.width) << "\n";
    cout << "The area of the square is " <<
            area(square.height, square.width) << "\n";
    cout << "The funny area is " <<
            area(square.height, box.width) << "\n";
    cout << "The bad area is " <<
            area(square.height, flag_pole.depth) << "\n";
}
```

Não há nada errado com esta forma de resolver o problema — o perigo está nas coisas tolas que podem ser feitas com estes dados. Por exemplo, para calcular uma das áreas está se multiplicando a altura de um quadrado com o comprimento de de uma caixa diferente. Em outro caso, ainda pior: calcula-se o produto da altura do quadrado com a profundidade de um mastro, seja lá o que isto quer dizer. Os resultados da execução do programa, no entanto,

parecem perfeitamente válidos:

```
The area of the box is 120
The area of the square is 64
The funny area is 80
The bad area is 48
```

Embora esta situação possa parecer tola quando observada neste pequeno exemplo, ela pode ocorrer com frequência em sistemas de porte maior. Não há modos de prevenir que isto aconteça em ANSI-C. A solução real para este problema está na encapsulação.

O exemplo a seguir ilustra a utilização de classes para modelar o mesmo problema. Retângulo é agora modelado como uma classe. Neste caso torna-se evidente que o cálculo de área é algo que só faz sentido (neste caso) para retângulos, pois o método correspondente é uma função membro desta classe. Mastro ainda é modelado como uma estrutura apenas para ilustrar que os dois tipos de construções podem coexistir sem problemas em um programa.

```
#include <iostream.h>

class rectangle {                                // A simple class
    int height;
    int width;
public:
    int area(void);                               // with two methods
    void initialize(int, int);
};

int rectangle::area(void)                        //Area of a rectangle
{
    return height * width;
}

void rectangle::initialize(int init_height, int init_width)
{
    height = init_height;
    width = init_width;
}

struct pole {
    int length;
    int depth;
};
```

```
main()
{
    rectangle box, square;
    pole flag_pole;

    box.initialize(12, 10);
    square.initialize(8, 8);

    flag_pole.length = 50;
    flag_pole.depth = 6;

    cout << "The area of the box is " <<
           box.area() << "\n";
    cout << "The area of the square is " <<
           square.area() << "\n";
}
```

O resultado da execução deste programa é

```
The area of the box is 120
The area of the square is 64
```

Observe que não há modo, neste caso, de cometer os enganos que ocorriam no exemplo anterior — simplesmente é impossível acessar diretamente os dados de objetos individuais da classe retângulo e, conseqüentemente, é impossível misturar a informação que eles carregam.

Esta técnica não é exatamente inédita. Sistemas operacionais usualmente adotam o conceito de *kernel*, um conjunto de serviços que pode apenas ser acessado através de rotinas pré-estabelecidas (*system calls*). A manipulação de arquivos é um exemplo de um destes serviços. A fim de manipular arquivos, o sistema operacional mantém um grande conjunto de dados. Entretanto, o usuário nem sabe da existência destes dados — tudo o que ele pode fazer é requisitar os serviços através do system call adequado. Isto evita que o usuário, inadvertidamente, corrompa o sistema de arquivos.

A definição de classes permite que o programador use este mesmo mecanismo na implementação de uma aplicação. No exemplo anterior, a definição da classe retângulo e de seus métodos poderia ter sido desenvolvida à parte, de forma totalmente isolada do programa que a utiliza. Uma vez que esta classe esteja desenvolvida e “aprovada”, ela pode tornar-se disponível para ser utilizada na aplicação. Observe como a utilização de classes promove de modo natural a técnica de dividir para conquistar. Em grandes projetos, esta

abordagem é repetidamente utilizada, sendo que algumas classes podem ser extremamente complexas internamente — mas para o usuário da classe esta complexidade é totalmente abstraída.

3.1.3 Funções em linha

Uma função *em linha*² é uma função cujo código é inserido no código do usuário. Conceitualmente, tais funções são equivalentes às macros em ANSI-C, e elas podem melhorar desempenho ao evitar a carga extra associada com a chamada de uma função. Na prática, funções em linha são melhores que macros pois evitam os famigerados erros de expansão: com funções em linha, cada argumento é avaliado apenas uma vez — da mesma forma que ocorre com funções normais. Outra vantagem da função em linha é que ela permite a verificação dos tipos de argumentos, algo que é impossível com macros.

Há duas formas básicas de se definir funções em linha:

1. através do uso da palavra chave *inline* na definição da função membro; ou
2. incluindo o código da função membro dentro da classe.

Um cuidado que deve ser tomado é evitar a definição de um número excessivo de funções em linha, o que pode ter um efeito negativo no desempenho da aplicação, principalmente em ambientes com paginação (a maior parte dos processadores modernos). Um número exato não pode ser especificado, pois isto depende de cada configuração.

3.1.4 Construtores e Destrutores

Construtores são basicamente funções de inicialização de uma classe, as quais são invocadas no momento em que objetos desta classe são criadas. Eles permitem inicializar campos internos da classe e alocar recursos que um objeto da classe possa demandar, tais como memória, arquivos, semáforos, soquetes, etc.

A função construtor de uma classe é reconhecida por ter o mesmo nome da classe. Por exemplo, um construtor para uma classe chamada *rectangle* seria a função *rectangle()*. É possível definir-se mais de um construtor; em outras palavras, a função membro construtor, assim como qualquer outra função membro de uma classe, pode ser sobrecarregada.

Construtores podem ser usados para suportar a inicialização de valores internos da classe durante a declaração de objetos. Neste caso, os argumentos

²Inline.

da função construtor são os valores que deverão ser inicializados para os dados do objeto. Quando este recurso for utilizado, é importante observar que construtores devem ser definidos de forma a cobrir todos as formas de inicialização desejadas — por exemplo, um construtor sem argumentos para objetos não inicializados na declaração, ou que assumam valores default.

Destrutores realizam a função inversa: são funções invocadas quando um objeto está para “morrer”. Caso um objeto tenha recursos alocados, destrutores devem liberar tais recursos. Por exemplo, se o construtor de uma classe alocou uma variável dinamicamente com *new*, o destrutor correspondente deve liberar o espaço ocupado por esta variável com o operador *delete*.

A função destrutor de uma classe tem o mesmo nome da classe com um til (~) como prefixo. Por exemplo, um destrutor para a classe *rectangle* seria chamado *~rectangle()*.

Quando um objeto é usado para inicializar outro, uma cópia bit a bit é feita. Em muitos casos, este mecanismo de cópia é perfeitamente adequado. Porém, há casos onde esta cópia simples pode ser perigosa; por exemplo, quando um objeto contém apontadores para áreas alocadas dinamicamente. Em tais casos, a cópia implicaria em um compartilhamento de dados que poderia não ter exatamente o efeito desejado.

Para tais situações, C++ permite a definição de um construtor de cópia. Quando um construtor de cópia existe, a cópia por bits não é utilizada. A forma geral de declarar um construtor de cópia é

nome-classe (**const** *nome-classe* &*obj*) { ... };

Onde *obj* é uma referência para o objeto do lado direito da inicialização. A forma de uso é, assumindo que exista uma classe *rectangle*,

rectangle x = y;

ou

rectangle x(y);

Em qualquer caso, uma referência para *y* seria passada para *obj*.

Observe que o construtor de cópia só é chamado automaticamente para inicializações. Um comando de atribuição no meio de uma função não ativa o construtor de cópia — se este for o efeito desejado, o operador = deve ser sobrecarregado.

3.1.5 Objetos e funções

Uma vez que objetos são equivalentes a qualquer dado de outro tipo, é possível passar objetos como argumentos de uma função. Objetos são passados para função por valor, ou seja, uma cópia do objeto é feita quando ele é passado para a função. Em outras palavras, um novo objeto é criado.

Da mesma forma, uma função pode ter como tipo de retorno uma classe, e assim retornar um objeto para a função anterior. O valor de retorno de uma função também pode ser uma referência, o que permite que uma função seja utilizada do lado esquerdo de uma atribuição. O principal uso para esta construção é na sobrecarga de operadores, permitindo por exemplo comandos tais como $a = b = c$, onde a , b e c podem ser objetos.

Quando um objeto é passado como um argumento para uma função, uma cópia do objeto é feita. Quando esta cópia é feita, o construtor do objeto *não* é chamado — a cópia é feita bit a bit. Entretanto, quando a função termina, a cópia é um objeto que deixará de existir e, portanto, o destrutor para o objeto será chamado. Quando este comportamento puder trazer problemas, o objeto deve ser passado por referência.

3.2 Arranjos e apontadores

Da mesma forma que para dados de outros tipos, objetos podem ser agrupados em arranjos e apontadores para objetos podem ser definidos.

3.2.1 Arranjos

A sintaxe para a declaração de arranjos de objetos é exatamente a mesma que é usada para arranjos de outros tipos de dados. Por exemplo, se *rectangle* é uma classe já definida, então um arranjo chamado *boxes* de 5 objetos desta classe seria definido como

rectangle box[5];

A ativação de um método *area* para o terceiro elemento deste arranjo seria invocado na forma

box[2].area();

Da mesma forma que para objetos simples, arranjos de objetos podem ser inicializados durante a declaração se um construtor adequado foi definido.

3.2.2 Apontadores

Apontadores para objetos também podem ser definidos exatamente da mesma forma que apontadores para variáveis de outros tipos de dados. O acesso a membros da classe (por exemplo, para invocar um método) é realizado a partir do apontador usando-se o operador *seta* (->) ao invés do operador ponto. Por exemplo, um apontador para um objeto da classe *rectangle* pode ser definido como em

```
rectangle *p;
```

e, assumindo-se que um objeto *square* desta classe exista, *p* poderia apontar para ele após o comando

```
p = &square;
```

O método *area* poderia ser ativado a partir de *p* na forma

```
p->area();
```

3.2.3 O apontador *this*

Quando uma função membro é chamada, ela automaticamente recebe como argumento um apontador para o objeto que ativou o método; este apontador é chamado *this*. Quando uma função membro de uma classe acessa diretamente um dado privativo de um objeto, na verdade este apontador está sendo implicitamente utilizado para identificar o objeto.

Algumas vezes, pode ser necessário ter que explicitar o uso deste apontador — por exemplo, quando se espera que a função membro retorne um objeto. Este tipo de situação será visto em mais detalhes quando o assunto de sobrecarga de operadores for abordado.

3.2.4 Alocação dinâmica de objetos

Como outros tipos de dados, objetos podem ser dinamicamente alocados usando os operadores *new* e *delete*. Por exemplo, para alocar um arranjo de seis objetos da classe *rectangle*, a seguinte construção poderia ser utilizada:

```
rectangle *p;                // pointer declaration
p = new rectangle [6];        // objects allocation
...                            // rest of program
delete [ ] p;                 // objects deallocation
```

3.3 Sobrecarga de operadores

Já foi discutido no capítulo anterior a questão da sobrecarga de nomes de funções. As mesmas regras se aplicam quando as funções sobrecarregadas são membros de uma classe — mesmo no caso de construtores.

Com relação a classes, há um aspecto muito relacionado à sobrecarga de funções — é a sobrecarga de operadores. Em outras palavras, é possível definir os símbolos usuais de operadores — tais como `+`, `*` ou `<<` — de forma a associá-los com métodos definidos para a classe.

A forma de implementar esta característica é através da definição de uma *função operador*. Uma função operador é geralmente uma função membro da classe. (Caso não o seja, ela deve ser uma função amiga, como explicado na próxima seção.) A declaração de uma função operador é da forma

tipo-de-retorno nome-da-classe:: operador# (lista-arg);

onde `#` é substituído pelo operador que irá ser sobrecarregado.

Por exemplo, suponha que uma classe de nome *loc* seja definida com dois dados privativos, *longitude* e *latitude*. Um operador de adição para esta classe poderia ser definido como

```
loc loc::operator+(loc op2) {  
    loc temp;  
  
    temp.longitude = op2.longitude + longitude;  
    temp.latitude = op2.latitude + latitude;  
    return temp;  
};
```

A forma de uso para este operador seria então

```
loc loc1, loc2, newloc;  
...  
newloc = loc1 + loc2;
```

Observe que *operator+()* tem apenas um parâmetro, mesmo sobrecarregando um operador binário (`+`). O que ocorre é que o parâmetro do lado esquerdo do sinal `+` é passado implicitamente para a função operador usando o apontador *this*. O operando do lado direito é passado como o parâmetro *op2*. O fato de que o operando do lado esquerdo é passado usando *this* tem uma implicação importante: quando operadores binários são sobrecarregados, é o objeto à esquerda que gera a chamada para a função operador.

É importante notar que novos operadores não podem ser criados — apenas aqueles que existem podem ser sobrecarregados por uma classe. Do mesmo modo, a precedência de um operador não pode ser modificada.

3.4 Amizade

Amizade é um mecanismo de C++ que permite que uma classe autorize acesso de sua parte privativa a outra classe ou função. A palavra chave *friend* é usada para indicar a autorização de acesso.

Uma função amiga tem tanto privilégio de acesso aos dados de uma classe quanto uma função membro. A maior diferença entre elas é que uma função amiga é usualmente invocada na forma $f(x)$, enquanto que uma função membro é invocada na forma $x.f()$. Sempre que possível, funções membros devem ser preferidas sobre funções amigas, devendo estas últimas ser reservadas para casos de necessidade ou para clarificar a sintaxe de uso das funções e operadores.

Classes amigas, similarmente, permitem o livre acesso a seus dados privativos entre si. Duas classes que cooperem de perto para a resolução de um problema muitas vezes precisam ser declaradas como amigas. Há casos em que as classes não são combinadas em uma única classe porque têm números distintos de objetos ou tempo de duração diferente; vistas “de fora”, elas poderiam parecer como uma única classe. Em tais casos, amizade pode ser útil.

Há muita discussão na comunidade de C++ com relação ao uso de amizade — se ela fere encapsulação, se deveria ser utilizada ou não. Bom senso é a recomendação neste caso. Já foi visto que esconder informação é uma qualidade. No entanto, se para evitar o uso de funções ou classes amigas é necessário tornar público um dado da classe — seja através da colocação do dado na seção pública, seja através da inclusão de funções públicas da forma *get* e *set* (onde tais funções não fazem sentido para um usuário, mas apenas para a classe que irá utilizá-las) — então certamente é preferível usar o mecanismo de amizade.

3.5 Tratamento de exceções

Em linguagens de programação tradicionais, assim como C, o tratamento de erros é geralmente realizado na forma de códigos de falha retornados por funções. Um exemplo típico desta abordagem são as indicações geralmente retornadas por *system calls*: por exemplo, quando se executa a rotina do sistema operacional para abrir um arquivo, *open*, um valor de retorno positivo indica sucesso da operação, enquanto que um valor negativo (-1) indica uma condição de erro (arquivo especificado não existe, usuário não tem permissão de acesso, sistema não consegue abrir mais arquivos). Todas as possíveis condições de erro retornadas por uma rotina do sistema devem ser testadas

— a não detecção de um erro no momento do retorno da rotina pode ser catastrófica para a aplicação e, em alguns casos, para o sistema como um todo.

De certo modo, mecanismo usual de manipulação de erros fere o conceito de encapsulação de C++. Um erro, o qual pode ter ocorrido em um nível bem interno da aplicação, tem que ser propagado até um ponto onde uma decisão de o que fazer possa ser tomada — muitas vezes, bem longe da classe responsável pelo erro. É parte do princípio de encapsulação e de boa programação que cada classe seja capaz de lidar com seus erros. O efeito de um erro deve estar restrito à classe que o gerou, e não se propagar pelo sistema “derrubando” outras classes em um efeito dominó. Duas características de um sistema robusto é que todas as falhas do sistema devem ser relatadas e, caso a falha seja tão séria que comprometa a continuação da execução da aplicação, o processo da aplicação deve ser terminado.

O mecanismo que C++ oferece para lidar com a manipulação de situações como esta é o *tratamento de exceções*³. As construções suportadas para este fim são *try*, *throw* e *catch*.

Um bloco *try* delimita um segmento do programa onde alguma coisa de errado pode acontecer. Se algum erro acontece durante a execução deste bloco, o comando *throw* (similar a um *return*) é chamado com um argumento que pode ser qualquer objeto; o mais comum é retornar com um inteiro. O valor enviado por *throw* é capturado por um (ou mais de um) bloco *catch*, na mesma rotina que contém o bloco *try*. No corpo do bloco *catch* se define a ação a ser tomada em caso de ocorrência de erros.

O exemplo a seguir ilustra a utilização deste mecanismo.

```
// rotina onde erros serao tratados
int f() {
    // bloco onde erro pode ocorrer
    try {
        return g();
    }

    // excecoes, se houver, sao tratadas aqui
    catch (int x) {
        // trata excecao...
    }

    catch (float x) {
    }
}
```

// este e o bloco try

// aqui, se um inteiro foi atirado

// ou aqui, se um float foi atirado

³Exception handling. A proposta de incorporação deste mecanismo à linguagem é relativamente recente, não sendo ainda suportada por todos compiladores.

```
        // trata execucao...
    }

    // se algo diferente foi atirado, sera atirado para a
    // a rotina que chamou f()
}

// rotina onde erro pode ocorrer
int g() {
    if (Erro_Que_Eu_Nao_Sei_Tratar_Ocorreu) {
        throw 2;                                // atira um inteiro
    }

    // continua, se tudo bem...
}
```

Capítulo 4

Herança

O mecanismo de herança é o que diferencia a programação orientada a objetos da programação com Tipos Abstratos de Dados. Desta forma, este é um dos conceitos mais importantes para a efetiva utilização de C++.

O conceito já foi introduzido no Capítulo 1 deste texto, sendo lá apresentado como um mecanismo de especificação de sistemas. E é justamente neste ponto que reside a elegância da programação orientada a objetos: o mesmo mecanismo que é empregado como uma ferramenta natural de especificação (*carro é-um-tipo-de veículo, roda é-uma-parte-de veículo*) é também utilizado para a implementação. Esta facilidade de mapear conceitos do mundo real para construções da implementação é devido à redução da *distância semântica*¹ entre os modelos adotados em cada fase.

4.1 Funcionamento básico

Para ilustrar como o mecanismo de herança é implementado em C++, considere o seguinte exemplo, onde uma classe para representar *veículo* é definida.

```
class vehicle {  
  protected:  
    int wheels;  
    float weight;  
  public:  
    void initialize(int in_wheels, float in_weight);  
    int get_wheels(void);  
    float get_weight(void);  
    float wheel_loading(void);  
}
```

¹Semantic gap


```
};
```

Esta classe consiste de quatro métodos que manipulam os dois atributos relacionados ao número de rodas e peso do veículo. (A função do rótulo *protected* será discutida em mais detalhes na próxima seção.) Observe que esta definição é genérica o suficiente para representar estes aspectos tanto para uma bicicleta quanto para um avião a jato. Esta é uma característica de classes base — elas *abstraem* as propriedades de um grupo de classes.

A questão agora é: como representar (por exemplo) carros e caminhões? Eles são tipos de veículos, e uma declaração de classe que simplesmente repetisse as propriedades de veículos para carros ou caminhões iria simplesmente perder esta informação. Pior ainda: se a classe de veículos fosse atualizada para introduzir uma nova propriedade (atributo ou método), esta mudança não seria refletida nas classes de carros ou caminhões. Este tipo de mudança ocorre muitas vezes em projetos de grande porte, quando há revisões de especificação para partes do projeto; em geral, é muito difícil controlar a propagação das modificações, o que geralmente envolve diversos grupos de trabalho.

Em C++, como em outras linguagens de programação orientadas a objetos, há um mecanismo para conectar a definição da classe derivada com a classe base. Observe abaixo como a classe para carros é definida, assumindo-se que a classe *vehicle* foi definida como acima.

```
class car : public vehicle {  
    int passenger_load;  
public:  
    void initialize(int in_wheels, float in_weight, int people = 4);  
    int passengers(void);  
};
```

O mecanismo para indicar que a classe *car* é derivada da classe *vehicle* é dado pela sintaxe : `public`, como em

```
class Derived : public Base { ...};
```

que poderia ser lido como *Derived é-um-tipo-de Base*. Isto indica que a classe *Derived* é composta por toda a informação que está contida na classe *Base*, além de sua própria informação.

Na definição da classe *car*, indica-se que carro tem a mesma informação que um veículo, além de informação adicional sobre o número de passageiros que ele comporta. Note também que a classe *car* define um método de nome

initialize, que era também um método da classe *vehicle*. Desta forma, quando *initialize* for ativado para um carro, a nova versão é que será ativada — para outros veículos, a antiga versão continua valendo. Entretanto, esta forma de sobreposição de métodos não é aconselhada — veja o Capítulo 5 para mais detalhes sobre este assunto.

Observe dos exemplos dados acima que não há nada de especial em relação a uma classe base: ela é simplesmente uma classe como qualquer outra, sem nenhuma indicação especial de que seus métodos e atributos serão herdados por outras classes. Da mesma forma, seria possível usar uma classe derivada como base para outras classes — ou seja, toda uma hierarquia de classes pode ser implementada.

Similarmente, uma classe para representar caminhões poderia ser derivada de veículos como em

```
class truck : public vehicle {
    int passenger_load;
    float payload;
public:
    void init_truck(int how_many = 2, float max_load = 24000.0);
    float efficiency(void);
    int passengers(void);
};
```

Observe que as características associadas a caminhões são diferentes daquelas de carros, e por este motivo é necessário definir uma nova classe derivada. É interessante ressaltar que não há nenhum relacionamento direto entre *car* e *truck* — elas são apenas classes que foram derivadas de uma mesma classe base.

O exemplo abaixo ilustra a utilização destas classes em um programa C++.

```
#include <iostream.h>
// ... definitions for classes vehicle, car, and truck here.

main()
{
    vehicle unicycle;

    unicycle.initialize(1, 12.5);
    cout << "The unicycle has " <<
        unicycle.get_wheels() << " wheel.\n";
    cout << "The unicycle's wheel loading is " <<
        unicycle.wheel_loading() << " pounds on the single tire.\n";
```

```
    cout << "The unicycle weighs " <<
           unicycle.get_weight() << " pounds.\n\n";

car sedan;

sedan.initialize(4, 3500.0, 5);
cout << "The sedan carries " << sedan.passengers() <<
      " passengers.\n";
cout << "The sedan weighs " << sedan.get_weight() << " pounds.\n";
cout << "The sedan's wheel loading is " <<
      sedan.wheel_loading() << " pounds per tire.\n\n";

truck semi;

semi.initialize(18, 12500.0);
semi.init_truck(1, 33675.0);
cout << "The semi weighs " << semi.get_weight() << " pounds.\n";
cout << "The semi's efficiency is " <<
      100.0 * semi.efficiency() << " percent.\n";
}
```

O resultado da execução deste programa é:

```
The unicycle has 1 wheel.
The unicycle's wheel loading is 12.5 pounds on the single tire
The unicycle weighs 12.5 pounds.
```

```
The sedan carries 5 passengers.
The sedan weighs 3500 pounds.
The sedan's wheel loading is 875 pounds per tire.
```

```
The semi weighs 12500 pounds.
The semi's efficiency is 72.929072 percent.
```

É interessante que se observe que um apontador declarado para a classe base (*vehicle*, neste exemplo) poderia ser usado para apontar para objetos tanto da classe base quanto para objetos das classes derivadas (*car* e *truck*, neste caso). Isto concorda com a intuição sobre hereditariedade — afinal de contas, um carro continua sendo um veículo, e ao apontar para um carro está se apontando para um veículo.

4.2 Controle de acesso

Quando uma classe é definida, seus membros (dados ou métodos) são colocados em seções internas cujo acesso pode ser público, privativo ou protegido. Seções de acesso público (precedidas pelo rótulo *public*) ou privativo (precedidas pelo rótulo *private*) já foram discutidas no capítulo anterior. Acesso a membros protegidos (em seção da declaração precedida pelo rótulo *protected*) é similar ao acesso a membros privativos, exceto para o caso acesso por parte de classes derivadas.

A forma geral de especificar derivação de classes é

```
class Derived : access-specifier base { ... };
```

Quando uma classe herda outra, todos os membros da classe base tornam-se também membros da classe derivada. O tipo de acesso da classe base de dentro da classe derivada é determinado pelo *especificador de acesso*. Este especificador de acesso pode ser *public*, *private* ou *protected*. Qualquer que seja o especificador de acesso, membros privativos da classe base continuam sendo propriedade privada da classe base, e não podem ser acessados por classes derivadas.

Quando o especificador de acesso é *public*, como especificado nos exemplos da seção anterior, então todos os membros públicos da classe base tornam-se membros públicos da classe derivada, e os membros protegidos da classe base tornam-se membros protegidos da classe derivada. (Esta é a diferença que existe entre membros privativos e membros protegidos.)

Quando o especificador de acesso é *private*, todos os membros públicos e protegidos da classe base tornam-se membros privativos da classe derivada — isto é, a classe derivada não pode transferir seus privilégios de acesso a outras classes.

Quando o especificador de acesso é *protected*, todos os membros públicos e protegidos da classe base tornam-se membros protegidos da classe derivada.

4.3 Herança múltipla

É possível para uma classe derivada herdar duas ou mais classes base; neste caso, está se utilizando do mecanismo de herança múltipla suportado por C++. A forma de declarar uma classe derivada por herança múltipla é

```
class Derived : access Base1, access Base2 {  
    ...  
};
```

Observe que para cada classe base é possível especificar um modo de acesso independente.

Há controvérsias sobre o uso de herança múltipla, se ela traz benefícios ou se ela deve ser evitada totalmente. O que se põe em questão não é o mecanismo em si, mas sua validade como ferramenta de modelagem de aplicações. Em muitos casos, exemplos para ilustrar a utilidade de herança múltipla são pouco naturais. Entretanto, como o mecanismo está presente na linguagem, sua utilização ou não acaba se tornando mais uma questão de estilo pessoal.

Um detalhe que deve ser observado com relação à herança múltipla é a manipulação de nomes de membros duplicados nas classes base — como especificar a qual deles a classe derivada se refere? O mecanismo adotado para resolver tais problemas é chamado de *qualificação*, que consiste em prefixar o nome do membro (atributo ou método) com o nome da classe base a que ele se refere, sendo os nomes separados por um duplo dois-pontos ::.

4.4 Herança de construtores e destrutores

É possível que classes base e derivada tenham construtores e destrutores associados. Uma questão que pode surgir é, durante a construção de um objeto derivado, qual a ordem de chamada dos construtores? Similarmente, qual a ordem de chamada de destrutores quando o objeto é removido?

Quando um objeto de uma classe base é criado, primeiro o construtor da classe base (se existe um construtor) é invocado, e depois o construtor da classe derivada é executado. Por outro lado, quando um objeto é removido, a ordem inversa é obedecida: primeiro o destrutor da classe derivada é executado, e apenas então o destrutor da classe base é invocado.

Este procedimento é o mesmo qualquer que seja o nível de derivação: na construção, de base para derivada, e na destruição, de derivada para base. O mesmo é válido mesmo no caso de herança múltipla.

Observe que não é preciso explicitar a chamada ao destrutor da classe base — a invocação da função é automática, sendo uma consequência da ligação entre as classes.

Quando o construtor da classe base requer argumentos, é preciso adotar a forma estendida de declaração de construtores. Esta forma pode ser representada como

```
construtor-derivado (lista-arg) : base1(lista-arg),  
                                base2 (lista-arg),  
                                ...,  
                                baseN (lista-arg)
```

```
{  
  // código do construtor derivado  
}
```

onde $base1, \dots, baseN$ são os nomes das classes bases que são herdadas pela classe derivada.

Capítulo 5

Polimorfismo

Polimorfismo pode ser “traduzido” pela frase “uma interface, múltiplos métodos.” Este mecanismo é suportado em C++ tanto em tempo de compilação, como já foi visto no caso de sobrecarga de funções e operadores, como em tempo de execução, através do mecanismo de herança e de funções virtuais.

5.1 Funções virtuais

Uma *função virtual* é uma função que é declarada como *virtual* em uma classe base e redefinida pela classe derivada. Para declarar uma função como sendo virtual, é preciso preceder sua declaração com a palavra chave *virtual*. A redefinição da função na classe derivada sobrepõe a definição da função na classe base. No fundo, a declaração da função virtual na classe base age como uma espécie de indicador que especifica uma linha geral de ação e estabelece uma interface de acesso. A redefinição da função virtual pela classe derivada especifica as operações realmente executadas pelo método.

Quando acessadas normalmente, funções virtuais se comportam como qualquer outro tipo de função membro da classe. Entretanto, o que torna funções virtuais importantes e capazes de suportar polimorfismo em tempo de execução é o seu modo de comportamento quando acessado através de um apontador. Lembre-se que um apontador para a classe base pode ser usado para apontar para qualquer classe derivada daquela classe base. Quando um apontador base aponta para um objeto derivado que contém uma função virtual, C++ determina qual versão daquela função chamar baseada no tipo do objeto apontado pelo apontador. Assim, quando objetos diferentes são apontados, versões diferentes da função virtual são executadas.

Considere o seguinte exemplo:

```
#include <iostream.h>
```

```
class base {
public:
    virtual void vfunc() {
        cout << "Esta e vfunc() de base\n";
    }
};

class derived1 : public base {
public:
    void vfunc() {
        cout << "Esta e vfunc() de derived1\n";
    }
};

class derived2 : public base {
public:
    void vfunc() {
        cout << "Esta e vfunc() de derived2\n";
    }
};

main() {
    base *p, b;
    derived1 d1;
    derived2 d2;

    // p aponta para base
    p = &b;
    p->vfunc();
    // p aponta para derived1
    p = &d1;
    p->vfunc();
    // p aponta para derived2
    p = &d2;
    p->vfunc();

    return(0);
}
```

O resultado da execução deste programa é:

```
Esta e vfunc() de base
Esta e vfunc() de derived1
```


Esta e `vfunc()` de `derived2`

Observe que:

- a declaração de `vfunc()` deve ser precedida pela palavra chave *virtual*;
- a palavra chave *virtual* não é necessária na declaração das funções `vfunc()` das classes derivadas;
- a definição de qual versão da função `vfunc()` é invocada só é obtida em tempo de execução: ela não é baseada no tipo declarado de `p`, mas sim para quem `p` aponta;
- polimorfismo em tempo de execução só é atingido através do uso de apontadores para classe base. Se `vfunc()` for chamada da maneira “usual,” isto é, usando o operador ponto, como em `d2.vfunc()`, então a função, mesmo sendo declarada como *virtual*, tem o mesmo comportamento de outras funções membro.

A redefinição de uma função *virtual* por uma classe derivada é similar a sobrecarga de funções. Entretanto, este termo não é aplicado para a redefinição de funções virtuais porque há diversas diferenças. Talvez a mais importante é que o protótipo para uma função *virtual* deve combinar exatamente o protótipo especificado para a classe base. Caso o protótipo seja diferente, o comportamento será exatamente o mesmo de uma função sobrecarregada — o comportamento *virtual* da função será perdido. Outra restrição importante é que funções virtuais devem ser membros da classe das quais elas fazem parte — não podem simplesmente ser funções amigas. Por estes motivos, a redefinição de funções virtuais por classes derivadas é usualmente chamada de *sobreposição*, e não de sobrecarga.

Deve-se observar que a “virtualidade” é hereditária. Isto é, quando uma função *virtual* é herdada, sua natureza *virtual* também é herdada — mesmo quando a palavra chave *virtual* não é explicitamente declarada no método da classe derivada. Assim, o caráter *virtual* da função é mantido, não importa qual o número de níveis na hierarquia de classes.

O uso da palavra chave *virtual* indica que a função pode ser sobreposta, mas não obriga a sobreposição. Caso a classe derivada não sobreponha uma função *virtual*, então o método correspondente da classe base será utilizado.

5.2 Funções virtuais puras e classes abstratas

Há situações em que a é desejável garantir que todas as classes derivadas de uma determinada classe base suportassem um dado método. Entretanto,

algumas vezes a classe base não tem informação suficiente para permitir alguma definição para uma função virtual. Para estes casos, C++ suporta o conceito de funções virtuais puras.

Uma função virtual pura não tem nenhuma definição dentro da classe base. Para declarar uma função virtual pura, usa-se a forma

`virtual tipo nome_função (lista-args) = 0;`

Quando uma função virtual é feita pura, qualquer classe derivada deve fornecer sua própria definição. Caso contrário, um erro de compilação será acusado.

Classes que contenham pelo menos uma função virtual pura são chamadas de *classes abstratas*. Como uma classe abstrata contém um ou mais métodos para os quais não há definição (as funções virtuais puras), objetos não podem ser criados a partir de classes abstratas. Pode-se dizer que uma classe abstrata é a definição de um tipo incompleto, que serve como fundação para classes derivadas.

Objetos não podem ser criados para classes abstratas, mas apontadores para classes abstratas são válidos. Isto permite que classes abstratas suportem polimorfismo em tempo de execução.

5.3 Gabaritos

*Gabaritos*¹, também chamados de tipos ou funções parametrizados ou genéricos, permitem a construção de uma família de funções ou classes relacionadas.

5.3.1 Gabaritos de funções

Considere a seguinte função que troca os valores de dois inteiros:

```
void swap (int& x, int& y) {  
    int tmp = x;  
    x = y;  
    y = tmp;  
}
```

Considere agora a extensão desta rotina para fazer a troca de valores para *floats*, *longs*, ou objetos de quaisquer outros tipos — a solução de repetir o código acima para todas é possível, mas certamente não deve ser a melhor.

¹Templates.

Um gabarito de função permite que o computador faça esta repetição de código, e não o programador. O gabarito para a função *swap* seria definido como

```
template<class T>
void swap (T& x, T& y) {
    T tmp = x;
    x = y;
    y =tmp;
}
```

Toda vez que a função *swap* for chamada com um dado par de tipos, o compilador C++ irá até a definição acima e criará uma outra *função gabarito* como uma instância do gabarito de função acima. Funções gabaritos podem ser sobrepostas por outras definições para tipos específicos, se necessário.

5.3.2 Gabarito de classes

Um gabarito de classe permite definir um padrão para definições de classes. Assim como para gabaritos de funções, a declaração de um gabarito de classe é precedida por *template<class T>*, onde *T* é apenas uma referência a tipo ou classe que será utilizada na declaração.

O seguinte exemplo ilustra a declaração de um gabarito de classe para declarar vetores de elementos para diversos tipos.

```
template<class T>
class Vector {
    T *data;
    int size;
public:
    Vector(int);
    ~Vector() { delete [] data; }
    T& operator[] (int i) { return data[i]; }
};

// observe a sintaxe para definicao fora da classe
template<class T>
Vector<T>::Vector(int n) {
    data = new T[n];
    size = n;
};

// exemplo de declaracao
```

```
main () {  
    Vector<int> ix(5);           // gera um vetor de inteiros  
    Vector<float> fx(6);        // gera vetor de floats  
  
    //...  
}
```

Observe que, ao contrário de funções gabaritos, classes gabaritos (as instâncias de gabaritos de classes) devem ser explícitas sobre os parâmetros sobre os quais elas irão instanciar.

Capítulo 6

Desenvolvimento de Aplicações

Até este ponto, os diversos aspectos da linguagem C++ foram apresentados. Há, entretanto, outros aspectos que não dizem respeito exatamente à linguagem C++ mas sim a sua utilização no desenvolvimento de aplicações. Alguns destes aspectos são apresentados neste capítulo.

6.1 Projeto e Implementação

É relativamente fácil implementar um projeto orientado a objetos — definido tal como as recomendações do Capítulo 1 — com uma linguagem orientada a objetos, uma vez que as construções são semelhantes.

O primeiro passo na implementação de um projeto orientado a objetos é a declaração de classes. O mapeamento entre a representação de classes, tal como no modelo de objetos do OMT, e uma declaração de classe C++ deve ser trivial — o nome da classe está presente, assim como seus atributos e métodos. Da mesma forma, o diagrama apresenta o relacionamento entre classes bases e derivadas, permitindo implementar o mecanismo de herança de C++ com um mapeamento direto.

O passo mais complexo é a representação de associações. C++, como a maior parte das linguagens de programação orientadas a objetos, não suporta o conceito de “objeto associação” diretamente. Desta forma, associações devem ser mapeadas em termos de outras construções da linguagem — tipicamente, apontadores para classes como membros de outras classes. Entretanto, a fim de se manter a consistência entre apontadores bidirecionais, é necessário utilizar o mecanismo de amizade em C++.

Compiladores C++ geralmente incluem uma biblioteca de classes genéricas, que podem ser utilizadas para agilizar o desenvolvimento da aplicação, promovendo reuso e uniformidade de código. Exemplos de classes usualmente

fornecidas nestas bibliotecas incluem estruturas de dados tais como conjuntos, arranjos, listas, filas, pilhas, dicionários, árvores e assim por diante. Estas classes, normalmente chamadas de *classes container*, servem como um *framework* para organizar coleções de outros objetos.

Outra facilidade que tem surgido com frequência é a utilização de ferramentas CASE¹, que permitem especificar um projeto de acordo com alguma metodologia (por exemplo, OMT) e a partir desta especificação gerar documentação e esqueletos de código C++ contendo todas as declarações de classes necessárias — cabe ao implementador definir os métodos associados.

O seguinte segmento ilustra um trecho do manual de uma destas ferramentas CASE, de nome OOD:

[...]

Correntemente, OOD tem as seguintes funções primárias:

- editor gráfico genérico;
- ferramenta para a definição de diagrama de objetos (com algumas adições em relação à notação OMT original);
- geração de esqueleto de código C++ (arquivos de cabeçalho e código fonte). Os comentários e código para funções membros individuais podem ser documentados ou editados diretamente a partir de OOD. O gerador de código C++ suporta o mecanismo de herança.

[...]

Há atualmente uma grande variedade de ferramenta CASE disponível no mercado, com funcionalidades e preços abrangendo uma grande faixa — desde sistemas em domínio público até sistemas de dezenas de milhares de dólares.

6.2 Empacotamento de classes

Nesta seção discute-se o empacotamento de classes, ou seja, como classes são usualmente distribuídas em arquivos para serem utilizadas em aplicações. Os conceitos presentes nesta seção são relevantes não apenas para estruturar o seu próprio código como também para entender como classes fornecidas por outras fontes podem ser utilizadas em suas aplicações.

¹Computer Aided Software Engineering.

6.2.1 Cabeçalhos

Como na linguagem C, C++ faz amplo uso da definição de arquivos de cabeçalhos², usualmente com nome com o sufixo “.h” e incluídos no início de arquivos de código fonte com a diretiva *#include*.

No caso de C++, cabeçalhos são usados para conter declarações de classes — usualmente, um arquivo para cada classe. Em geral, definições dos métodos não são incluídas em cabeçalhos, mas sim mantidas em um arquivo à parte (o arquivo de implementação, discutido na próxima seção).

Considere o exemplo apresentado no Capítulo 4, com definições das classes *vehicle*, *car* e *truck*. Pela estratégia de empacotamento, três arquivos de cabeçalho seriam criados — por exemplo, *vehicle.h*, *car.h* e *truck.h*, respectivamente. Mais ainda, os cabeçalhos *car.h* e *truck.h* devem incluir a linha

#include “vehicle.h”

antes das declarações das classes, uma vez que a definição da classe base é necessária para a definição de suas classes derivadas.

A observação acima revela um problema potencial. Em uma aplicação que inclua dois mais dos cabeçalhos acima, um erro de compilação seria gerado por estar definindo a classe base mais de uma vez. A solução para evitar este problema, adotada também em ANSI-C, é usar as diretivas de pré-processamento *#define* e *#ifndef* juntamente com uma “variável de controle”: quando um cabeçalho é incluído pela primeira vez, a variável de controle não estará definida; assim, o segmento do arquivo de cabeçalho com a definição da classe é incluído e a variável de controle é definida. Caso contrário, a variável de controle já estará definida, e o restante do arquivo de cabeçalho (a declaração da classe) pode ser ignorado.

O exemplo abaixo ilustra a utilização desta técnica na codificação do cabeçalho *vehicle.h*:

```
// vehicle header file
#ifndef VEHICLE_H
#define VEHICLE_H

class vehicle {
protected:
    int wheels;
    float weight;
public:
    void initialize(int in_wheels, float in_weight);
```

²Header files.

```
    int get_wheels(void);  
    float get_weight(void);  
    float wheel_loading(void);  
};  
  
#endif
```

6.2.2 Implementação

Como observado na seção anterior, usualmente não se integra a definição de métodos de uma classe a arquivos de cabeçalhos. Os principais motivos para isto são:

desempenho de compilação: arquivos de cabeçalho são arquivos em código fonte, sendo portanto recompilados quando a aplicação que os incluem é modificada. Mantendo métodos em um arquivo separado, seu código pode ser pré-compilado e integrado à aplicação durante a fase de ligação;

segredo tecnológico: ao entregar a implementação de métodos em um arquivo de cabeçalho, o projetista da classe estaria entregando o código fonte de como as suas funções foram implementadas. Em alguns casos, isto seria altamente indesejável. A pré-compilação da implementação dos métodos permitiria que apenas a declaração das classes fossem abertas, com o código dos métodos sendo mantido fechado em bibliotecas de código objeto.

6.3 Estilo de programação C++

Um dos principais objetivos de C++ é reduzir a ocorrência de erros e facilitar a manutenção de projetos de software. Para atingir estes objetivos de forma efetiva, entretanto, não basta simplesmente usar C++. É preciso usá-lo de forma consistente, livres de erros comuns e com programas que sejam fáceis de ler e entender, mesmo por outros programadores que não o autor original.

A seguir, são apresentadas algumas recomendações que, sendo seguidas durante o desenvolvimento da aplicação, podem auxiliar que este objetivo seja alcançado.

1. Otimize código apenas quando você *sabe* que você tem um problema de desempenho, Pense duas vezes antes de fazê-lo.

2. Um arquivo de cabeçalho não deve conter mais do que uma definição de classe.
3. Se há código na sua aplicação que depende da máquina onde a aplicação será executada, coloque este código em um arquivo isolado, de forma que seja fácil localizá-lo quando transferir sua aplicação para outra máquina.
4. Todo arquivo que contém código fonte deve ser documentado com comentários introdutórios dando informação sobre o nome do arquivo e seu conteúdo.
5. Todos arquivos devem incluir informação de *copyright*.
6. Escreva comentários descritivos antes de cada função.
7. Todo arquivo de cabeçalho deve conter o mecanismo para evitar múltiplas inclusões do mesmo arquivo.
8. Escolha nomes de variáveis que sugiram o seu uso.
9. Na definição de classes, ordene a declaração de seções na ordem: pública, protegida, privativa.
10. Não defina funções membro dentro da definição da classe.
11. Sempre explicito o tipo de retorno de uma função.
12. Não especifique dados na seção pública de uma classe.
13. Funções pequenas (por exemplo, simplesmente retornar um valor de um membro da classe) devem ser definidas como função em linha.
14. Contrutores e destrutores não devem ser definidos como funções em linha.
15. Uma função membro que não afeta o estado de um objeto deve ser declarada como *const*.
16. Se um construtor de uma classe usa *new*, então um construtor de cópia deve ser fornecido para esta classe.
17. Uma classe que usa *new* para alocar instâncias gerenciadas pela classe deve definir um operador de atribuição.
18. Use sobrecarga de operadores com parcimônia e de maneira uniforme.

19. Quando dois operadores têm usualmente significados opostos (como `==` e `!=`), é apropriado definir ambos.
20. Evite funções com muitos argumentos.
21. Evite o uso de número variável de argumentos.
22. Quando sobrecarregando funções, todas as variações devem ter a mesma semântica, isto é, devem ser usadas para o mesmo fim.
23. Uma função pública nunca deve retornar uma referência ou um apontador para uma variável local.
24. Evite funções longas e complexas.
25. Não use a diretiva de processador `#define` para definir macros; prefira a definição de funções em linha.
26. Defina constantes usando `const` ou `enum`; nunca use `#define`.
27. Evite o uso de valores numéricos no código; dê preferência ao uso de valores simbólicos.
28. Variáveis devem ser declaradas dentro do menor escopo possível.
29. Toda variável que é declarada deve receber um valor antes de ser utilizada.
30. Evite o uso de apontadores para apontadores.
31. Evite o uso de conversões explícitas de tipos (*casts*).
32. Não escreva código que dependa de funções que usam conversão implícita de tipos.
33. O código após um *case* deve sempre ser seguido por um comando *break*.
34. Um comando *switch* deve sempre ter uma opção *default* para manipular casos inesperados.
35. Nunca use *goto*.
36. Sempre use *unsigned* para uma variável que se espera que não vá assumir valores negativos.
37. Evite o uso de *continue*.

38. O uso de *break* para sair de laços deve ser preferido ao uso de variáveis de *flag*.
39. Use parênteses para deixar claro a ordem de avaliação de operadores em expressões.
40. Não use *malloc*, *realloc* ou *free*.
41. Sempre use colchetes vazios (`[]`) quando removendo arranjos com *delete*.
42. Evite usar dados globais.
43. Não aloque memória esperando que outro alguém irá desalocá-la mais tarde.
44. Verifique os códigos de falhas de funções de bibliotecas por mais seguro que as chamadas possam parecer.
45. Não assuma que um *int* tem 32 bits ou que tem o mesmo tamanho de um *long* ou de um apontador.
46. Não assuma que você sabe como um certo tipo de dado é representado em memória.

Referências

1. Object-Oriented Modeling and Design
James Rumbaugh, Michael Blaha, William Premerlani, Frederick Eddy,
William Lorensen
Prentice Hall International Editions, 1991.
2. The C++ Programming Language, 2nd Edition
Bjarne Stroustrup
Addison Wesley, 1991.
3. C++: the Complete Reference
Herbert Schildt
Osborne/McGraw-Hill, 1991.
4. C++ Tutorial
Gordon Dodrill
Coronado Enterprises, 1992.
5. The C++ Answers to Frequently Asked Questions
Marshall P. Cline (Ed.)
news://comp.lang.c++, 1994
6. High Tea C++ Lectures
Irwin Sheer
Superconducting Super Collider Laboratory, 1993
7. Programming in C++: Rules and Recommendations
Erik Nyquist, Mats Henricson
Ellemtel Telecommunication Systems Laboratories, 1992
8. Borland C++: Programmer's Guide
Borland International, 1992
9. User's Guide to GNU C++
Michael D. Tiemann
Free Software Foundation, 1990

10. User's Guide to GNU C++ Library
Doug Lea
Free Software Foundation, 1991