Capítulo 1

Panorâmica da Programação Orientada a Objetos

Imagination is more important than knowledge.

Albert Einstein

OBJETIVOS

- Entender o porquê do surgimento do paradigma de programação orientada a objetos (POO).
- Contextualizar as diferenças entre a programação orientada a objetos e a programação estruturada.
- Apresentar as principais características das linguagens de programação orientada a objetos.
- Discutir as principais diferenças entre as linguagens C e C++.

A programação orientada a objetos (POO) é uma das maiores inovações na área de desenvolvimento de software. Esse paradigma de programação surgiu devido a limitações nas abordagens anteriores. Que limitações são essas? Sabe-se que a linguagem C++ é derivada da linguagem C e, por causa disso, outra questão emana: o que se ganha adotando C++ na solução de um problema?

Entender esse paradigma de programação é imprescindível para saber como identificar situações nas quais se pode empregá-la.

1.1. INTRODUÇÃO

A atividade de programação visa ao desenvolvimento de programas que implementam determinadas funcionalidades. Cada programa consiste em um conjunto organizado de instruções que operam sobre um conjunto de dados, processando-os, a fim de realizar alguma funcionalidade e produzir uma saída, conforme ilustrado na Figura 1.1.

Figura 1.1 – Função de um programa.

Note que essa tarefa pode ser facilmente feita quando o número de instruções é pequeno. Entretanto, quando os programas se tornam maiores, o principal problema da elaboração desses programas, ou na atividade de programação, é lidar com sua complexidade. Mas, por quê?

Programas maiores são, provavelmente, as entidades mais complicadas de criar. Observe que quanto maior for o programa, maior será a necessidade de encontrar uma maneira de organizar suas instruções de modo a otimizar a utilização de recursos computacionais (como, por exemplo, memória e CPU). Por causa dessa complexidade, os programas são muito menos previsíveis, havendo a tendência de apresentarem erros, e os erros de software podem ter elevado custo de correção, bem como resultar em situações indesejáveis, como a indisponibilidade de sistemas, ou até colocar vidas em perigo.

Dentro desse contexto, a programação orientada a objetos vem oferecer uma nova forma para tratar essa complexidade. A meta é obter programas que sejam mais confiáveis e de fácil manutenção. A seguir, apresentam-se justificativas da origem do paradigma de programação orientada a objetos, bem como um conjunto de características encontradas nas linguagens orientadas a objetos.

1.2. ORIGEM DA PROGRAMAÇÃO ORIENTADA A OBJETOS

À medida que os sistemas de software crescem, também cresce a complexidade associada a eles e torna-se mais difícil satisfazer a um número cada vez maior de requisitos (do sistema). Como isso acontece?

A programação orientada a objetos (POO) é uma abordagem de programação que serve de elo entre os problemas existentes e as soluções computacionais apresentadas no campo da programação. Esse elo é de suma importância na solução de problemas grandes e complexos. Por quê?

Para que você entenda, é preciso investigar o que existia. Antes do surgimento do paradigma de programação orientada a objetos, havia um obstáculo conceitual para os programadores quando eles tentavam adaptar as entidades reais às restrições impostas pelas linguagens e técnicas de programação tradicionais.

O que acontece em uma situação real de um sistema qualquer?

Imagine, por exemplo, uma biblioteca. Nela, você tem livros, revistas e outros itens que compõem o acervo da biblioteca. Você também encontrará os usuários e os funcionários, além de outros recursos e funcionalidades que visam ao empréstimo de itens do acervo. Em tal situação, o ser humano tende a raciocinar em termos dos objetos ou entidades reais. Todavia, é importante também lembrar que, antes da POO, os programadores eram ensinados a raciocinar sobre os problemas em termos de blocos de código (funções) ou procedimentos e sobre a forma como esses (blocos de código ou funções) atuavam sobre os dados.

Observe que essas duas abordagens são completamente distintas e constituem um problema quando existe a necessidade de desenvolver um sistema complexo e/ou de grande porte.

Nesse sentido, a POO se apresenta como um paradigma de programação que permite aos programadores raciocinar e solucionar problemas em termos de objetos, os quais estão diretamente associados às entidades ou "coisas" reais. Como resultado desse mapeamento natural, utilizando a POO um programador pode concentrar-se mais nos objetos que compõem o sistema, em vez de tentar vislumbrar como o sistema poderia ser decomposto em um conjunto de procedimentos e dados.

Você já deve ter percebido que a POO é uma forma natural e lógica pela qual os seres humanos, e especificamente os programadores, raciocinam. Outra motivação para o surgimento da POO foram as limitações encontradas no paradigma de programação anterior, conforme discutido na próxima seção.

1.3. LIMITAÇÕES DAS LINGUAGENS PROCEDIMENTAIS

A programação orientada a objetos (POO) foi desenvolvida devido às limitações encontradas nas abordagens anteriores de programação. Para entender a contribuição da POO, precisamos entender quais são essas limitações e como elas surgiram nas linguagens de programação tradicionais.

1.3.1. Linguagens Procedimentais

Pascal, C, Basic e Fortran são linguagens procedimentais, isto é, cada declaração em uma linguagem procedimental diz para o computador fazer alguma tarefa: pegar um dado de entrada, adicionar uma constante, dividir por um número e mostrar o resultado (por exemplo). Um programa em uma linguagem procedimental é uma *lista de instruções* organizada em termos de funções.

Você deve observar que, para programas pequenos, nenhum princípio organizacional (ou paradigma de programação) é necessário. Basta o programador criar uma lista de instruções (isto é, código) que será executada num computador.

1.3.2. Divisão do Programa em Funções

Divisão e conquista é uma prática que também é empregada na computação, e, à medida que os programas se tornam maiores, torna-se difícil tratar adequadamente todas as funcionalidades de um sistema em uma única lista de instruções. Além disso, poucos programadores podem compreender um programa que possua mais de poucas centenas de instruções, a menos que o programa seja *quebrado* (dividido) em unidades menores (funções).

Por essa razão, inicialmente a função foi adotada como uma forma de tornar os programas mais compreensíveis para seus criadores (seres humanos). O termo *função* é usado tanto em C++ quanto em C. Em outras linguagens, o mesmo conceito pode ser referido como uma sub-rotina, um subprograma ou um procedimento. Um programa é dividido em funções, e (idealmente, no mínimo) cada função tem uma proposta claramente definida, bem como uma interface bem definida com as outras funções do programa.

A ideia de quebrar um programa em funções pode ser adicionalmente estendida através do agrupamento de várias funções em uma entidade maior, denominada *módulo*. Porém, o princípio é similar: um grupo de componentes que executam tarefas específicas.

1.3.3. Programação Estruturada

Dividir um programa em funções e módulos é um dos fundamentos da *programação estruturada*. Trata-se de uma disciplina de programação que tem influenciado a organização de programas há vários anos. Mas esse paradigma possui problemas. Então, *quais são os problemas de programação estruturada?*

À medida que os programas se tornam maiores e mais complexos, até mesmo a abordagem de programação estruturada começa a mostrar "sinais de fraqueza". Que sinais são esses? Você pode até já ter escutado que, devido à complexidade do projeto, se tornou difícil cumprir as metas previstas no cronograma de desenvolvimento de um software.

Analisar as razões para essas dificuldades ou defeitos revela que existem limitações no paradigma de programação estruturada. Não importa quão bem empregada seja a programação estruturada, os programas maiores exigirão maior esforço de desenvolvimento e manutenção. Então, *quais são as razões para essas desvantagens nas linguagens procedimentais?*

1.3.4. Dados – Componente Essencial na Modelagem de um Sistema

Parte da resposta a essa questão vem do fato de que os dados têm suma importância na solução de um problema, e esse é um dos motivos para adoção da POO, já que ela descreve objetos físicos do mundo real (como, por exemplo, livro e aluno) através das entidades denominadas objetos, em uma linguagem orientada a objetos. Perceba que os dados constituem a principal razão de um sistema. Em outras palavras, *os dados constituem a essência de um sistema.*

Por outro lado, em uma linguagem procedimental, a ênfase não está nas coisas (objetos) nem nos dados, mas em fazer coisas como, por exemplo, ler dados digitados pelo usuário, ordenar um conjunto de números e verificar a condição lógica (verdadeiro ou falso) de uma variável. Ou seja, o foco de uma linguagem procedimental são as funções.

Assim, *o que acontece aos dados no paradigma de programação estruturada?*

Para responder e entender essa questão considere, por exemplo, um sistema de controle de estoque (de produtos). O que seria mais importante nesse sistema? Seria, por acaso, a função que permite a você realizar a busca por um determinado item ou o cadastro de um novo produto?

Não. A parte importante de um programa de controle de estoque não é a função que realiza a busca por item ou que mostra os dados (do estoque) nem tampouco a função que checa erros de dados de entrada. A parte mais importante é o próprio conjunto de dados do estoque.

Em um programa como esse, os dados do sistema são tratados como variáveis globais. As variáveis globais constituem os dados que são declarados fora de qualquer função e que são acessíveis a todas as funções. Essas funções executam operações de leitura, atualização ou escrita sobre esses dados. Os dados são, portanto, a razão da existência do programa.

1.3.5. Uso de Variáveis em Linguagens de Programação Estruturada

Linguagens de programação estruturada como C e Pascal fazem uso de variáveis globais e locais. Em tais linguagens, as variáveis locais são usadas apenas dentro do escopo da função onde são declaradas. Além disso, as variáveis locais não permitem que dados importantes sejam acessados por muitas e diferentes funções. Em tal situação, é necessário o uso de variáveis globais. A Figura 1.2 mostra a relação entre variáveis globais e locais.

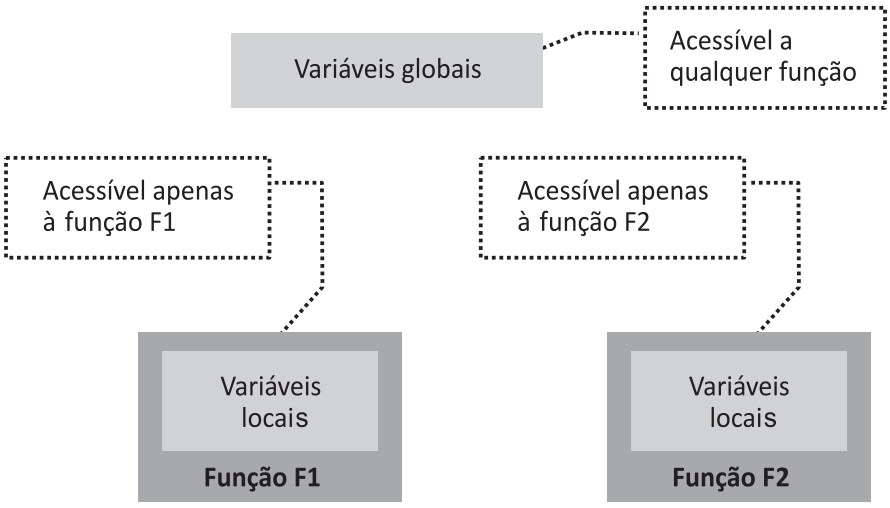

Figura 1.2 – Relação entre variáveis locais e globais.

A Figura 1.2 ilustra que as variáveis locais declaradas na função F1 têm seu escopo apenas na função F1, não sendo visível às outras funções do programa, como F2. Em outras palavras, se você declarar uma variável *x* como sendo do tipo inteiro na função F1, ela poderá apenas ser utilizada em F1. Se, por acaso, você tiver outra função em F2 que faça uso de uma variável *x*, necessitará declarar outra variável *x* que terá seu escopo limitado à função F2.

Agora, considere a situação em que muitas funções têm acesso aos dados (globais). A relação entre funções e dados em programas procedimentais é ilustrada na Figura 1.3.

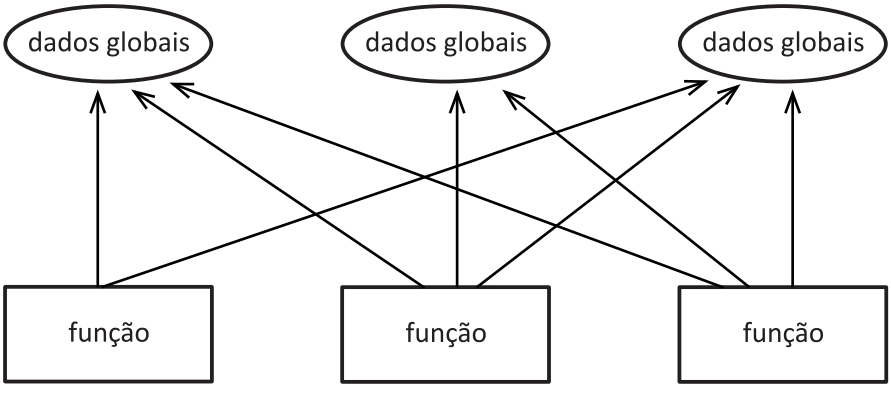

Figura 1.3 – Relações entre funções e dados em programas procedimentais.

Nesse cenário, a forma como os dados estão armazenados e são acessados torna-se crítica. Perceba que a organização dos dados não poderá ser modificada sem

modificar todas as funções que têm acesso a eles. Se novos dados são adicionados, será necessário modificar todas as funções que têm acesso a esses dados para que elas também possam ter acesso aos novos dados. Portanto, será difícil achar tais funções, permitir que elas tenham acesso adequado aos dados e mesmo mais difícil modificá-las corretamente. Então, o que é necessário?

Observe que é importante encontrar uma forma de restringir o acesso aos dados, escondê-los de todas as funções críticas, exceto de algumas poucas. Isso dará proteção aos dados, simplificará a manutenção e trará outros benefícios.

1.3.6. Problemas de Programas Procedimentais

Costuma-se dizer que os programas procedimentais são frequentemente difíceis de ser projetados. Por quê? O problema é com os seus principais componentes, isto é, as funções e as estruturas de dados não modelam o mundo real muito bem. Em outras palavras, *programas procedimentais não retratam bem a relação com a realidade*. Considere o caso em que você vai escrever um programa para criar os elementos de uma interface gráfica, envolvendo menus, janelas etc. De quais funções e estruturas de dados você necessitará?

Em vez de começar a conceber o programa em termos de funções, seria muito mais interessante utilizar as janelas e menus como elementos dos elementos do programa. É exatamente isso que a programação orientada a objetos faz, como apresentado neste livro.

Adicionalmente, existem outros problemas com as linguagens procedimentais. Um deles é a dificuldade de criar novos tipos de dados. As linguagens de programação, tipicamente, têm vários tipos de dados embutidos: inteiros, reais, caracteres etc. Se você desejar criar um novo tipo dado, por exemplo, trabalhar com coordenadas bidimensionais ou datas, isso não é tão facilmente tratado, como ocorre nas linguagens de programação orientada a objetos. A habilidade de criar os dados desejados é chamada de *extensibilidade*. Ou seja, você pode estender as características da linguagem. O resultado é que esses programas são mais difíceis de ser escritos e mantidos, embora você possa fazer isso com as linguagens procedimentais.

1.4. CARACTERÍSTICAS DAS LINGUAGENS ORIENTADAS A OBJETOS

1.4.1. A Abordagem de Orientação a Objetos

Devido ao exposto, considera-se importante a introdução da programação orientada a objetos (POO). A ideia por trás das linguagens de programação orientada a objetos é combinar em uma única entidade tanto os dados quanto as funções que operam sobre esses dados. Tal entidade é denominada *objeto*.

As funções de um objeto, chamadas funções-membro em C++, tipicamente oferecem uma única forma de acesso aos dados. Se você precisar ler dados de um objeto, chame a função-membro desse objeto. Essa função lerá os dados e retornará o valor para você. É importante observar que não se pode acessar os dados diretamente. Isso acontece porque os *dados ficam escondidos* (*hidden*). Como consequência, eles ficam seguros quanto a qualquer alteração acidental.

1.4.2. Encapsulamento de Dados

Em linguagens de POO, os *dados e funções são encapsulados* em uma única entidade – o *objeto*. O encapsulamento de dados e a ocultação de dados (ou *data hiding*) são aspectos importantes na descrição de linguagens orientada a objetos. O que isso implica?

Considere a situação em que você quer modificar o(s) dado(s) de um objeto; é preciso saber exatamente quais funções interagem com esse objeto (isto é, as funções-membros do objeto). Outras funções não podem ter acesso ao(s) dado(s). Isso evita alterações indevidas, além de simplificar a elaboração, a depuração e a manutenção do programa.

Tipicamente, um programa C++ consiste em muitos objetos, os quais se comunicam entre si através da chamada às funções-membros do(s) outro(s) objeto(s). Aqui, chamar a função-membro de um objeto pode ser entendido como *enviar uma mensagem* para o objeto. A Figura 1.4 mostra a organização de um programa C++.

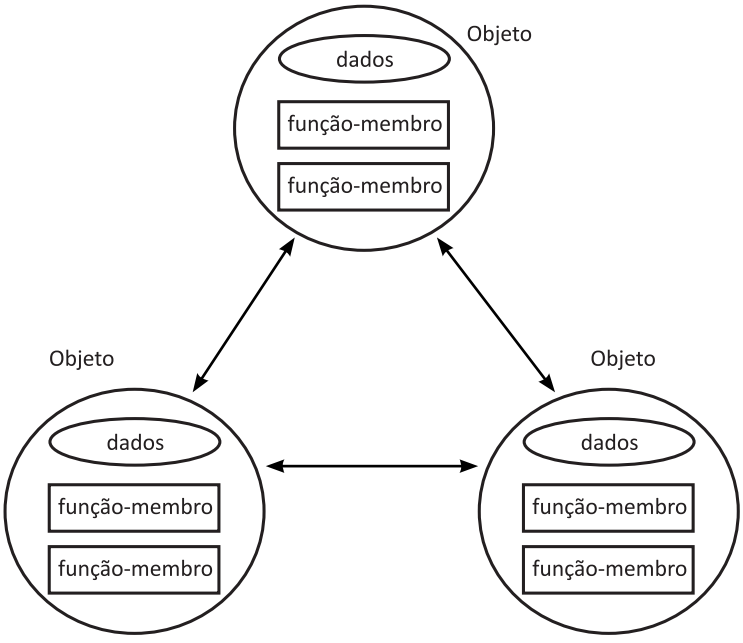

Figura 1.4 – Interação entre objetos.

1.4.3. Raciocínio em Termos de Objetos

É importante ressaltar que, quando um problema é tratado por uma linguagem orientada a objetos (OO), o programa é dividido em objetos, e não mais em funções. *Pensar em termos de objetos* tem um efeito positivo sobre quão facilmente os programas podem ser projetados. Isso é resultado da relação existente entre os objetos no sentido de programação e objetos no mundo real. *Quais coisas ou entidades se tornam objetos em programas orientados a objetos?*

Isso depende de nossa imaginação. A seguir, um conjunto de exemplos é apresentado.

1. Objetos físicos: automóveis em uma simulação de fluxo de tráfego.
2. Componentes elétricos em um programa de projeto de circuito.
3. Componentes de um aplicativo no computador: janelas, menus, objetos gráficos (linhas, retângulos).
4. Tipos (de dados) definidos pelo usuário: tempo, pontos no plano.

Em POO, diz-se que os objetos são membros de classes. O que isso significa? Considere a seguinte analogia. Quase todas as linguagens de programação possuem tipos de dados predefinidos. Por exemplo, o tipo de dado *int* para inteiro é predefinido em C++. Assim, você pode declarar tantas variáveis do tipo *int* quantas desejar em um programa, como no exemplo a seguir.

```
int dia;
int contador;
int resposta;
```

1.4.4. Classe

De modo similar, você pode definir muitos objetos como pertencentes à mesma classe. Uma *classe* serve como um padrão, modelo ou *template*. Ela especifica quais dados e quais funções serão incluídos nos objetos daquela classe. Definir uma classe não cria quaisquer objetos, assim como a simples existência de um tipo de dados *int* não cria quaisquer variáveis.

Uma classe é, portanto, uma coleção de objetos similares. Isso é semelhante ao conhecimento que você tem da palavra *classe*. Por exemplo, Pelé, Rivelino e Jairzinho são membros da classe dos jogadores de futebol. Note que não existe uma pessoa chamada *jogador de futebol*. Porém, pessoas específicas com nomes específicos são membros dessa classe se possuem determinadas características.

1.4.5. Herança

A ideia de classes conduz à ideia de herança. No cotidiano, utilizamos o conceito de classes divididas em subclasses. Por exemplo, sabemos que a classe dos animais é dividida em mamíferos, anfíbios, insetos, pássaros etc. A classe dos veículos é dividida em carros, caminhões, ônibus e motocicletas.

O princípio de herança considera a divisão de uma classe em subclasses que compartilham características comuns com a classe da qual ela é derivada. No exemplo anterior, carros, caminhões, ônibus e motocicletas possuem rodas e um motor. Essas são as características que definem os veículos.

Adicionalmente às características compartilhadas com outros membros da classe, cada subclasse também possui suas próprias características, como, por exemplo, os ônibus possuem assentos para muitas pessoas. Assim, ônibus é uma subclasse ou especialização da classe veículos. Essa ideia é ilustrada na Figura 1.5.

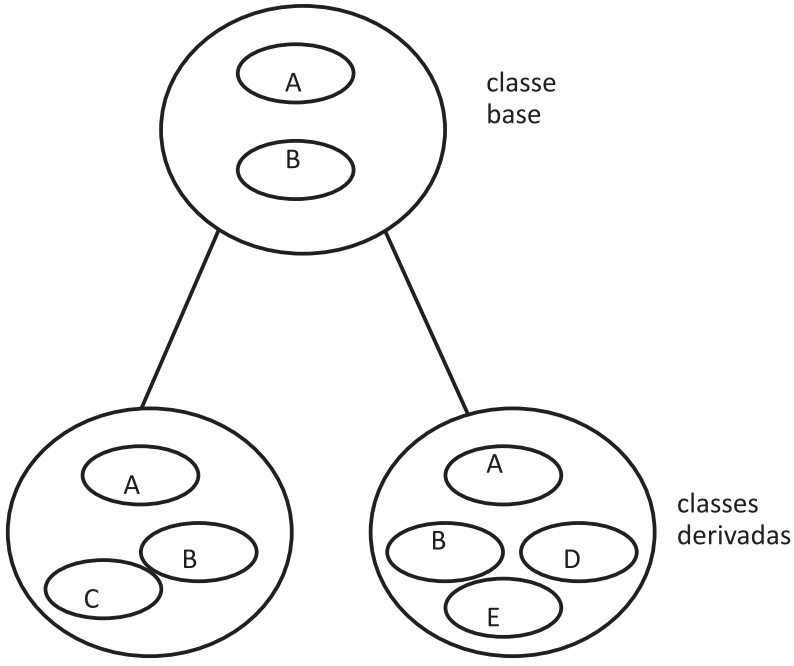

Figura 1.5 – Herança de classes.

Observe que características A e B (pertencentes à classe base) são comuns a todas as classes derivadas. Todavia, cada classe possui, adicionalmente, suas próprias características (o que configura uma especialização de classe).

Similarmente ao exemplo anterior, em uma linguagem POO uma classe pode ser dividida em subclasses. Em C++, a classe original é denominada classe base ou superclasse. Outras classes podem ser definidas compartilhando características da classe base, porém também adicionando características próprias. Estas são denominadas classes derivadas.

Não confunda a relação de objetos e classes com a relação de uma classe base e suas classes derivadas. Os objetos possuem características exatas de suas classes, as quais servem como modelo ou *template*. As classes derivadas herdam algumas características de sua classe base, porém adicionam novas características também.

Herança é análogo a usar funções para simplificar um programa procedimental. Se achamos três diferentes seções de um programa procedimental que fazem exatamente a mesma coisa, identificamos a oportunidade de extrair os elementos comuns dessas três seções para colocá-los juntos em uma única função. As três seções do programa podem chamar a função para executar as ações comuns e realizar seus processamentos individuais.

De modo similar, uma classe base contém elementos comuns a um grupo de classes derivadas. Assim como as funções em um programa procedimental, a herança reduz o tamanho do programa OO e torna mais clara a relação entre os elementos do programa.

1.4.6. Reusabilidade

Uma vez que uma classe tenha sido criada, escrita e depurada, ela pode ser distribuída para outros programadores para que eles a usem em seus programas. Isso é chamado de *reusabilidade*. É similar à forma pela qual uma biblioteca de funções em uma linguagem procedimental pode ser incorporada em diferentes programas.

Todavia, em POO, o conceito de herança oferece uma importante extensão à ideia de *reusabilidade*. Um programador pode pegar uma classe existente e, sem modificá-la, adicionar novas características a ela. Isso é feito através da derivação de uma nova classe a partir da já existente. A nova classe herdará as características da antiga, porém observe que ela fica livre para ter novas características incorporadas.

1.4.7. Criação de Novos Tipos de Dados

Um dos benefícios dos objetos é que eles oferecem ao programador uma forma conveniente de construir novos tipos de dados. Suponha que você necessite trabalhar com coordenadas x e y (par de quantidades numéricas independentes) no

seu programa. Seria interessante expressar operações sobre esses valores simplesmente usando operações aritméticas normais, tais como:

```
posição1 = posição2 + origem
```

onde as variáveis `posição1`, `posição2` e origem representam coordenadas.

Perceba que, ao criar uma classe que incorpore esses dois valores, e declarando `posição1`, `posição2` e origem como objetos dessa classe, você estará criando um novo tipo de dado. Diversos aspectos de C++ têm o objetivo de facilitar a criação de novos tipos de dados.

1.4.8. Polimorfismo e *Overloading* (Sobrecarga)

Observe que os operadores **=** (igual) e **+** (soma), usados na operação aritmética discutida, não atuam da mesma forma quando usados nas operações com os tipos predefinidos como *int* (inteiro). Os objetos posição1 e os demais não são predefinidos em C++. Todavia, esses objetos são definidos pelo programador da classe *Posição*. Como os operadores = e + podem ser usados com esses objetos?

A resposta é que você pode definir novas operações para esses operadores. Esses operadores serão funções-membros da classe *posição*. Usar operadores ou funções de formas diferentes, dependentes do que eles estão operando, é denominado *polimorfismo* (isto é, uma coisa com várias formas distintas). Quando um operador existente, como = ou +, tem a capacidade de ser usado em novos tipos de dados, diz-se que ele está *sobrecarregado* (*overloaded*). *Overloading* é um tipo de polimorfismo e mais uma das características de C++.

1.5. DIFERENÇAS ENTRE C E C++

C++ é uma linguagem derivada de C. Portanto, C++ retém a maioria das características da linguagem C. Pode-se dizer que C++ é um superconjunto de C, ou seja, quase toda instrução correta em C é também correta em C++, embora o inverso não seja verdade. C++ oferece suporte à maior quantidade de estilos de programação, tornando-a mais versátil e flexível.

Os elementos mais importantes adicionados à linguagem C para criar C++ compreendem classes, objetos e a POO. No entanto, C++ possui muitas outras características novas também, incluindo uma abordagem melhorada para entrada e saída (E/S) e nova forma para comentários, que serão apresentadas nos capítulos subsequentes.

Dentro desse contexto, você deve observar que o fato de ter programas em C++ compatíveis com programas em C permite não apenas fazer uso da eficiência da linguagem C, mas também adicionar flexibilidade e facilitar a programação de sistemas de software. Trata-se de uma demanda dos desenvolvedores de software.

RESUMO

Neste capítulo, você teve a oportunidade de aprender sobre as origens da programação orientada a objetos (POO) e entender por que esse novo paradigma de programação se tornou necessário. Descobriu as limitações existentes no paradigma de programação anterior à POO, onde se tem as linguagens procedimentais. Um conjunto de características das linguagens orientadas a objetos foi apresentado, e foram destacadas as diferenças entre as linguagens C++ e C. No próximo capítulo, você conhecerá os principais componentes de um programa na linguagem C++ e irá explorar exemplos de entrada e saída de dados, declaração, uso e conversão de variáveis de diversos tipos, além de conhecer vários operadores e funções da biblioteca C++.

QUESTÕES

1. Quais as principais características do paradigma da programação estruturada?
2. Quais as principais características do paradigma da programação orientada a objetos?
3. Quais as limitações da programação estruturada?
4. O que é encapsulamento de dados?
5. Qual a diferença entre classes e objetos?
6. Qual as principais diferenças entre a linguagem C++ e C?

EXERCÍCIOS

Neste capítulo, você teve a oportunidade de estudar o paradigma orientado a objetos, o qual oferece vantagens como definição de novos tipos (de dados), suporte a herança e reusabilidade. Faça uma pesquisa visando responder as seguintes questões:

1. Em que situações é essencial utilizar herança em um programa? Por quê?
2. O que o reuso de (artefatos de) software proporciona? Apresente exemplos.