

Objetos e Classes

Before God we are equally wise – and equally foolish.

Albert Einstein

OBJETIVOS

- Entender e diferenciar classes e objetos.
- Compreender os mecanismos de criação e destruição de objetos.
- Saber como utilizar argumentos objetos em funções.
- Entender abstração e encapsulamento em orientação a objetos

Até aqui, você aprendeu que estruturas oferecem uma forma de agrupar tipos de dados, enquanto funções permitem organizar melhor seu programa, tornando-o mais modular. Neste capítulo, você estudará classes e objetos, procurando destacar inicialmente os detalhes das classes e objetos. Neste capítulo, você estudará funções, que servem para agrupar várias instruções em uma unidade (a função). Um destaque é dado à forma pela qual você pode criar e destruir objetos. Este capítulo apresenta os recursos de classes e objetos e ilustra seu uso com diversos exemplos. Nesse sentido, questões a serem exploradas são: como definir uma classe e criar (ou destruir) objetos? Objetos podem ser utilizados como argumento de funções? Como? O que é encapsulamento? Responder a essas e outras questões é o objetivo deste capítulo e, para tanto, exemplos são usados para ilustrar situações em que é adequado o uso de classes e objetos.

6.1. INTRODUÇÃO

Classes e objetos constituem um dos fundamentos da programação orientada a objetos. Diferentemente da programação estruturada, pela qual se procura racio-

cinar e organizar o código em termos de funções, aqui você estará raciocinando e organizando o código em termos de objetos.

6.1.1. Objetos e Classes

Cada objeto possui dados (geralmente, denominados atributos) e operações que em C++ são chamadas de funções-membros. Você pode ainda agrupar objetos em classes. Uma classe consiste em uma coleção de objetos, enquanto cada objeto é uma instância de uma classe.

Os fundamentos da programação orientada a objetos compreendem classes, objetos, mensagens (operações), herança, polimorfismo, abstração e encapsulamento. Neste capítulo, você terá a oportunidade de aprender mais um pouco sobre cada um deles.

6.1.2. Atributos e Operações em Classes

Todos os objetos que pertencem a uma classe compartilham os mesmos atributos (ou dados) e funcionalidades dessa classe. Em geral, cada objeto tem um único estado, o qual é definido pelos valores dos atributos. Já as funcionalidades da classe determinam as operações realizadas pelos objetos da referida classe.

6.1.3. Mensagens (Funções-membros)

Em programação orientada a objetos, há interação entre objetos que se dá através da troca de eventos ou mensagens. Na interação entre objetos, enviar uma mensagem de um objeto para outro representa uma chamada de função-membro ou método.

6.1.4. Herança

Em programação orientada a objetos, você pode derivar uma classe de outra já existente. Ao derivar uma nova classe, você está refinando ou especializando o comportamento da classe pai. Em tal situação, você pode adicionar novas funções à classe derivada.

6.1.5. Polimorfismo

Em programação orientada a objetos, você pode escrever classes que contêm funções (membros) que respondem de modo diferente em determinadas situações.

Em outras palavras, o polimorfismo possibilita que objetos de classes distintas respondam à mesma função de maneira adequada para cada objeto.

6.1.6. Encapsulamento

Em programação orientada a objetos, os dados e operações (funções) são *encapsulados* em uma única entidade (objeto). O encapsulamento de dados e a ocultação de dados (ou *data hiding*) são características importantes na descrição de linguagens orientadas a objetos.

6.1.7. Abstração (de Dados)

A linguagem C++ suporta muitas das funcionalidades oferecidas pela linguagem C, mas é enriquecida com várias outras características, como suporte ao paradigma de programação orientada a objetos, abstração de dados (ou tipos definidos pelo programador), além das outras discutidas antes. C++ provê suporte para utilizar classes (que possui relação direta com as entidades reais que são modeladas) e permite definir novos tipos de dados que podem ser usados como tipos predefinidos.

6.1.8. Exemplificando uma Classe

Para entender melhor o conceito de classe e objetos, vamos explorar um exemplo. Inicialmente, vamos entender o programa da Listagem 6.1, que possui uma classe e dois objetos dessa classe. Embora ele seja simples, o programa apresenta a sintaxe e aspectos gerais de uma classe em C++.

```
1. #include <iostream>
2. using namespace std;
3. // Programa para ilustrar definicao de uma classe
4. class minhaClasse // especificacao de classe
5. {
6.     private:
7.         int x;
8.     public:
9.         void setValor(int d) // funcao membro
10.        { x = d; }
11.        void mostraValor() // funcao membro
12.        { cout << "\nValor = " << x << endl; }
13. };
14. int main()
15. {
16.     minhaClasse obj1,obj2; // declaracao de 2 objetos
```

```
17.  
18. obj1.setValor(11); // chamada a funcao membro para definir  
    valores  
19. obj2.setValor(22);  
20. obj1.mostraValor(); // chamada a funcao membro para mostrar  
    valores  
21. obj2.mostraValor();  
22. cout << endl;  
23. system("PAUSE");  
24. return 0;  
25. }
```

Listagem 6.1

A classe *minhaClasse* da Listagem 6.1 possui um item de dado e duas funções-membros. Essas funções oferecem o acesso aos dados da classe. Note que dados e funções foram colocados juntos em uma só entidade, a qual é a ideia de programação orientada a objetos. O programa da Listagem 6.1 ilustra a definição de uma classe *minhaClasse* e a criação de dois objetos. Execute o programa e, depois, o programa exibirá a saída mostrada na Figura 6.1.

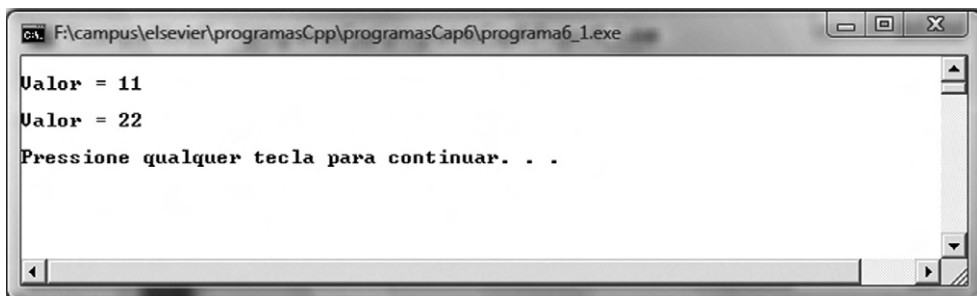


Figura 6.1 – Saída do programa da Listagem 6.1.

No Capítulo 1, você estudou os conceitos do paradigma de programação orientada a objetos. Lembrando: um objeto é uma instância de uma classe. No programa da Listagem 6.1, você tem dois objetos, os quais são instâncias da classe *minhaClasse*. Agora, se você observar a especificação da classe, verá que o corpo da classe possui duas palavras-chave: *private* e *public*.

6.1.9. Ocultação de Dados

Uma característica importante em programação orientada a objetos é a ocultação de dados (*data hiding*). Isso significa que os dados estão escondidos na classe, de modo

que eles não podem ser acessados acidentalmente por funções fora da classe. O mecanismo básico para a ocultação de dados é colocar os dados na classe e torná-los *private*. Dados ou funções do tipo *private* podem ser acessados apenas de dentro da classe. Por outro lado, dados e funções do tipo *public* podem ser acessados de fora da classe.

6.1.10. Dado *private*

No programa da Listagem 6.1, temos apenas um item de dado (x) do tipo inteiro. Contudo, você pode ter qualquer número de itens de dado em uma classe, similarmente como em uma estrutura. O dado x é *private* e, portanto, pode ser acessado apenas de dentro da classe.

6.1.11. Função *public*

Além disso, temos duas funções-membros (ou métodos) dentro da classe. Como elas são *public*, podem ser acessadas de fora da classe. Geralmente, utilizam-se os dados de uma classe como *private* e as funções como *public*. Isso ocorre para evitar a manipulação acidental dos dados, bem como permitir o acesso às funções de fora da classe. Todavia, isso não é regra e pode-se ter dados *public* e/ou funções *private*.

6.1.12. Função-membro

Observe que as funções-membros *setValor()* e *mostrarValor()* são definições (isto é, suas definições estão dentro da classe). Funções-membros definidas dentro da classe dessa forma são criadas como funções *inline* (por default). Depois, você verá que também é possível declarar uma função em uma classe e defini-la em outro local.

Para chamar as funções-membros, você deve utilizar a sintaxe mostrada nas linhas 18 e 19, reproduzidas a seguir:

```
obj1.setValor(11);  
obj2.setValor(22);
```

Uma função-membro é sempre chamada para atuar sobre um objeto específico ou é acessada por um objeto específico. A sintaxe é similar à de estruturas. O *operador ponto* é denominado operador de acesso a membro da classe. Assim, a primeira instrução, da linha 18, executa a função-membro *setValor()* no objeto *obj1* (isto é, a função atribui o valor 11 à variável x do objeto *obj1*). Também se pode pensar nas chamadas às funções-membros como mensagens. Dessa forma, a instrução *obj1.mostrarValor()*, da linha 20, pode ser interpretada como enviar uma mensagem a *obj1* dizendo para ele mostrar o valor de seu dado.

6.2. OBJETOS C++

Objetos C++ compartilham os mesmos atributos e funcionalidades dos outros objetos da mesma classe. Além disso, há uma relação direta entre objetos C++ com os objetos físicos. Para entender melhor esse conceito, vamos explorar outro exemplo.

Praticando um Exemplo. Escreva um programa em C++ que cria uma classe chamada *estoqueCelular*, a qual possui três itens de dados (atributos): *codigoFabricante*, *codigoModelo* e *custo*. Todos esses dados (atributos) devem ser do tipo `private`. Além disso, você deve também implementar duas funções *setDados()* e *mostraDados()*. A função *setDados()* fornece os valores dos itens de dados, e a função *mostraDados()* mostra os valores armazenados nesses três itens. Para elaborar esse programa, você deve criar uma classe que atenda a esses requisitos. Note que você deve criar essas funções-membros e chamá-las a partir da função `main()` juntamente com a criação de objetos da classe *estoqueCelular*. Agora, feche o livro e tente implementar sua solução. Depois, consulte a solução do exemplo mostrada na Listagem 6.2.

```
1. #include <iostream>
2. using namespace std;
3. // Programa para ilustrar definicao de uma classe
4. class estoqueCelular // especificacao de classe
5. {
6.     private:
7.         int codigoFab;
8.         int codigoModelo;
9.         double custo;
10.    public:
11.        void setDados(int f, int m, double c) // definicao de dados
12.        {
13.            codigoFab = f;
14.            codigoModelo = m;
15.            custo = c;
16.        }
17.        void mostraDados() // mostra dados
18.        {
19.            cout << "\nCodigo do fabricante = " << codigoFab;
20.            cout << "\nCodigo do modelo = " << codigoModelo;
21.            cout << "\nCusto do aparelho: R$ " << custo;
22.            cout << endl;
23.        }
24. };;
```

```

25. int main()
26. {
27.     estoqueCelular obj1,obj2; // declaracao de 2 objetos
28.     obj1.setDados(1234,777, 117.5 ); // chamada a funcao membro
        para definir valores
29.     obj2.setDados(4567,999, 599.99);
30.     obj1.mostraDados(); // chamada a funcao membro para mostrar
        valores
31.     obj2.mostraDados();
32.     cout << endl;
33.     system("PAUSE");
34.     return 0;
35. }

```

Listagem 6.2

O programa da Listagem 6.2 ilustra a definição da classe *estoqueCelular* e a criação de dois objetos. Execute o programa e, depois, o programa exibirá a saída mostrada na Figura 6.2.

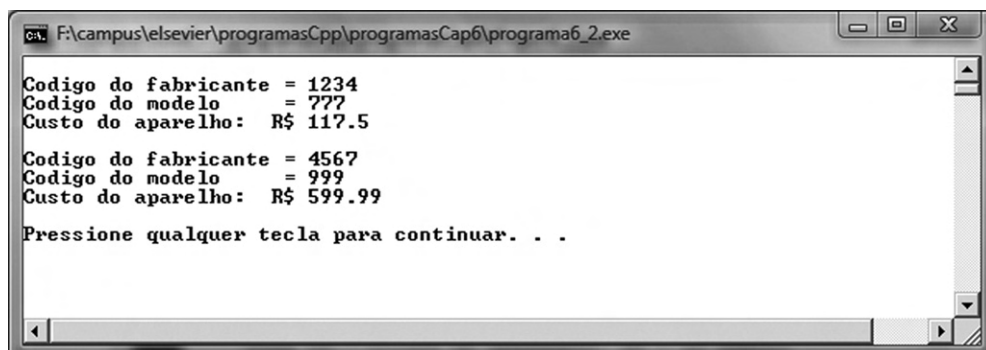


Figura 6.2 – Saída do programa da Listagem 6.2.

Nesse exemplo, dois objetos *obj1* e *obj2* são criados. A função-membro *setDados()* inicializa os três itens de dados, e a função-membro *mostraDados()* mostra esses valores.

É importante observar que, se você estiver desenvolvendo um programa para controlar o estoque de aparelhos celulares, poderá criar, por exemplo, uma classe como *estoqueCelular*. Esse é um exemplo de objeto C++ representando um objeto físico do mundo real, isto é, um dispositivo celular (que possui código de fabricante, código de modelo e custo).

6.3. CLASSES E OBJETOS

6.3.1. Classes, Objetos e Memória

Provavelmente, a ideia que temos de objetos é que eles são criados a partir das classes e que possuem cópias de dados e funções-membros da classe.

Embora isso seja uma boa aproximação, as coisas não são tão simples. É verdade que cada objeto possui seus próprios itens de dados separados. Porém, ao contrário do que você poderia imaginar, todos os objetos de uma dada classe usam as mesmas funções-membros.

As funções-membros são criadas e colocadas em apenas um local na memória (quando são definidas no especificador da classe). Não há justificativa em se querer duplicar todas as funções-membros em uma classe toda vez que um objeto daquela classe for criado porque as funções usadas pelos objetos são idênticas.

Por outro lado, os itens de dados (isto é, atributos) podem conter valores diferentes; daí existirem instâncias separadas de cada item de dado em cada objeto. Dessa forma, dados são colocados na memória toda vez que um objeto é definido.

A Figura 6.3 ilustra o fato de as funções-membros de objeto serem compartilhadas em uma classe. Ou seja, não existe duplicação de funções quando da criação de objetos. Perceba também que não existe conflito (desconsiderando um sistema multitarefas), pois apenas uma função é executada por vez. O importante é fazermos a abstração de que cada objeto possui seus dados e funções.

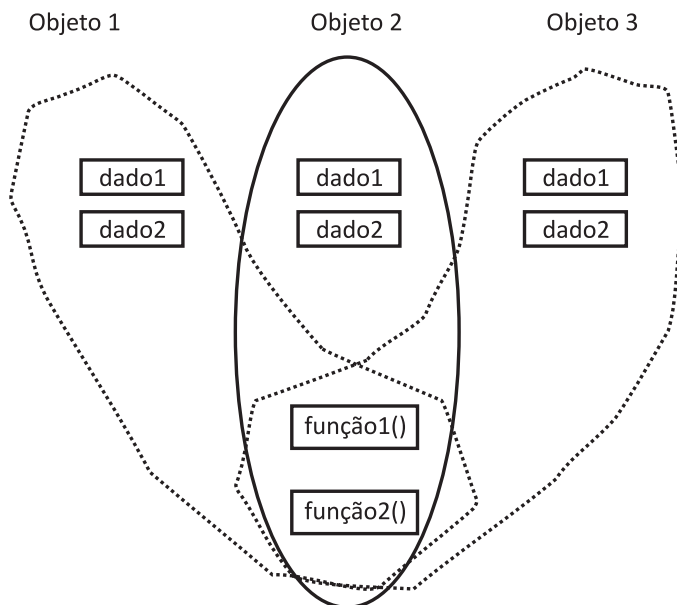


Figura 6.3 – Funções-membros compartilhadas por objetos de uma classe.

6.3.2. Objetos C++ como Tipos de Dados

Objetos em C++ representam variáveis de tipos de dados definidos pelo usuário. Você pode utilizar objetos para representar entidades que tenha em um problema. Para entender melhor esse conceito, vamos explorar um exemplo.

Praticando um Exemplo. Escreva um programa em C++ que cria uma classe chamada *Medida*, a qual possui dois itens de dados (atributos): *metro* e *centímetro*. Além disso, a classe *Medida* tem três funções: *setMedida()*, que define os valores dos itens de dados, *getMedida()*, que requisita ao usuário entrar com valores das porções metro e centímetro da medida, e a função *mostraDados()*, que exibe os valores armazenados nos dois itens. Para elaborar esse programa, você deve criar uma classe que atenda a esses requisitos e criar dois objetos medida (*m1* e *m2*). O objeto *m1* deve ter seus itens inicializados com a função *setMedida()*, enquanto o objeto *m2* usa a função *getMedida()* para solicitar os dados do usuário. Note que você deve criar essas funções-membros e chamá-las a partir da função *main()* juntamente com a criação de objetos da classe *Medida*. Agora, feche o livro e tente implementar sua solução. Depois, consulte a solução do exemplo mostrada na Listagem 6.3.

```

1. #include <iostream>
2. using namespace std;
3. // Programa para ilustrar definicao de uma classe
4. class Medida // Especificacao da classe Medida
5. {
6.     private:
7.         int metro;
8.         float centimetro;
9.     public:
10.        void setMedida(int m, double c) // define medida
11.        {
12.            metro = m;
13.            centimetro = c;
14.        }
15.        void getMedida() // obtem medida do usuario
16.        {
17.            cout << "\nDigite a parte de metros da medida: ";
18.            cin >> metro;
19.            cout << "Digite a parte de centimetros da medida: ";
20.            cin >> centimetro;
21.        }
22.        void mostraMedida() // exibe medida
23.        {
24.            cout << (metro + centimetro/100) << " metros \n";
25.        }

```

```
26. };  
27. int main()  
28. {  
29.     Medida m1, m2; // cria dois objetos Medida  
30.     m1.setMedida(25, 50); // define valores do objetos m1  
31.     m2.getMedida(); // obtem valores para objeto m2  
32.     cout << "\nMedida 1 = "; m1.mostraMedida(); // exibe valor  
        total  
33.     cout << "\nMedida 2 = "; m2.mostraMedida();  
34.     cout << endl;  
35.     system("PAUSE");  
36.     return 0;  
37. }
```

Listagem 6.3

O programa da Listagem 6.3 ilustra a definição da classe *Medida* e a criação de dois objetos *Medida* *m1* e *m1*. Execute o programa e digite, por exemplo, 100 para a porção de metros da medida e 25 para a porção de centímetros; depois, o programa exibirá a saída mostrada na Figura 6.4.

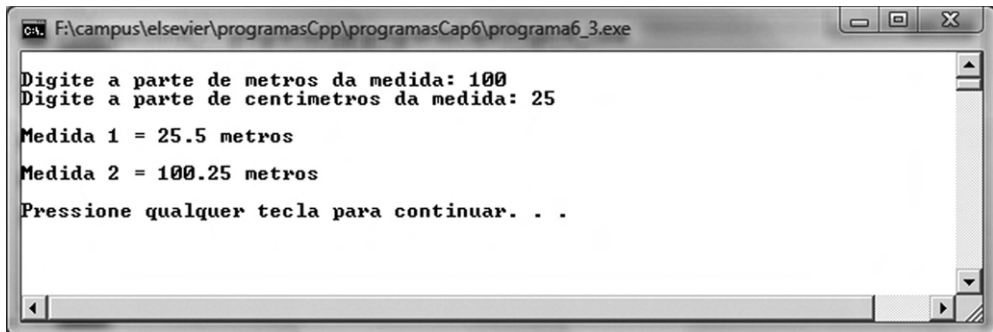


Figura 6.4 – Saída do programa da Listagem 6.3.

No programa da Listagem 6.3, a classe *Medida* possui dois itens de dados: *metro* e *centímetro*. Esse exemplo é similar ao exemplo de estrutura *Medida* que você estudou no Capítulo 4. Entretanto, neste exemplo você tem três funções-membros que também são definidas: *setMedida()*, *getMedida()* e *mostraMedida()* nas linhas 10-14, 15-21 e 22-25, respectivamente.

Na função *main()* do programa, dois objetos da classe *Medida* são definidos na linha 29: *m1* e *m2*. O primeiro tem seu item de dado inicializado pela função *setMedida()* na linha 30, enquanto o segundo é inicializado como valor fornecido pelo usuário na linha 31.

O programa da Listagem 6.3 mostra duas formas de atribuir valores a itens de dados de um objeto. Algumas vezes, contudo, é conveniente inicializar o objeto quando ele é criado (sem precisar fazer uma chamada a uma função-membro, como ocorreu no exemplo anterior).

Cabe destacar que uma inicialização automática é feita usando uma função-membro especial denominada construtor (*constructor*), a qual é tratada na próxima seção.

6.4. CONSTRUTORES E DESTRUTORES

Construtores e destrutores (na linguagem C++) trabalham de maneira automática para assegurar que haja criação e remoção adequada de instâncias de classe (isto é, objetos).

6.4.1. Construtor

Um construtor é uma função-membro que é executada automaticamente sempre que um objeto é criado. A sintaxe para especificar um construtor é:

```
class NomeClasse
{
    Public:
        NomeClasse(); // construtor default
        NomeClasse(NomeClasse& x); // construtor copia
        NomeClasse(<lista de parâmetros>); // outro construtor
};
```

Para entender melhor como utilizar construtores, veja o próximo exemplo.

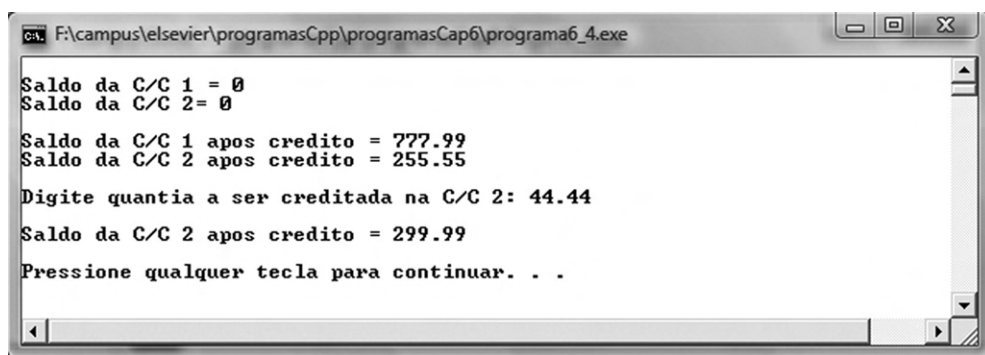
Praticando um Exemplo. Escreva um programa em C++ que cria uma classe chamada *Conta* (conta-corrente bancária), a qual possui apenas um item de dado: *saldo*. Além disso, a classe *Conta* tem duas funções: *creditar(x)*, que faz um crédito de *x* na conta referenciada, *getSaldo()*, que retorna o saldo da conta referenciada. Para elaborar esse programa, você deve criar uma classe que atenda a esses requisitos e criar dois objetos *Conta* (*c1* e *c2*). Você deve ter um construtor *Conta()* {*saldo = 0;*} para inicializar os saldos das duas contas com valor 0. Em seguida, por exemplo, chame a função *creditar(x)* para fazer depósitos nas duas contas, exiba os saldos atualizados e, por fim, solicite que o usuário informe a quantia que ele deseja creditar na conta *c2*. Note que você deve criar essas funções-membros e chamá-las a partir da função *main()* juntamente com a criação de objetos da classe *Conta*. Agora, feche o livro e tente implementar sua solução. Depois, consulte a solução do exemplo mostrada na Listagem 6.4.

```
1. #include <iostream>
2. using namespace std;
```

```
3. // Programa para ilustrar o uso de construtores
4. class Conta
5. {
6.     private:
7.         double saldo;
8.     public:
9.         Conta() { saldo = 0; } // construtor
10.        void creditar(double quantia) { saldo = saldo + quantia;
11.            } // funcao credito
12.        double getSaldo() { return saldo; } // funcao getSaldo
13. };
14. int main()
15. {
16.     double quantiaDeposito;
17.     Conta c1, c2; // cria dois objetos Conta
18.     cout << "\nSaldo da C/C 1 = " << c1.getSaldo(); // exhibe
19.         saldos iniciais
20.     cout << "\nSaldo da C/C 2= " << c2.getSaldo();
21.     c1.creditar(777.99); // credita 777.99 na c/c 1
22.     c2.creditar(255.55); // credita 255.55 na c/c 2
23.     cout << "\n\nSaldo da C/C 1 apos credito = " << c1.getSaldo();
24.         //exibe saldos
25.     cout << "\nSaldo da C/C 2 apos credito = " << c2.getSaldo();
26.     cout << "\n\nDigite quantia a ser creditada na C/C 2: ";
27.     cin >> quantiaDeposito; // ler quantia de deposito
28.     c2.creditar(quantiaDeposito);
29.     cout << "\nSaldo da C/C 2 apos credito = " << c2.getSaldo()
30.         << endl << endl;
31.     system("PAUSE");
32.     return 0;
33. }
```

Listagem 6.4

O programa da Listagem 6.4 ilustra o uso de construtor na definição da classe *Conta* e a criação de dois objetos *Medida c1* e *c1*. Execute o programa e digite, por exemplo, a quantia de depósito de 500,50 para conta *c2*; depois, o programa exibirá a saída mostrada na Figura 6.5.



```
cmd: F:\campus\elsevier\programasCpp\programasCap6\programa6_4.exe
Saldo da C/C 1 = 0
Saldo da C/C 2 = 0
Saldo da C/C 1 apos credito = 777.99
Saldo da C/C 2 apos credito = 255.55
Digite quantia a ser creditada na C/C 2: 44.44
Saldo da C/C 2 apos credito = 299.99
Pressione qualquer tecla para continuar. . .
```

Figura 6.5 – Saída do programa da Listagem 6.4.

No exemplo da Listagem 6.4, a classe *Conta* criada serve para definir o saldo de uma conta ou fazer depósito. Note que as funções `getSaldo()` e `creditar()` podem ser chamadas qualquer número de vezes.

Agora, considere que seu programa que contém a classe *Conta* deva ser acessado por muitas e diferentes funções. Em linguagens procedimentais, um item de dado saldo provavelmente seria implementado fazendo-se uso de variável externa. Todavia, como você viu no Capítulo 1, as variáveis externas podem ser modificadas acidentalmente. Entretanto, no programa da Listagem 6.4, esse item de dado será modificado apenas através de suas funções-membros.

A classe *Conta* possui um item de dado do tipo *Double*, conforme a linha 7. Também há duas funções-membros `creditar()` e `getSaldo()` nas linhas 10 e 11, respectivamente.

Quando um objeto do tipo *Conta* é criado, você deseja que ele seja inicializado para o valor 0. Você poderia, por exemplo, ter uma função `setSaldo()` e depois chamá-la com o argumento 0. Mas, se você fizer isso, toda vez que criar um objeto terá de executar essa função, como mostrado a seguir:

```
Conta c1;
c1.setSaldo();
```

6.4.2. Inicialização com Construtor

É mais conveniente (especialmente quando há grande quantidade de objetos de uma determinada classe) fazer com que cada objeto se inicie sozinho. No programa da Listagem 6.4, o construtor `Conta()` faz isso. Essa função é chamada automaticamente toda vez que um novo objeto *Conta* é criado.

■ *Note que os construtores têm o mesmo nome da classe. Isso ocorre porque é dessa forma que o compilador identifica que eles são construtores. Perceba também que os construtores não possuem tipo de retorno porque eles são chamados automaticamente pelo sistema e, portanto, não têm para quem retornar qualquer coisa.*

6.4.3. Destrutor

Você viu um tipo de função especial (construtor) que é chamada toda vez que um objeto é criado. Você pode imaginar que deve haver outra função que também é automaticamente chamada toda vez que um objeto é destruído. De fato, essa função existe e é denominada destrutor. Um destrutor possui o mesmo nome que o construtor (isto é, o nome da classe) precedido por um sinal de til (~), como mostrado a seguir.

```
class Exemplo
{
private:
    int dado1
public:
    Exemplo() { dado1 = 0; } // construtor
    ~Exemplo() { } // destrutor
}
```

Assim como os construtores, os destrutores não possuem valor de retorno nem argumento. O principal uso dos destrutores é para dealocar ou liberar memória que havia sido alocada a um objeto no momento que o construtor é chamado.

6.5. OBJETOS COMO ARGUMENTOS E RETORNO DE FUNÇÃO

6.5.1. Objetos como Argumentos de Função

Quando trabalhando com funções, você pode ter objetos que sejam usados como argumentos para a função, e também é possível ter uma função que retorna um objeto como argumento. Para entender melhor, vamos explorar o próximo exemplo, apresentado na Listagem 6.5.

```
1. #include <iostream>
2. using namespace std;
3. // Programa para ilustrar uso de objeto como argumento
4. class Medida // Especificacao da classe Medida
5. {
6.     private:
7.         int metro;
8.         double centimetro;
9.     public:
10.         Medida ()
```

```
11.     {
12.         metro = 0;
13.         centimetro = 0.0;
14.     }
15.
16. Medida(int m, double c) // define medida
17. {
18.     metro = m;
19.     centimetro = c;
20. }
21. void getMedida() // obtem medida do usuario
22. {
23.     cout << "\nDigite a parte de metros da medida: ";
24.     cin >> metro;
25.     cout << "Digite a parte de centimetros da medida: ";
26.     cin >> centimetro;
27. }
28. void mostraMedida() // exibe medida
29. {
30.     cout << (metro + centimetro/100) << " metros \n";
31. }
32. Medida somaMedida( Medida ); // adiciona medidas
33. };
34.
35. Medida Medida::somaMedida(Medida m1) // retorna o valor da soma
36. {
37.     Medida temp;
38.     temp.centimetro = centimetro + m1.centimetro;
39.     if(temp.centimetro >= 100.0) // testa se centimetro >= 100
40.     {
41.         temp.centimetro -= 100.0; // decrementa centimetro
42.         temp.metro = 1;
43.     }
44.     temp.metro += metro + m1.metro;
45.     return temp;
46. }
47. int main()
48. {
49.     Medida m1, m3; // cria dois objetos Medida
50.     Medida m2(100, 25.0);
51.     m1.getMedida(); // obtem valores para objeto m1
52.     cout << "\nValores iniciais das medidas"; // exibe as medidas
53.     cout << "\nMedida 1 = "; m1.mostraMedida();
54.     cout << "\nMedida 2 = "; m2.mostraMedida();
55.     cout << "\nMedida 3 = "; m3.mostraMedida();
```

```
56.  
57. m3 = m2.somaMedida(m1); // m3 = m2 + m1  
58. cout << "\nValores atualizados das medidas \n";  
59. cout << "\nMedida 1 = "; m1.mostraMedida(); // exibe valor total  
    em metros  
60. cout << "\nMedida 2 = "; m2.mostraMedida();  
61. cout << "\nMedida 3 = "; m3.mostraMedida();  
62. cout << endl;  
63. system("PAUSE");  
64. return 0;  
65. }
```

Listagem 6.5

O programa da Listagem 6.5 ilustra o uso de objeto como argumento em uma função na qual se definiu a classe *Medida*, e três objetos *Medida* *m1*, *m2* e *m3* foram criados. Execute o programa e digite, por exemplo, o valor 150 para a parte de metros e 25 para a parte de centímetros; depois o programa exibirá a saída mostrada na Figura 6.6.

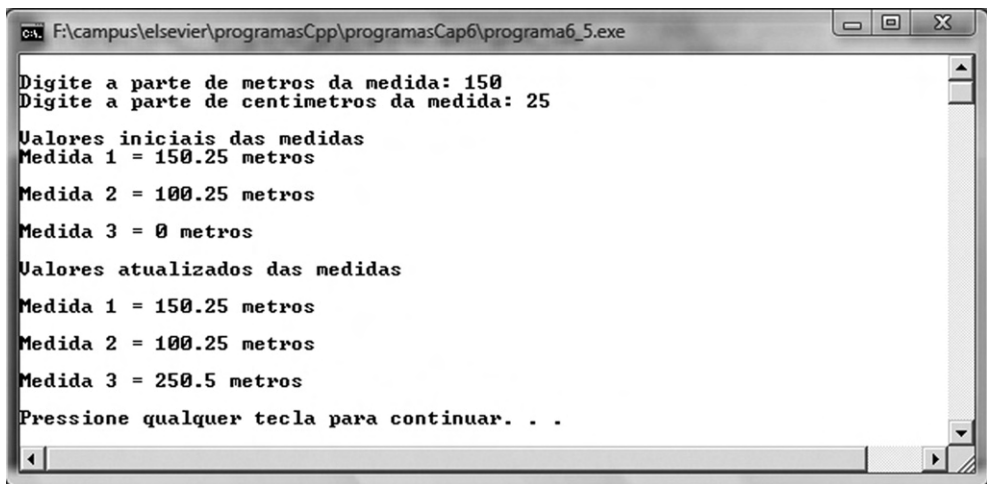


Figura 6.6 – Saída do programa da Listagem 6.5.

No programa da Listagem 6.5, duas medidas (*m1* e *m2*) são passadas como argumentos para *somaMedida()*, na linha 57, e o resultado é armazenado em um objeto (*m3*) da classe (*Medida*), da qual *somaMedida()* é membro.

Observe ainda que, no programa da Listagem 6.5, a medida (*m1*) é passada para *somaMedida()* como argumento. Ela é adicionada ao objeto *m2* (da classe da qual *somaMedida()* é membro) e o resultado é retornado da função.

Em *main()* o resultado é atribuído a *m3*. Embora os dois programas façam a mesma coisa, o uso de sinal de atribuição (=) na Listagem 6.5 é mais natural.

Depois veremos como usar o operador aritmético de soma (+) para podermos usar uma instrução como:

```
m3 = m1 + m2;
```

Adicionalmente, vejamos como o argumento e o valor de retorno (operandos) diferem entre funções-membros e não membros. Como regra, tem-se que há sempre um operando a menos em uma função-membro (de uma classe) em comparação a funções não membros. Portanto, para uma função normal que soma dois valores e retorna o resultado, teríamos:

```
v3 = funcaoSoma(v1, v2); // 3 operandos
```

Para funções-membros, existem três formas de fazer a operação equivalente, ou seja:

```
obj3.funcaoSoma(obj2, obj3); // 2 operandos; funcaoSoma() é mem-  
bro de obj3
```

ou

```
obj3 = obj2.funcaoSoma(obj3); //2 operandos; funcaoSoma() é mem-  
bro de obj2
```

Em ambos os casos, existem apenas dois operandos porque *funcaoSoma()* é membro do terceiro objeto. A mesma situação pode ser aplicada a funções que operam sobre apenas uma variável. Por exemplo, considere uma função não membro que calcula a raiz quadrada de um número (têm-se dois operandos: *v1* e *v2*).

```
v2 = raiz(v1); // 2 operandos
```

Outra versão com função-membro poderia ser escrita como:

```
obj2 = obj1.raiz(); // 1 operando; raiz() é membro de obj1
```

ou

```
obj2.raiz(obj1); // 1 operando; raiz() é membro de obj2
```

Como observação complementar, cabe destacar uma pequena diferença entre estruturas e classes. Estruturas têm sido usadas como uma forma de agrupar dados, enquanto classes são usadas como uma forma de agrupar tanto dados quanto funções. Estruturas são usadas quase com o mesmo propósito que as classes. A única diferença é que em uma classe os membros são *private* por default, ao passo que em uma estrutura eles são *public* por default.

```
class exemplo  
{  
    private:  
        int dado;
```

```
public:
    void funcao();
};
```

Visto que os dados nas classes são `private` por default, tal palavra-chave é desnecessária. Assim, poderíamos escrever:

```
class exemplo
{
    int dado;
public:
    void funcao();
};
```

Embora esteja correto, a primeira forma é preferida, bem como torna a listagem do programa mais facilmente entendida e clara.

6.6. RETORNO DE OBJETOS DE FUNÇÕES

Até aqui, você tem estudado funções-membros de classes que têm sido definidas dentro da especificação de uma classe. Entretanto, você também pode definir uma função-membro fora da especificação de uma classe. Como isso pode ser feito?

Isso aconteceu nas linhas 35-46 do programa da Listagem 6.5, reproduzido a seguir.

```
35. Medida Medida::somaMedida(Medida m1) // retorna o valor da soma
36. {
37. Medida temp;
38. temp.centimetro = centimetro + m1.centimetro;
39. if(temp.centimetro >= 100.0) // testa se centimetro >= 100
40. {
41. temp.centimetro -= 100.0; // decrementa centimetro
42. temp.metro = 1;
43. }
44. temp.metro += metro + m1.metro;
45. return temp;
46. }
```

O fragmento de código contém uma sintaxe não familiar. A função *somaMedida()* é precedida por um novo operador (`::`) e pelo nome da classe *Medida*. Esse novo operador (`::`) é chamado de operador de resolução de escopo. Trata-se de uma

forma de especificar a qual classe a função-membro *somaMedida()* está vinculada. Portanto, fazer:

```
Medida::somaMedida (Medida m1)
```

como na linha 35, significa que a função *somaMedida()* é uma função-membro da classe *Medida*.

Adicionalmente, perceba que essa função *somaMedida()*, definida fora do escopo da classe, retorna um dado após terminar seu processamento. Trata-se do objeto temporário chamado *temp*. Esse objeto (*m3*) retornado pela função *somaMedida()* é do tipo *Medida* (e, portanto, um objeto) que resulta da soma dos valores de dois outros objetos (*m1* e *m2*). Note ainda que o objeto *temp* da classe *Medida* é retornado pela função *somaMedida()* à função *main()* utilizando a instrução da linha 45:

```
return temp;
```

Com isso, o dado retornado é atribuído ao objeto *m3*. Observe que *m2* não sofre qualquer modificação, uma vez que ele apenas provê dado para a função *somaMedida()*.

6.7. ABSTRAÇÃO E ENCAPSULAMENTO

6.7.1. Abstração

O uso da abstração serve como uma forma de lidar com situações de complexidade. Nesse sentido, você está ocultando detalhes e focalizando aspectos essenciais. Em outras palavras, quando utiliza abstração, você procura apenas assimilar o essencial ignorando os detalhes.

6.7.2. Encapsulamento

Em programação orientada a objetos, há uma relação direta entre abstração e encapsulamento. Encapsulamento, juntamente com ocultação de dados, é uma característica importante em linguagens orientadas a objetos. Especificamente, o encapsulamento permite delimitar o escopo de uma entidade (isto é, uma classe) definindo o que está dentro e fora da entidade.

Em uma classe, há dados e funções-membros. Os dados e as funções são encapsulados em uma única entidade denominada objeto. Portanto, podemos dizer que uma classe, e mais especificamente um objeto, tem a propriedade de encapsular dados e operações (ou funções-membros) que atuam sobre esses dados. Para entender mais, vamos examinar o próximo exemplo.

Praticando um Exemplo. Escreva um programa em C++ que cria uma classe chamada Retangulo, a qual possui dois itens de dado: largura e comprimento. A classe Retangulo tem quatro funções: você deve ter uma função-membro atribuir() que recebe os parâmetros de largura e comprimento do retângulo e os atribui aos respectivos dados. A classe *Retangulo* tem outras duas funções: *getComprimento()* e *getLargura()* que retornam comprimento e largura do retângulo, respectivamente. E, além disso, você deve ter a classe *calculaArea()* que obtém a área do retângulo. Para elaborar esse programa, você deve criar uma classe que atenda a esses requisitos e criar um objeto *retangulo* (*r1*). Note que você deve criar essas funções-membros e chamá-las a partir da função *main()* juntamente com a criação do objeto da classe *Retangulo*. Agora, feche o livro e tente implementar sua solução. Depois, consulte a solução do exemplo mostrada na Listagem 6.6.

```
1. #include <iostream>
2. using namespace std;
3. // Programa para ilustrar encapsulamento de objeto
4. class Retangulo // Especificacao da classe Retangulo
5. {
6.     private:
7.         double largura;
8.         double comprimento;
9.     public:
10.        Retangulo ()
11.        {
12.            atribuir(0, 0);
13.        }
14.        Retangulo(double l, double c) // define retangulo
15.        {
16.            atribuir(l, c);
17.        }
18.        double getLargura() // obtem medida do usuario
19.        {
20.            return largura;
21.        }
22.        double getComprimento()
23.        {
24.            return comprimento;
25.        }
26.        double calculaArea()
27.        {
28.            return largura * comprimento;
29.        }
30.        void atribuir(double l, double c);
31.};
```

```

32. void Retangulo::atribuir(double l, double c)
33. {
34.     largura = l;
35.     comprimento = c;
36. }
37.
38. int main()
39. {
40.     Retangulo r1; // cria um objeto Retangulo
41.     double l, c;
42.     cout << "\nDigite o valor da largura do retangulo: ";
43.     cin >> l;
44.     cout << "\nDigite o valor do comprimento do retangulo: ";
45.     cin >> c;
46.
47.     r1.atribuir(l, c);
48.     cout << "\nArea do retangulo = " << r1.calculaArea()
         << endl << endl;
49.     system("PAUSE");
50.     return 0;
51. }

```

Listagem 6.6

O programa da Listagem 6.6 ilustra o encapsulamento da função *calculaArea()* que recebe dois parâmetros e retorna o valor da área de um retângulo. Perceba que os dados (largura e comprimento) e a implementação da função-membro *calculaArea()* estão encapsulados na classe *Retangulo*, e apenas um objeto retângulo pode ter acesso àquela funcionalidade fazendo uso de alguma função-membro que tenha acesso público. Execute o programa e digite, por exemplo, o valor 15 para largura e 25 para comprimento e, depois, o programa exibirá a saída mostrada na Figura 6.7.

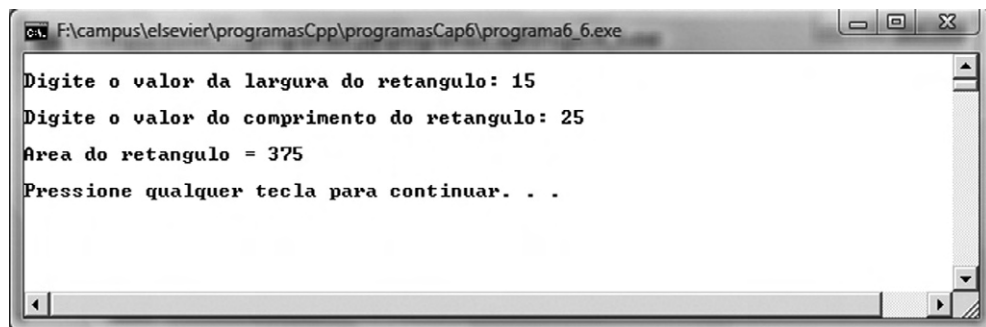


Figura 6.7 – Saída do programa da Listagem 6.6.

No programa da Listagem 6.6, os valores de *l* (largura) e *c* (comprimento) são lidos nas linhas 43 e 45; na linha 47, a função *main()* envia a mensagem *atribuir(l, c)* para o objeto *r1*. Os parâmetros da mensagem são as variáveis *l* e *c*. O objeto responde chamando a função-membro *Rectangle::atribuir(Double, double)* na linha 32. A linha 48 exibe a área do retângulo do referido objeto *r1*. Para tanto, objeto *r1* chama a função *calculaArea()*, linhas 26-29, que retorna o valor calculado para a função *main()*.

■ *Note que, se o programador precisar modificar o(s) dado(s) de um objeto, ele terá de saber exatamente quais funções interagem com aquele objeto (isto é, funções-membros). Outras funções não podem ter acesso ao(s) dado(s). Como resultado, obtém-se uma simplificação da escrita, bem como depuração e manutenção do programa.*

6.8. DADOS DE CLASSE STATIC

Até aqui, você aprendeu que cada objeto possui seu(s) próprio(s) dado(s), porém é importante observar que se um item de dado de uma classe for definido como *static* apenas um único item é criado para toda a classe, não importando quantos objetos existam.

6.8.1. Dado-membro static

Dessa forma, um item de dado *static* é útil quando todos os objetos da mesma classe compartilham um item de dado comum (de informação). Note que uma variável (ou item de dado-membro) definida como *static* tem características similares a uma variável *static* normal. Nesse caso, ela é visível apenas dentro da classe, porém seu tempo de vida é o do programa (isso foi discutido no Capítulo 5).

Todavia, enquanto uma variável *static* normal é usada para reter informação entre chamadas a uma função, dados-membros *static* de uma classe são usados para compartilhar informação entre os objetos de uma classe.

Agora, você pode questionar: onde posso desejar usar dados-membros *static*?

Imagine um jogo e que você tenha interesse em saber quantos jogadores ainda restam em uma competição. Perceba que essa situação é similar à condição de um objeto que precisa saber quantos dos demais objetos fazem parte de uma classe. Nesse caso, uma variável *static* contador pode ser usada como um membro da classe. Todos os objetos teriam acesso a essa variável. Assim, todos veriam a mesma variável, desde que ela é a mesma para todos. Para entender melhor, vejamos um exemplo.

Praticando um Exemplo. Escreva um programa em C++ que cria uma classe chamada *ContagemJogo*, a qual possui um único item de dado, *contador*, e você declara esse dado como *static*. A classe *ContagemJogo* tem uma função *getContador()* que retorna o valor do contador toda vez que é chamada. Para elaborar esse programa, você deve criar uma classe que atenda a esses requisitos e criar um objeto *ContagemJogo* (*c1*). Note que você deve criar a função-membro e chamá-la a partir da função *main()* juntamente com a criação do objeto da classe *ContagemJogo*. Agora, feche o livro e tente implementar sua solução. Depois, consulte a solução do exemplo mostrada na Listagem 6.7.

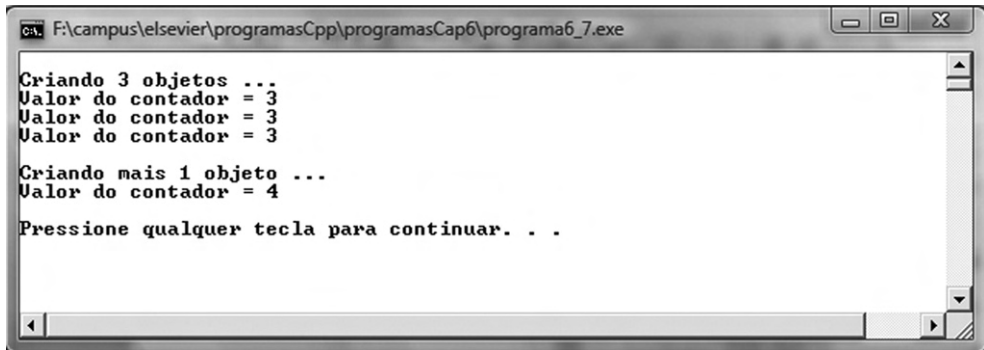
```

1. #include <iostream>
2. using namespace std;
3. // Programa para ilustrar uso de static para item de dado
4.
5. class ContagemJogo
6. {
7.     private:
8.         static int contador; // unico item de dado para todos ob-
          jetos
9.     public:
10.         ContagemJogo() { contador++; } // incrementa contador apos
          criacao de objeto
11.         int getContador() { return contador; } // obtem valor atual
          de contador
12. };
13. int ContagemJogo::contador;
14.
15. int main()
16. {
17.     ContagemJogo c1, c2, c3; //, c4; // cria 3 objetos ContagemJogo
18.     cout << "\nCriando 3 objetos... ";
19.     cout << "\nValor do contador = " << c1.getContador();
          // Todos objetos
20.     cout << "\nValor do contador = " << c2.getContador();
          // veem mesmo valor
21.     cout << "\nValor do contador = " << c3.getContador() << endl;
22.
23.     ContagemJogo c4;
24.     cout << "\nCriando mais 1 objeto... ";
25.     cout << "\nValor do contador = " << c4.getContador()
          << endl << endl;
26.     system("PAUSE");
27.     return 0;
28. }

```

Listagem 6.7

O programa da Listagem 6.7 ilustra o uso de *static* para definir a categoria de armazenamento de um item de dado. Execute o programa e, depois, o programa exibirá a saída na Figura 6.8.



```
ca. F:\campus\elsevier\programasCpp\programasCap6\programa6_7.exe
Criando 3 objetos ...
Valor do contador = 3
Valor do contador = 3
Valor do contador = 3
Criando mais 1 objeto ...
Valor do contador = 4
Pressione qualquer tecla para continuar. . .
```

Figura 6.8 – Saída do programa da Listagem 6.7.

A classe *ContagemJogo* da Listagem 6.7 possui um item de dado do tipo *static int*. O construtor para essa classe faz *contador* ser incrementado na linha 10. Em *main()*, três objetos (*c1*, *c2* e *c3*) da classe *ContagemJogo* são criados. Uma vez que o construtor é chamado três vezes nas linhas 19, 20 e 21, o item de dado *contador* é incrementado três vezes. Assim, quando a função-membro *getContador()* é chamada, ela retorna o valor de *contador*, isto é, 3 como mostrado na Figura 6.8.

As variáveis de classe do tipo *static* não são usadas frequentemente, mas são importantes em situações especiais. Portanto, entendê-las esclarece mais ainda como as variáveis comuns (*automatic*) podem ser utilizadas.

Variáveis comuns são declaradas (isto é, o compilador é informado de seu nome e tipo) e definidas (quando o compilador aloca memória para a variável) na mesma instrução.

■ Note que um dado-membro *static* exige duas instruções separadas. A declaração da variável aparece no especificador da classe, porém a variável é realmente definida fora da classe, similarmente a uma variável externa.

É importante ressaltar que, se um dado-membro *static* for definido dentro da especificação da classe, ele estará violando a ideia de que uma especificação de classe é apenas uma declaração e não aloca qualquer memória. Colocar a definição de dado-membro *static* fora da classe também serve para enfatizar que o espaço de

memória para esse dado é alocado apenas uma vez no início do programa e que uma variável-membro *static* é acessada por toda a classe.

RESUMO

Neste capítulo, você aprendeu sobre classes e objetos. Considerando os programas deste capítulo e aqueles do Capítulo 4, você perceberá que é possível fazer a mesma coisa com estruturas em uma linguagem procedimental. Um benefício que pode ser vislumbrado é a relação próxima que se tem entre as coisas do mundo real (que estão sendo modeladas por um programa) e os objetos C++ no programa. O objeto *Conta* representa uma conta bancária, tornando mais fácil conceituar um problema de programação. Assim, você abstrai das partes do problema que podem ser representadas como objetos. Em algumas situações, pode não ser óbvio que partes de uma problema/situação do mundo real devem ser objetos. Não existem regras rigorosas para esse tipo de análise. Uma forma de proceder é a *tentativa e erro*, pela qual você divide o problema em objetos e depois escreve os especificadores de classe para esses objetos. Se as classes parecem estar casando com a realidade, você continua. Do contrário, você precisará reiniciar o processo, identificando novas entidades como classes. No entanto, à medida que fica mais experiente com a abordagem/programação orientada a objetos, o processo de dividir o problema de programação em classes se tornará mais fácil. No próximo capítulo, você irá estudar e explorar uso de *arrays* e como eles podem ser usados na programação orientada a objetos.

QUESTÕES

1. Qual a diferença entre objetos e classes? Use exemplos para ilustrar sua resposta.
2. O que é polimorfismo? Em que situação é apropriado fazer uso de polimorfismo?
3. O que são construtores e destrutores? Use um exemplo para ilustrar sua resposta.
4. O que é encapsulamento? Em que situações pode ser empregado? Use um exemplo para ilustrar sua resposta.

EXERCÍCIOS

1. Faça uma pesquisa visando responder à seguinte questão: em quais situações é apropriado definir dados-membros de uma classe como *static*? Apresente um exemplo para ilustrar sua resposta.
2. Escreva um programa que implemente a classe *ContaCorrente*, que deve possuir funções-membros para permitir ao usuário fazer depósito, saque e consulta de saldo.

3. Escreva um programa que implemente a classe Estudante, que tem as funções-membros `getNome`, `getNota`, `mostraNota`, `calculaMedia`.
4. Escreva um programa que implemente a classe Retangulo, que deve possuir funções-membros para permitir ao usuário entrar com medidas (largura e altura) do retângulo, além de exibir a área do retângulo.
5. Escreva uma classe que implemente a classe Distancia, que possui o membro (`getDados`) que obtém dados das coordenadas de dois pontos `p1` e `p2` e depois retorna a distância (função `mostraDados()`) entre esses dois pontos.