

## Capítulo 9

# Herança

*Reality is merely an illusion, albeit a very persistent one.*

Albert Einstein

### OBJETIVOS

- Entender e saber como utilizar herança.
- Aprender a especificar classes base e derivada.
- Conhecer e explorar a hierarquia de classes.
- Entender e usar especificadores `public`, `protected` e `private`.

No capítulo anterior, você aprendeu o que é sobrecarga de operadores, funções `friend` e como utilizá-las. Depois do conceito de classes e objetos, herança é, provavelmente, o aspecto mais importante em programação orientada a objetos (POO). Herança é o processo de criar novas classes (classes derivadas) a partir das classes existentes (classes-base). A classe derivada herda todas as propriedades da classe-base e, além disso, adiciona novas características ou faz refinamentos. Herança é um ingrediente importante em POO, pois permite a reusabilidade de código. Uma vez que uma classe-base esteja escrita e depurada, não precisará ser alterada novamente. Todavia, ela pode ser adaptada para ser usada em situações diferentes. Reusar o código existente economiza tempo e dinheiro. Adicionalmente, herança pode auxiliar na modelagem e solução de um problema (de programação) e no projeto do programa. Este capítulo explora esse tópico e busca responder a questões: como herdar propriedade de uma classe? Como definir hierarquia de classes? É possível obter herança múltipla? Responder a essas e outras questões é o objetivo deste capítulo e, para tanto, os exemplos usados ilustram situações em que usar herança é apropriado.

## 9.1. INTRODUÇÃO

### 9.1.1. Herança

Juntamente com o conceito de classes e objetos, herança é, provavelmente, o aspecto mais importante em programação orientada a objetos (POO). Herança permite criar novas classes, chamadas de classes derivadas ou subclasses, a partir das classes existentes, conhecidas como classes base.

### 9.1.2. Classe Derivada

A classe derivada herda todas as propriedades da classe base e, portanto, especializa a classe base, já que ela adiciona novas características ou faz refinamentos sobre a classe base. Observe que, nesse processo, a classe base permanece inalterada.

### 9.1.3. Importância da Herança

Herança é uma característica importante em POO, pois oferece suporte para reusabilidade. Note que o reuso se dá não apenas no código já escrito e testado, mas também no componente implementado (que não requer nova implementação e, portanto, pode ser reutilizado). Observe ainda que uma classe já implementada pode também ser adaptada para ser usada em situações diferentes. Reusar o código existente economiza tempo e dinheiro. Adicionalmente, herança pode auxiliar na modelagem e solução de um problema (de programação) e no projeto do programa.

### 9.1.4. Reusabilidade

Como resultado da reusabilidade, tem-se facilidade para distribuir biblioteca de classes. Assim, um programador pode utilizar uma classe criada por outra pessoa sem ter necessidade de modificá-la, bastando derivar outras classes que sejam adequadas para uma situação específica. Veremos esses tópicos a seguir.

## 9.2. CLASSES BASE E DERIVADA

Anteriormente, no Capítulo 8, você implementou a classe *Contador*. Agora você precisa implementar o operador de decremento, além do operador de incremento (já implementado) para permitir, por exemplo, contar o número de pessoas que entram e saem de uma biblioteca.

Em tal situação, você pode inserir uma rotina de decremento no código da classe *Contador*. Contudo, desde que essa classe já esteja funcionando, e supondo que gastamos

muito tempo depurando-a, é natural que você não queira alterá-la (pois isso implica a necessidade de depurá-la e testá-la novamente, ou seja, mais tempo a ser gasto).

A fim de evitar problemas dessa natureza, você pode usar herança para criar uma nova classe baseada em *Contador*. Para entender melhor, vamos examinar o exemplo seguinte.

**Praticando um Exemplo.** Modifique o programa da Listagem 8.1 que implementa a classe *Contador*. Note que essa classe define o construtor *Contador()*, a função *getContador()* e sobrecarrega o operador de incremento (++). Entretanto, você precisa adicionar uma nova classe *Decrementa*, derivada da classe *Contador*, utilizando a seguinte sintaxe:

```
class Decrementa : public Contador
```

Essa classe deve sobrecarregar o operador de decremento usando a declaração *Contador operator -- ()*. Seu programa deve criar um objeto *c1*. *c1* é incrementado quatro vezes e decrementado três vezes. Em seguida, o valor de *c1* é mostrado, chamando a função *getContador()* a partir de *main()*. Agora, feche o livro e tente implementar sua solução. Depois, consulte a solução do exemplo mostrada na Listagem 9.1.

```
1. #include <iostream>
2. using namespace std;
3. // Programa para ilustrar uso de heranca
4.
5. class Contador // Especificacao da classe base Contador
6. {
7.     protected:
8.         unsigned int contador;
9.     public:
10.        Contador() // construtor sem argumento
11.        {
12.            contador = 0;
13.        }
14.        Contador(int c) // construtor com 1 argumento
15.        {
16.            contador = c;
17.        }
18.        int getContador() // retorna contador
19.        {
20.            return contador;
21.        }
22.        Contador operator ++ () // pre-incremento de contador
23.        {
24.            return Contador(++contador);
```

```
25.     }
26. };
27. class Decrementa: public Contador // classe derivada
28. {
29.     public:
30.         Contador operator -- ()      // decremento de contador
31.         {
32.             return Contador(--contador);
33.         }
34. };
35. int main()
36. {
37.     Decrementa c1; // cria 1 objeto Decrementa
38.     cout << "\nValor inicial do objeto contador c1 = " <<
        c1.getContador();
39.
40.     ++c1; ++c1; ++c1; ++c1; // incrementa c1 4 vezes
41.     cout << "\nValor depois do incremento de c1 = " <<
        c1.getContador();
42.
43.     --c1; --c1; --c1;      // decrementa c1 3 vezes
44.     cout << "\nValor depois do decremento de c1 = " <<
        c1.getContador();
45.     cout << endl << endl;
46.     system("PAUSE");
47.     return 0;
48. }
```

### Listagem 9.1

O programa da Listagem 9.1 define a classe *Contador* e cria um objeto *Contador* chamado de *c1*. O objeto *c1* é incrementado quatro vezes e decrementado três vezes. Em seguida, o valor de *c1* é mostrado. Execute o programa e, depois, o programa exibirá a saída mostrada na Figura 9.1.

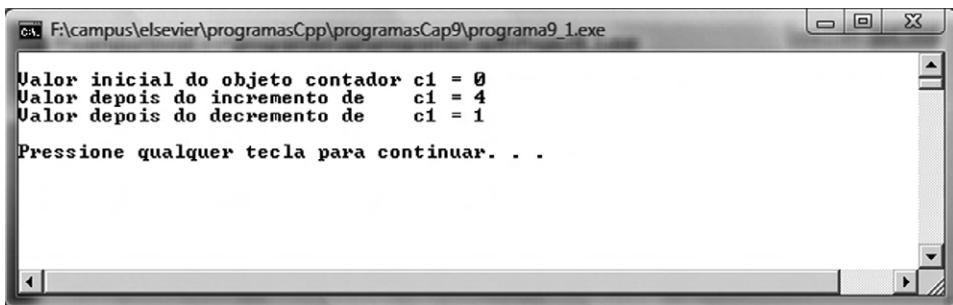


Figura 9.1 – Saída do programa da Listagem 9.1.

O programa da Listagem 9.1 ilustra a derivação de uma nova classe que adiciona o operador de decremento. Analisando esse programa, você tem a especificação de uma nova classe *Decrementa* que incorpora a função *operator--()* (para decremento).

Note, contudo, que a nova classe herda todas as características da classe *Contador*, ou seja, *Decrementa* não precisa de um construtor ou *getContador()* ou o *operator++()*, uma vez que essas funções já existem na classe *Contador*. Na linha 27 da listagem, a instrução

```
class Decrementa: public Contador
```

diz que a classe *Decrementa* é derivada da classe *Contador*. O sinal de dois-pontos (:) é usado para estabelecer a relação entre classes. Essa relação diz que a classe derivada tem acesso à classe base, porém o inverso não ocorre.

Outra questão importante em herança é saber quando uma função-membro na classe base pode ser usada pelos objetos da classe derivada. Isso é denominado *acessibilidade*. No programa anterior, um objeto da classe *Decrementa* (*c1*) é criado, na linha 37, com a instrução:

```
CountDn c1;
```

Essa instrução cria o objeto *c1* e ele é inicializado para 0. Como isso ocorre?

Não existe construtor na classe *Decrementa* e, nesse caso, a classe derivada usa um construtor apropriado da classe base (construtor sem argumento). Tal flexibilidade por parte do compilador (de usar outra função quando uma não está disponível) ocorre geralmente em situações de herança.

Ainda no programa anterior, você pôde aumentar a funcionalidade de uma classe sem modificá-la. Ou melhor, quase sem modificá-la. Perceba que o especificador de acesso aos dados da classe é agora *protected*. O que isso significa?

Lembre-se de que os membros de uma classe (dados e funções) podem ser acessados por funções dentro da própria classe, sejam eles *private* ou *public*. Contudo, objetos de uma classe definidos fora da classe podem ter acesso apenas aos membros *public*.

Agora, quando fazemos uso de herança, temos mais detalhes que precisam ser considerados. Por exemplo, as funções-membros da classe derivada podem ter acesso aos membros da classe base?

A resposta é sim, mas apenas se os membros da classe base são especificados como *public* ou *protected*. Eles não podem ter acesso a membros *private*.

No programa anterior, a variável *contador* não é *public*, já que isso permitiria a ela ser acessada por qualquer função no programa e eliminaria as vantagens de ocultação de dados (*data hiding*). A variável *contador* é declarada como um membro *protected*

(linhas 7 e 8) e pode ser acessada pelas funções-membros da própria classe, bem como da classe derivada.

A Tabela 9.1 ilustra a acessibilidade de membros para os especificadores `public`, `protected` e `private`.

Tabela 9.1

Especificador de acesso	Acessível da própria classe	Acessível da classe derivada	Acessível a objetos de fora da classe
<code>public</code>	sim	sim	sim
<code>protected</code>	sim	sim	não
<code>private</code>	sim	não	não

Nesse caso, quando escolher um dos especificadores possíveis? Quando estiver escrevendo uma classe e suspeitar de que pode precisar usar essa classe no futuro, como uma classe base para outras classes. Em tal situação, você deve utilizar o especificador *protected* em vez de *private*.

---

■ É importante observar que herança não funciona no sentido inverso. Isto é, a classe base desconhece a existência da(s) classe(s) derivada(s).

É importante também mencionar que existem desvantagens em fazer os membros de uma classe ser *protected*. Considere o caso em que você tenha escrito uma biblioteca de classes e distribua a mesma. Dessa forma, qualquer programador que venha a usar essa biblioteca poderá ter acesso aos membros *protected* da classe simplesmente derivando classes dela. Assim, é preciso ponderar quando e onde empregar *protected* como especificador de classes para nossos programas. Ou seja, quando e onde esse especificador trará vantagens no seu uso.

## 9.3. CONSTRUTORES DE CLASSE

### 9.3.1. Construtores de Classes Derivadas

Existe um defeito no programa *counten.cpp* (visto anteriormente). Qual é? O que ocorre se você inicializar um objeto *Decrementa* com um valor? O construtor com um argumento pode ser utilizado?

A resposta é não. Conforme visto no programa da Listagem 9.1, o compilador pode utilizar um construtor sem argumento da classe base, mas ele não o faz para construtores mais complexos.

Para fazer o programa funcionar corretamente, você precisa escrever um novo conjunto de construtores para classe derivada. Para entender mais, vamos examinar o próximo exemplo.

**Praticando um Exemplo.** Modifique o programa da Listagem 9.1, que implementa a classe *Contador*, adicionando dois novos construtores na classe *Contador*. O construtor sem argumento é chamado pela instrução:

```
Decrementa() : Contador()
{ }
```

Essa instrução faz o construtor *Decrementa()* chamar o construtor *Contador()* na classe base. Seu programa deve criar dois objetos *c1* e *c2*. *c1* é inicializado pelo construtor enquanto *c2* é inicializado no programa com a instrução *Decrementa c2(10)*. *c1* é incrementado quatro vezes e decrementado três vezes, enquanto *c1* é decrementado quatro vezes. Em seguida, os valores de *c1* e *c2* são mostrados, chamando a função *getContador()* a partir de *main()*. Agora, feche o livro e tente implementar sua solução. Depois, consulte a solução do exemplo mostrada na Listagem 9.2.

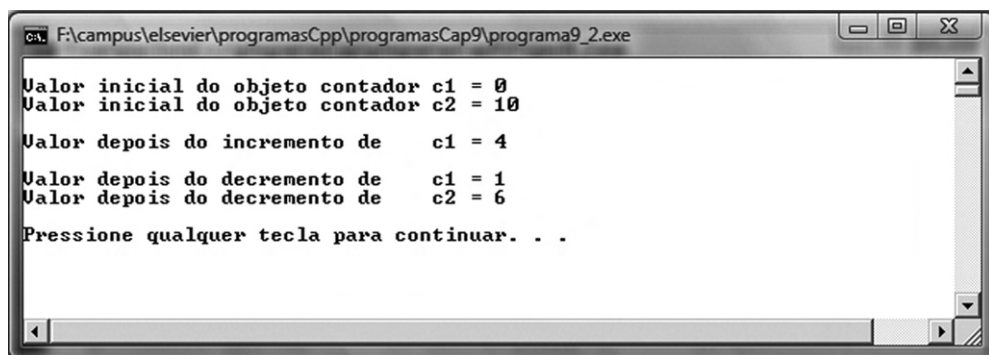
```
1. #include <iostream>
2. using namespace std;
3. // Programa para ilustrar uso de heranca
4.
5. class Contador // Especificacao da classe base Contador
6. {
7.     protected:
8.         unsigned int contador;
9.     public:
10.        Contador() // construtor sem argumento
11.        {
12.            contador = 0;
13.        }
14.        Contador(int c) // construtor com 1 argumento
15.        {
16.            contador = c;
17.        }
18.        int getContador() // retorna contador
19.        {
20.            return contador;
21.        }
22.        Contador operator ++ () // pre-incremento de contador
23.        {
24.            return Contador(++contador);
```

```
25.     }
26. };
27. class Decrementa: public Contador // classe derivada
28. {
29.     public:
30.         Decrementa(): Contador() // construtor sem argumentos
31.         { }
32.         Decrementa(int c): Contador(c) // construtor com 1 argumento
33.         { }
34.         Decrementa operator -- ()    // decremento de contador
35.         {
36.             return Decrementa(--contador);
37.         }
38. };
39. int main()
40. {
41.     Decrementa c1; // cria 1 objeto Decrementa
42.     Decrementa c2(10);
43.
44.     cout << "\nValor inicial do objeto contador c1 = " <<
         c1.getContador();
45.     cout << "\nValor inicial do objeto contador c2 = " <<
         c2.getContador();
46.     ++c1; ++c1; ++c1; ++c1; // incrementa c1 4 vezes
47.     cout << "\n\nValor depois do incremento de c1 = " <<
         c1.getContador();
48.
49.     --c1; --c1; --c1; // decrementa c1 3 vezes
50.     --c2; --c2; --c2; --c2; // decrementa c2 4 vezes
51.     cout << "\n\nValor depois do decremento de c1 = " <<
         c1.getContador();
52.     cout << "\nValor depois do decremento de c2 = " <<
         c2.getContador();
53.     cout << endl << endl;
54.     system("PAUSE");
55.     return 0;
56. }
```

### Listagem 9.2

O programa da Listagem 9.2 define a classe *Contador* e cria objetos *Contador* chamados de *c1* e *c2*. Novos construtores são implementados para a classe *Decrementa* derivada da classe *Contador*. Execute o programa e, depois, o programa exibirá a saída mostrada na Figura 9.2.





```

F:\campus\elsevier\programasCpp\programasCap9\programa9_2.exe
Valor inicial do objeto contador c1 = 0
Valor inicial do objeto contador c2 = 10
Valor depois do incremento de c1 = 4
Valor depois do decremento de c1 = 1
Valor depois do decremento de c2 = 6
Pressione qualquer tecla para continuar. . .
  
```

Figura 9.2 – Saída do programa da Listagem 9.2.

Adicionalmente, em *main()* a instrução *Decrementa c2(10)*; usa o construtor de um argumento em *Decrementa*. Esse construtor chama o correspondente construtor na classe base com a instrução:

```
Decrementa(int c): Contador(c)
{ }
```

Essa construção faz o argumento *c* ser passado de *Decrementa()* para *Contador()*, onde ele é usado para inicializar o objeto.

## 9.4. HERANÇA EM CLASSE

O uso e o tratamento de herança em classes oferecem recursos adicionais na solução de problemas. Nesse sentido, cabe destacar que você pode usar funções-membros em uma classe derivada com os mesmos nomes daquelas existentes na classe base. Isso pode ser feito para que os objetos operem da mesma forma com objetos das classes base e derivada.

### 9.4.1. Ignorando Funções-membros

Considere o programa da Listagem 7.6 do Capítulo 7, que modela uma pilha. Esse programa permitia tanto inserir quanto remover inteiros de uma pilha. Todavia, ele tinha um problema. Se, por acaso, você tentasse colocar muitos itens na pilha, isso resultava em problema porque poderia extrapolar o valor máximo do tamanho de pilha. O mesmo poderia ocorrer se você tentasse remover muitos itens da pilha.

Objetivando corrigir esse problema, vamos examinar o próximo exemplo, de modo a fazer uso da propriedade de herança para tratar essa questão.

**Praticando um Exemplo.** O programa da Listagem 7.6 implementa uma classe de *Pilha*. Essa classe deve possuir dois membros de dados: um array *aPilha[tamanhoArray]* e uma variável *topo* que serve para indicar o último elemento colocado na pilha. Para colocar um item na pilha, você deve chamar a função-membro *push()* com o valor armazenado como argumento. Para remover um item da pilha, você deve usar a função-membro *pop()*, a qual retorna o valor do item. Você deve modificar o programa da Listagem 7.6 criando uma nova classe *Pilha2* da classe *Pilha*. Os objetos de *Pilha2* se comportam da mesma forma que os de *Pilha*, exceto que o usuário será alertado se tentar colocar ou remover muitos itens (excedendo ao máximo) na pilha. Adicionalmente, a classe *Pilha* deve ser a mesma do programa anterior, exceto que seus dados-membros são agora *protected*. Já a classe *Pilha2* possui duas funções: *push()* e *pop()*. Essas funções têm os mesmos nomes, argumentos e tipos de retorno que as funções em *Pilha*.

Note que as funções *push()* e *pop()* são funções-membros da classe *Pilha* e devem ser chamadas a partir de *main()*. Agora, feche o livro e tente implementar sua solução. Depois, consulte a solução do exemplo apresentada na Listagem 9.3.

```
1. #include <iostream>
2. const int MAX = 3; // tamanho da pilha
3. using namespace std;
4. // Programa para ilustrar heranca de classe
5.
6. class Pilha
7. {
8.     protected:
9.         int aPilha[MAX]; // array pilha
10.        int topo; // elemento do topo da pilha
11.    public:
12.        Pilha() // construtor
13.        { topo = -1; }
14.        void push(int x) // push - colocar dado na pilha
15.        { aPilha[++topo] = x; }
16.        int pop() // pop - remove take um dado da pilha
17.        { return aPilha[topo--]; }
18. };
19.
20. class Pilha2: public Pilha
21. {
22.     public:
23.        void push(int x) // coloca dado na pilha
24.        {
25.            if(topo >= MAX - 1) // testa se pilha esta cheia
26.            {
```

```

27.     cout << "\nErro: a pilha esta cheia!\n\n";
28.     system("PAUSE"); exit(1);
29. }
30. Pilha::push(x); // chama push() da classe Pilha
31. }
32. int pop() // remove dado da pilha
33. {
34.     if(topo < 0) // testa se pilha esta vazia
35.     {
36.         cout << "\nErro: a pilha esta vazia!\n\n";
37.         system("PAUSE"); exit(1);
38.     }
39.     return Pilha::pop(); // chama pop() da classe Pilha
40. }
41. };
42. int main()
43. {
44.     Pilha2 p;
45.     cout << "Dados na pilha: " << endl << endl;
46.     p.push(100); // coloca dados na pilha
47.     p.push(200);
48.     p.push(300);
49.
50.     cout << "1: " << p.pop() << endl; // remove dados da pilha
51.     cout << "2: " << p.pop() << endl;
52.     cout << "3: " << p.pop() << endl;
53.
54.     p.push(400);
55.     p.push(500);
56.     p.push(600);
57.     p.push(700); // causa erro por pilha cheia
58.     cout << "4: " << p.pop() << endl;
59.     cout << "5: " << p.pop() << endl << endl;
60.     system("PAUSE");
61.     return 0;
62. }

```

### Listagem 9.3

O programa da Listagem 9.3 define as classes *Pilha* e *Pilha2* e cria um objeto *Pilha2* chamado de *p*. Esse programa ilustra o funcionamento de uma pilha e a sobrecarga de funções nas classes base e derivada. Execute o programa e, depois, o programa exibirá a saída mostrada na Figura 9.3.

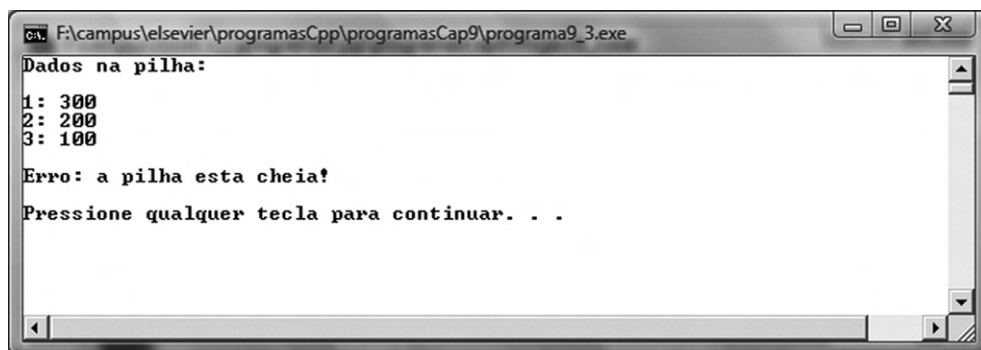


Figura 9.3 – Saída do programa da Listagem 9.3.

Perceba que, nesse programa, a classe *Pilha* é a mesma do programa da Listagem 7.6, exceto que seus dados-membros são agora *protected*. Já a classe *Pilha2* possui duas funções: *push()* e *pop()*. Essas funções têm os mesmos nomes, argumentos e tipos de retorno que as funções em *Pilha*. Então, como o compilador sabe a quem deverá chamar? Quando uma função existe tanto na classe base quanto na derivada, a função da classe derivada sempre é chamada. Dessa forma, diz-se que a função da classe derivada ignora a função da classe base.

---

■ *Note que, quando uma função existe tanto na classe base quanto na derivada, a função da classe derivada sempre é chamada. Dessa forma, diz-se que a função da classe derivada ignora a função da classe base.*

Perceba que fazemos uso do operador de resolução de escopo (::) nas instruções das linhas 30 e 39, respectivamente, como mostrado a seguir:

```
Pilha::push(x);
```

e

```
return Pilha::pop();
```

Essas instruções especificam que as funções *push()* e *pop()* em *Pilha* devem ser chamadas.

## 9.5. HIERARQUIA E NÍVEIS DE CLASSE

Nos exemplos vistos até o momento, herança tem sido utilizada para adicionar funcionalidade a uma classe já existente. Agora, vamos examinar um exemplo em que herança é usada como uma parte do projeto de um programa. Para entender mais, vamos examinar o exemplo a seguir.

**Praticando um Exemplo.** Escreva um programa que implemente uma classe de *Funcionario*. Nesse exemplo há apenas três tipos de funcionários: *Gestor*, *Professor* e *Técnico*. Esses três tipos de funcionários são subclasses da classe *Funcionario*. A base de dados desse programa armazena o nome e a identificação de todos os empregados (não importando a categoria). Para os gestores, ela também armazena dados de seus cargos e salários. Para os professores, a classe também armazena a quantidade de cursos lecionados. Os técnicos não precisam de dados adicionais. Você deve implementar as funções *lerDados()* e *mostraDados()*, para leitura e exibição, respectivamente, de todos os dados dos funcionários. Note que as funções *lerDados()* e *mostraDados()* são funções-membros da classe *Funcionario* e devem ser chamadas a partir de *main()*. Para testar o programa, você deve criar objetos *Gestor*, *Professor* e *Técnico*. Agora, feche o livro e tente implementar sua solução. Depois, consulte a solução do exemplo apresentada na Listagem 9.4.

```

1. #include <iostream>
2. using namespace std;
3.
4. // Programa para ilustrar heranca de classe
5. const int MAX = 100; // quantidade maxima de nomes
6.
7. class Funcionario
8. {
9.     private:
10.         char nome[MAX]; // nome do funcionario
11.         unsigned long id; // No. de identificacao
12.     public:
13.         void lerDados()
14.         {
15.             cout << "\nDigite seu nome: ";
16.             cin >> nome;
17.             cout << "Digite seu numero de identificacao: ";
18.             cin >> id;
19.         }
20.         void mostraDados()
21.         {
22.             cout << "\nNome: " << nome;
23.             cout << "\nIdent.: " << id;
24.         }
25. };
26. class Gestor: public Funcionario
27. {
28.     private:
29.         char cargo[MAX];
30.         double salario;
31.     public:
32.         void lerDados()

```

```
33.     {
34.         Funcionario::lerDados();
35.         cout << "Informe seu cargo: ";
36.         cin >> cargo;
37.         cout << "Informe seu salario: ";
38.         cin >> salario;
39.     }
40.     void mostraDados()
41.     {
42.         Funcionario::mostraDados();
43.         cout << "\nCargo: " << cargo;
44.         cout << "\nSalario do gestor R$: " << salario;
45.     }
46. };
47. class Professor: public Funcionario
48. {
49.     private:
50.         int cursos; // quantidade de cursos
51.     public:
52.         void lerDados()
53.         {
54.             Funcionario::lerDados();
55.             cout << "Informe a quantidade de cursos: ";
56.             cin >> cursos;
57.         }
58.         void mostraDados()
59.         {
60.             Funcionario::mostraDados();
61.             cout << "\nQuantidade de cursos oferecidos: " << cursos;
62.         }
63. };
64. class Tecnico: public Funcionario
65. { };
66. int main()
67. {
68.     Gestor g1, g2;
69.     Professor p1;
70.     Tecnico t1;
71.     cout << "\n\nInforme dados do gestor 1: ";
72.     g1.lerDados();
73.     cout << "\nInforme dados do gestor 2: ";
74.     g2.lerDados();
75.     cout << "\nInforme dados do professor 1: ";
76.     p1.lerDados();
77.     cout << "\nInforme dados do tecnico 1: ";
78.     t1.lerDados();
79.
80.     cout << "\nDados do gestor 1";
```

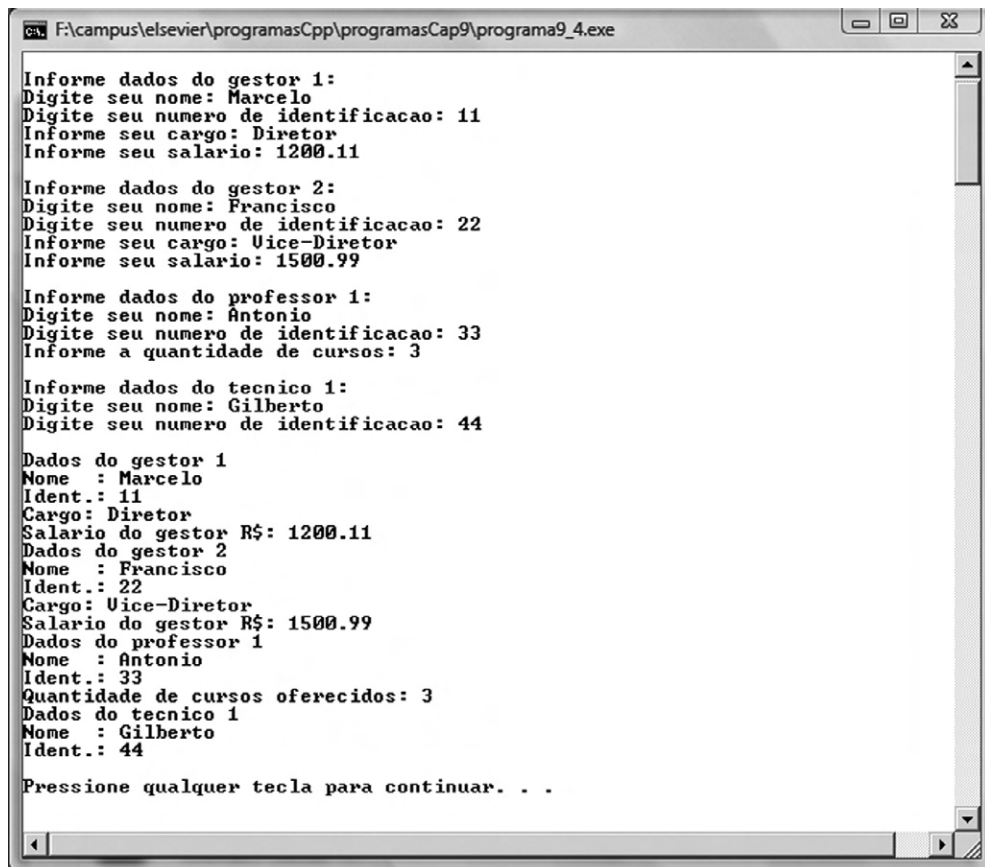
```

81. g1.mostraDados();
82. cout << "\nDados do gestor 2";
83. g2.mostraDados();
84. cout << "\nDados do professor 1";
85. p1.mostraDados();
86. cout << "\nDados do tecnico 1";
87. t1.mostraDados();
88. cout << endl << endl;
89. system("PAUSE");
90. return 0;
91. }

```

#### Listagem 9.4

O programa da Listagem 9.4 define as classes *Funcionario* e subclasses *Gestor*, *Professor* e *Técnico*. O programa modela uma base de dados de funcionários, solicitando um conjunto de dados e depois exibindo essas informações. Esse programa serve para ilustrar a hierarquia de classes. Execute o programa e, depois, o programa exibirá a saída mostrada na Figura 9.4.



```

F:\campus\elsevier\programasCpp\programasCap9\programa9_4.exe
Informe dados do gestor 1:
Digite seu nome: Marcelo
Digite seu numero de identificacao: 11
Informe seu cargo: Diretor
Informe seu salario: 1200.11

Informe dados do gestor 2:
Digite seu nome: Francisco
Digite seu numero de identificacao: 22
Informe seu cargo: Vice-Diretor
Informe seu salario: 1500.99

Informe dados do professor 1:
Digite seu nome: Antonio
Digite seu numero de identificacao: 33
Informe a quantidade de cursos: 3

Informe dados do tecnico 1:
Digite seu nome: Gilberto
Digite seu numero de identificacao: 44

Dados do gestor 1
Nome : Marcelo
Ident.: 11
Cargo: Diretor
Salario do gestor R$: 1200.11
Dados do gestor 2
Nome : Francisco
Ident.: 22
Cargo: Vice-Diretor
Salario do gestor R$: 1500.99
Dados do professor 1
Nome : Antonio
Ident.: 33
Quantidade de cursos oferecidos: 3
Dados do tecnico 1
Nome : Gilberto
Ident.: 44

Pressione qualquer tecla para continuar. . .

```

Figura 9.4 – Saída do programa da Listagem 9.4.

■ *Note, ainda, que não há qualquer construtor nas classes base e derivada. Assim, o compilador cria objetos das várias classes automaticamente quando encontra uma definição como `Gestor g1, g2`; (linha 68). Isso é feito com o construtor default de `Gestor` chamando o construtor default da classe `Funcionario`.*

A listagem do programa começa com a classe base *Funcionario*, especificada nas linhas 7-25. Essa classe manipula os nomes e o número dos empregados. A partir dessa classe, três outras classes são derivadas: *Gestor*, *Professor* e *Técnico*. As classes *Gestor* e *Professor* adicionam dados e funções membros nas linhas 26-46 e 47-63, respectivamente, para manipular esses dados.

### 9.5.1. Classe Abstrata

Observe que nenhum objeto da classe *Funcionario* foi definido. Essa classe foi usada unicamente para derivar outras classes. Classes usadas apenas para derivar outras classes (como *Funcionario*) são, em geral, chamadas de classes abstratas. Isso significa que nenhum objeto dessa classe é criado.

## 9.6. HERANÇA COM PUBLIC E PRIVATE

### 9.6.1. Herança Usando public e private

C++ oferece uma variedade de formas de acesso aos membros de uma classe. Esse mecanismo de controle de acesso é dado pela forma pela qual as classes derivadas são declaradas. No programa visto anteriormente, você tinha a classe declarada fazendo uso do especificador *public*, como em:

```
class Gestor: public Funcionario
```

Qual o efeito da palavra-chave *public*?

*public* é um especificador de acesso. Ele serve para especificar que os objetos da classe derivada têm acesso às funções-membros da classe base. Outra alternativa é usar *private*. Nesse caso, os objetos da classe derivada não podem ter acesso às funções-membros da seção *public* da classe base. Existem diversas possibilidades de acesso. Para entender melhor, vamos examinar o seguinte exemplo.

```
1.  #include <iostream>
2.  using namespace std;
3.  // Programa para ilustrar uso de public e private na heranca
   de classe
4.
5.  class X // classe base
6.  {
```



```

7.     private:
8.         int dadoPrivadoX;
9.     protected:
10.        int dadoProtegidoA;
11.    public:
12.        int dadoPublicoX;
13. };
14.
15. class Y: public X
16. {
17.     public:
18.         void funcao()
19.         {
20.             int x;
21.             x = dadoPrivadoX; // erro: nao acessivel
22.             x = dadoProtegidoX;
23.             x = dadoPublicoX;
24.         }
25. };
26.
27. class Z: public X
28. {
29.     public:
30.         void funcao()
31.         {
32.             int x;
33.             x = dadoPrivadoX; // erro: nao acessivel
34.             x = dadoProtegidoX;
35.             x = dadoPublicoX;
36.         }
37. };
38.
39. int main()
40. {
41.     int x;
42.     Y objetoY;
43.     x = objetoY.dadoPrivadoX; // erro: nao acessivel
44.     x = objetoY.dadoProtegidoX; // erro: nao acessivel
45.     x = objetoY.dadoPublicoX; // Ok: X publico para Y
46.
47.     Z objetoZ;
48.     x = objetoZ.dadoPrivadoX; // erro: nao acessivel
49.     x = objetoZ.dadoProtegidoX; // erro: nao acessivel
50.     x = objetoZ.dadoPublicoX; // erro: nao acessivel
51.     system("PAUSE");
52.     return 0;
53. }

```

Esse programa especifica a classe base *X* com dados especificados em *private*, *protected* e *public*. Duas classe *Y* e *Z* são derivadas de *X*. *Y* é derivada usando-se *public* e *Z* é derivada usando-se *private*.

Como visto anteriormente, funções da classe derivada podem ter acesso a dados em *public* e *protected* na classe base. Objetos das classes derivadas não podem ter acesso a membros de *private* ou *protected* da classe base.

Adicionalmente, observe que objetos de classes derivadas (como *Y*) usando *public* podem ter acesso a membros *public* da classe base (*X*). Por outro lado, objetos de classes derivadas (*Z*) usando *private* não podem ter acesso a membros *public* da classe base (*X*). Note ainda que, se você não fornecer o especificador para uma classe, este será assumido como *private* (que é o default).

Agora, se você tentar executar o programa, não terá sucesso devido a essa restrição de acesso imposta no uso de dados-membros especificados como *private*.

Observe que, na Listagem 9.5, há situações de erro, conforme comentários nas linhas 21, 33, 43, 44, 45, 48, 49 e 50. Entretanto, se você comentar as linhas 7-10 da Listagem 9.5 e redefinir a classe *X*, como ilustrado no fragmento de código a seguir, com os dados-membros *dadoPrivadoX* e *dadoProtegidoX* especificados como *public*:

```
class X // classe base
{
// private:
// int dadoPrivadoX;
// protected:
// int dadoProtegidoX;
public:
    int dadoPrivadoX;
    int dadoProtegidoX;
    int dadoPublicoX;
};
```

e executar novamente o programa, não haverá mais erros, resultando na saída mostrada na Figura 9.5.

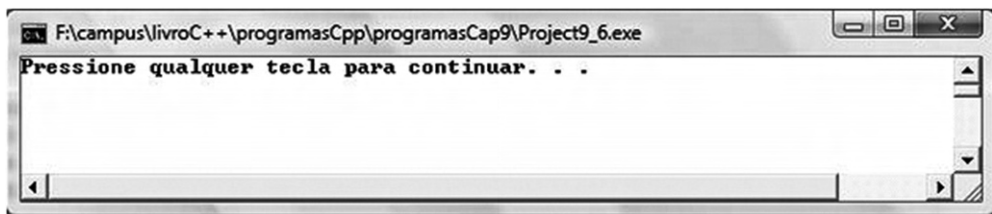


Figura 9.5 – Saída do programa da Listagem 9.5 (modificada).

### 9.6.2. Níveis de Herança

Classes podem ser derivadas de outras classes que já são classes derivadas. A seguir, você tem o fragmento de um programa que ilustra isso.

```
class X
{ };
class Y: public X
{ };
class Z: public Y
{ };
```

Nesse exemplo, *Y* é derivada de *X* e *Z* é derivada de *Y*. Esse processo pode ser estendido a um número arbitrário de níveis. Para entender mais, vamos examinar o próximo exemplo.

**Praticando um Exemplo.** Modifique o programa da Listagem 9.4 de modo a adicionar uma outra classe (*TecnicoAdm*) derivada da classe de *Tecnico*. Note que a hierarquia de classes resulta da generalização de características comuns. Assim, tem-se que a classe *Funcionario* é mais geral do que *Tecnico* e que a classe *Tecnico* é mais geral do que *TecnicoAdm*. Você deve ainda adicionar o dado-membro *horasExtras*, do tipo *double*, na classe *TecnicoAdm*, bem como suas funções *lerDados()* e *mostraDados()* para a classe, para leitura e exibição, respectivamente, dos dados de horas extras. Note que as funções *lerDados()* e *mostraDados()* são chamadas a partir de *main()*. Para testar o programa, você deve criar objetos *Gestor*, *Professor*, *Tecnico*, *TecnicoAdm*. Agora, feche o livro e tente implementar sua solução. Depois, consulte a solução do exemplo apresentada na Listagem 9.6.

```
1. #include <iostream>
2. using namespace std;
3. // Programa para ilustrar heranca de classe
4.
5. const int MAX = 100; // quantidade maxima de nomes
6.
7. class Funcionario
8. {
9.     private:
10.         char nome[MAX]; // nome do funcionario
11.         unsigned long id; // No. de identificacao
12.     public:
13.         void lerDados()
14.         {
15.             cout << "\nDigite seu nome: ";
16.             cin >> nome;
17.             cout << "Digite seu numero de identificacao: ";
```

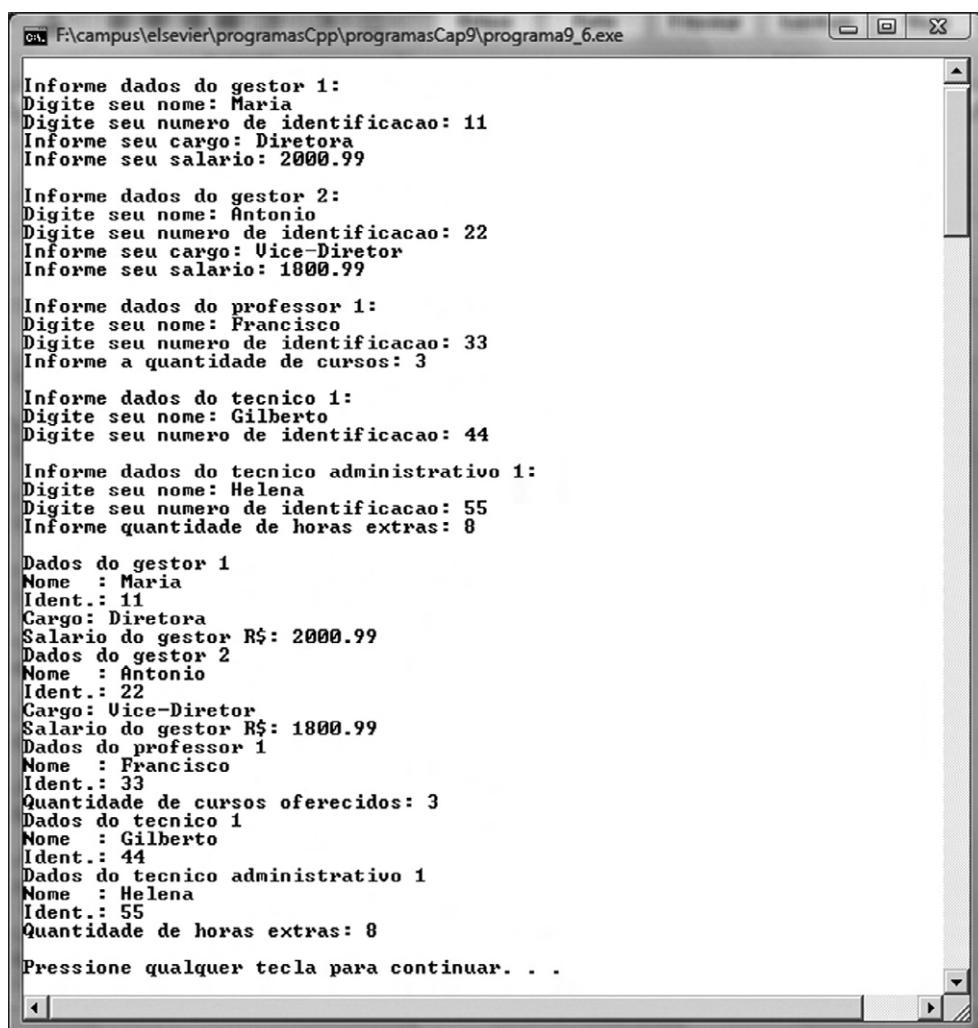
```
18.     cin >> id;
19.     }
20.     void mostraDados()
21.     {
22.         cout << "\nNome: " << nome;
23.         cout << "\nIdent.: " << id;
24.     }
25. };
26. class Gestor: public Funcionario
27. {
28.     private:
29.         char cargo[MAX];
30.         double salario;
31.     public:
32.         void lerDados()
33.         {
34.             Funcionario::lerDados();
35.             cout << "Informe seu cargo: ";
36.             cin >> cargo;
37.             cout << "Informe seu salario: ";
38.             cin >> salario;
39.         }
40.         void mostraDados()
41.         {
42.             Funcionario::mostraDados();
43.             cout << "\nCargo: " << cargo;
44.             cout << "\nSalario do gestor R$: " << salario;
45.         }
46. };
47. class Professor: public Funcionario
48. {
49.     private:
50.         int cursos; // quantidade de cursos
51.     public:
52.         void lerDados()
53.         {
54.             Funcionario::lerDados();
55.             cout << "Informe a quantidade de cursos: ";
56.             cin >> cursos;
57.         }
58.         void mostraDados()
59.         {
60.             Funcionario::mostraDados();
61.             cout << "\nQuantidade de cursos oferecidos: " << cursos;
62.         }
63. };
```

```
64. class Tecnico: public Funcionario
65. { };
66. class TecnicoAdm: public Tecnico // foreman class
67. {
68.     private:
69.         double horasExtras; // percent of quotas met successfully
70.     public:
71.         void lerDados()
72.         {
73.             Tecnico::lerDados();
74.             cout << "Informe quantidade de horas extras: "; cin >>
                horasExtras;
75.         }
76.         void mostraDados()
77.         {
78.             Tecnico::mostraDados();
79.             cout << "\nQuantidade de horas extras: " << horasExtras;
80.         }
81. };
82. int main()
83. {
84.     Gestor g1, g2;
85.     Professor p1;
86.     Tecnico t1;
87.     TecnicoAdm a1;
88.
89.     cout << "\n\nInforme dados do gestor 1: ";
90.     g1.lerDados();
91.     cout << "\nInforme dados do gestor 2: ";
92.     g2.lerDados();
93.     cout << "\nInforme dados do professor 1: ";
94.     p1.lerDados();
95.     cout << "\nInforme dados do tecnico 1: ";
96.     t1.lerDados();
97.     cout << "\nInforme dados do tecnico administrativo 1: ";
98.     a1.lerDados();
99.
100.     cout << "\nDados do gestor 1";
101.     g1.mostraDados();
102.     cout << "\nDados do gestor 2";
103.     g2.mostraDados();
104.     cout << "\nDados do professor 1";
105.     p1.mostraDados();
106.     cout << "\nDados do tecnico 1";
107.     t1.mostraDados();
108.     cout << "\nDados do tecnico administrativo 1";
```

```
109.      a1.mostraDados();  
110.      cout << endl << endl;  
111.      system("PAUSE");  
112.      return 0;  
113.      }
```

#### Listagem 9.6

O programa da Listagem 9.6 define as classes *Funcionario* e subclasses *Gestor*, *Professor*, *Tecnico* e *Tecnico.Adm*. O programa modela uma base de dados de funcionários, solicitando um conjunto de dados e depois exibindo essas informações. Esse programa ilustra níveis da herança de classes. Execute o programa e, depois, o programa exibirá a saída mostrada na Figura 9.6.



```
Informe dados do gestor 1:  
Digite seu nome: Maria  
Digite seu numero de identificacao: 11  
Informe seu cargo: Diretora  
Informe seu salario: 2000.99  
  
Informe dados do gestor 2:  
Digite seu nome: Antonio  
Digite seu numero de identificacao: 22  
Informe seu cargo: Vice-Diretor  
Informe seu salario: 1800.99  
  
Informe dados do professor 1:  
Digite seu nome: Francisco  
Digite seu numero de identificacao: 33  
Informe a quantidade de cursos: 3  
  
Informe dados do tecnico 1:  
Digite seu nome: Gilberto  
Digite seu numero de identificacao: 44  
  
Informe dados do tecnico administrativo 1:  
Digite seu nome: Helena  
Digite seu numero de identificacao: 55  
Informe quantidade de horas extras: 8  
  
Dados do gestor 1  
Nome : Maria  
Ident.: 11  
Cargo: Diretora  
Salario do gestor R$: 2000.99  
Dados do gestor 2  
Nome : Antonio  
Ident.: 22  
Cargo: Vice-Diretor  
Salario do gestor R$: 1800.99  
Dados do professor 1  
Nome : Francisco  
Ident.: 33  
Quantidade de cursos oferecidos: 3  
Dados do tecnico 1  
Nome : Gilberto  
Ident.: 44  
Dados do tecnico administrativo 1  
Nome : Helena  
Ident.: 55  
Quantidade de horas extras: 8  
  
Pressione qualquer tecla para continuar. . .
```

Figura 9.6 – Saída do programa da Listagem 9.6.

Note que a hierarquia de classes resulta da generalização de características comuns. Assim, tem-se que a classe *Funcionario* é mais geral do que *Tecnico* e que *Tecnico* é mais geral do que *Tecnico.Adm*.

## 9.7. HERANÇA MÚLTIPLA

Uma classe pode ser derivada de uma ou mais classes. A isso denominamos herança múltipla. A sintaxe para herança múltipla é similar àquela para herança simples (ou de uma única classe).

Para entender melhor, vamos examinar o seguinte exemplo. Considere uma classe *Z* derivada das classes *X* e *Y*. Nesse caso, essas duas classes (*X* e *Y*) são listadas após os dois-pontos (:) na especificação da classe *Z*, conforme ilustrado a seguir.

```
class X // classe base
{ };
class Y // classe base
{ };
class Z: public X, public Y
```

Considere a classe *Funcionario*, já especificada no exemplo da Listagem 9.4, e suponha que você deseje registrar dados de formação (educacional) de um conjunto de funcionários. No entanto, lembre-se de que você já tem a classe *Funcionario*. Vamos supor que haja uma classe chamada *Estudante* disponível. Então, em tal situação, você poderia adicionar esses novos dados através da herança múltipla da classe *Estudante* em vez de modificar a classe *Funcionario* incorporando os dados educacionais. A classe *Estudante* armazena o nome da universidade onde o estudante faz o curso e a titulação recebida (bacharel, especialista, mestre ou doutor). Além disso, duas funções *lerDadosEducacao()* e *mostraDadosEducacao()* solicitam essas informações do usuário e as mostram, respectivamente. A seguir, você tem um fragmento de programa que mostra as relações entre classes.

```
class Estudante
{ };
class Funcionario
{ };
class Gestor: private Funcionario, private Estudante
{ };
class Professor: private Funcionario, private Estudante
{ };
class Tecnico: public Funcionario
{ };
```

**Praticando um Exemplo.** Modifique o programa da Listagem 9.4 de modo a adicionar outra classe (*Estudante*). Depois derive as classes *Gestor* e *Professor* fazendo uso de herança múltipla das classes *Funcionario* e *Estudante*. As funções *lerDados()* e *mostraDados()* nas classes *Gestor* e *Professor* incorporam chamadas a funções na classe *Estudante*, como:

```
Estudante::lerDadosEducacao();
```

e

```
Estudante::mostraDadosEducacao();
```

Essas rotinas são acessíveis nas classes *Gestor* e *Gestor* porque essas classes são descendentes da classe *Estudante*. Adicionalmente, as funções *lerDados()* e *mostraDados()* devem ser chamadas a partir de *main()*. Para testar o programa, você deve criar objetos *Gestor*, *Professor* e *Técnico*. Agora, feche o livro e tente implementar sua solução. Depois, consulte a solução do exemplo apresentada na Listagem 9.7.

```
1. #include <iostream>
2. using namespace std;
3.
4. // Programa para ilustrar heranca de classe
5.
6. const int MAX = 100; // quantidade maxima de nomes
7.
8. class Estudante // educational background
9. {
10. private:
11.     char universidade[MAX]; // name of school or university
12.     char titulacao[MAX]; // highest degree earned
13. public:
14.     void lerDadosEducacao()
15.     {
16.         cout << "Informe o nome da universidade onde estudou: ";
17.         cin >> universidade;
18.         cout << "Informe o grau obtido \n";
19.         cout << " (bacharel, especialista, mestre, doutor): ";
20.         cin >> titulacao;
21.     }
22.     void mostraDadosEducacao()
23.     {
24.         cout << "\nUniversidade: " << universidade;
25.         cout << "\nGrau obtido: " << titulacao;
26.     }
27. };
28. class Funcionario
29. {
```



```
30. private:
31.     char nome[MAX]; // nome do funcionario
32.     unsigned long id; // No. de identificacao
33. public:
34.     void lerDados()
35.     {
36.         cout << "\nDigite seu nome: ";
37.         cin >> nome;
38.         cout << "Digite seu numero de identificacao: ";
39.         cin >> id;
40.     }
41.     void mostraDados()
42.     {
43.         cout << "\nNome: " << nome;
44.         cout << "\nIdent.: " << id;
45.     }
46. };
47. class Gestor: private Funcionario, private Estudante
48. {
49. private:
50.     char cargo[MAX];
51.     double salario;
52. public:
53.     void lerDados()
54.     {
55.         Funcionario::lerDados();
56.         cout << "Informe seu cargo: ";
57.         cin >> cargo;
58.         cout << "Informe seu salario: ";
59.         cin >> salario;
60.         Estudante::lerDadosEducacao();
61.     }
62.     void mostraDados()
63.     {
64.         Funcionario::mostraDados();
65.         cout << "\nCargo: " << cargo;
66.         cout << "\nSalario do gestor R$: " << salario;
67.         Estudante::mostraDadosEducacao();
68.     }
69. };
70. class Professor: private Funcionario, private Estudante
71. {
72. private:
73.     int cursos; // quantidade de cursos
74. public:
```

```
75. void lerDados()
76. {
77.     Funcionario::lerDados();
78.     cout << "Informe a quantidade de cursos: ";
79.     cin >> cursos;
80.     Estudante::lerDadosEducacao();
81. }
82. void mostraDados()
83. {
84.     Funcionario::mostraDados();
85.     cout << "\nQuantidade de cursos oferecidos: " << cursos;
86.     Estudante::mostraDadosEducacao();
87. }
88. };
89. class Tecnico: public Funcionario
90. { };
91.
92. int main()
93. {
94.     Gestor g1, g2;
95.     Professor p1;
96.     Tecnico t1;
97.
98.     cout << "\n\nInforme dados do gestor 1: ";
99.     g1.lerDados();
100.     cout << "\nInforme dados do gestor 2: ";
101.     g2.lerDados();
102.     cout << "\nInforme dados do professor 1: ";
103.     p1.lerDados();
104.     cout << "\nInforme dados do tecnico 1: ";
105.     t1.lerDados();
106.     cout << "\nDados do gestor 1";
107.     g1.mostraDados();
108.     cout << "\nDados do gestor 2";
109.     g2.mostraDados();
110.     cout << "\nDados do professor 1";
111.     p1.mostraDados();
112.     cout << "\nDados do tecnico 1";
113.     t1.mostraDados();
114.     cout << endl << endl;
115.     system("PAUSE");
116.     return 0;
117. }
```

Listagem 9.7

Note que as classes *Gestor* e *Professor* no programa da Listagem 9.7 são derivadas usando o especificador *private*, conforme linhas 47 e 70, respectivamente.

Nesse caso, você não precisa usar o especificador *public* porque os objetos das classes *Gestor* e *Professor* nunca chamam rotinas nas classes base *Funcionario* e *Estudante*. Todavia, a classe *Tecnico* deve ser derivada (*Funcionario*) usando o especificador *public* (linha 89) porque ela não possui funções-membros e utiliza as de *Funcionario*.

### 9.7.1. Ambiguidade em Herança Múltipla

Considere o caso em que duas classes base têm funções com o mesmo nome, ao passo que uma classe derivada não possui função com tal nome. Como os objetos da classe derivada têm acesso à função correta da classe base, o nome da função simplesmente não é suficiente para o compilador identificar qual das duas funções está sendo chamada. O que fazer?

Nesse caso, o problema pode ser resolvido usando-se o operador de resolução de escopo para especificar a que classe a função pertence (ou se é função-membro). Assim, você poderia especificar:

```
objZ.X::mostrarDados(); // mostrarDados() na classe X
enquanto
```

```
objZ.Y::mostrarDados(); // mostrarDados() na classe Y.
```

O operador de resolução de escopo resolve esta ambiguidade e satisfaz a necessidade do compilador.

## RESUMO

Neste capítulo, você teve oportunidade de estudar herança e explorar os conceitos de classe base e classe derivada através de vários exemplos. Depois do conceito de classes e objetos, herança é, provavelmente, o aspecto mais importante em programação orientada a objetos (POO). Herança é o processo de criar novas classes (classes derivadas) a partir das classes existentes (classes base). Adicionalmente, você aprendeu como utilizar os especificadores *public*, *protected* e *private*, além de explorar como a hierarquia de classes pode ser implementada. Finalmente, você estudou situações nas quais a herança múltipla pode ser empregada. Diversos exemplos foram usados para apresentação do conteúdo. No próximo capítulo, você estudará e explorará o uso de ponteiros e como eles podem ser usados na programação orientada a objetos, além de como as classes *Lista*, *Pilha* e *Fila* podem ser utilizadas.

## QUESTÕES

1. Explique o conceito de herança em programas orientados a objetos. Use exemplos para ilustrar sua resposta.
2. O que é reusabilidade? Como podemos obtê-la? Use exemplos para ilustrar sua resposta.
3. Qual a diferença entre classe base e classe derivada? Use um exemplo para ilustrar sua resposta.
4. Qual o significado dos especificadores de acesso public, protected e private? Em que situações eles devem ser utilizados? Use exemplos para ilustrar sua resposta.
5. O que é herança múltipla? Em que situações pode ser empregada? Use exemplos para ilustrar sua resposta.

## EXERCÍCIOS

1. Faça uma pesquisa visando responder à seguinte questão: em que situações é adequado usar herança? Apresente um exemplo para ilustrar sua resposta.
2. Escreva um programa que ilustre o uso de herança no qual você deve derivar uma classe Esfera a partir da classe Circulo.
3. Escreva um programa que implemente a classe Veiculo, tendo como classes derivadas VeiculoTerrestre e VeiculoAquatico. Adicionalmente, você deve também ter a classe Anfíbio, que é derivada de VeiculoTerrestre e VeiculoAquatico, portanto uma situação de herança múltipla.